面向过程与面向对象

面向过程

考虑我们要做一个宠物管理系统，需要存储宠物的各种数据，对宠物进行各种操作。具体要求如下：

存储数据：

- 宠物名称
- 宠物种类
- 宠物年龄

操作：

- 过了一年（宠物年龄+1）
- 向宠物打招呼（打印：你好，我<宠物年龄>岁的<宠物种类>，<宠物名称>）

结合C语言中结构体和函数的知识，我们很容易写出如下代码：

```c
#include <stdio.h>
#include <stdlib.h>

struct pet{
    char *name;
    char *kind;
    int age;

};
typedef struct pet *Pet;

Pet init_pet(char *name, char *kind, int age){
    Pet pet = (Pet)malloc(sizeof(struct pet));
    if(pet == NULL)return NULL;
    pet->name = name;
    pet->kind = kind;
    pet->age = age;
    return pet;
```

```
}

void spend_a_year(Pet pet){
    pet->age++;
    return;
}

void greet_to_pet(Pet pet){
    printf("Hello, my  %d-year-old %s, %s\n", pet->age, pet->kind,
pet->name);
    return;
}

void free_pet(Pet pet){
    free(pet);
    return;
}

int main(void){
    Pet jinmao = init_pet("jinmao", "dog", 10);
    greet_to_pet(jinmao);
    spend_a_year(jinmao);
    greet_to_pet(jinmao);
    free_pet(jinmao);
    return 0;
}
```

通过前面的一堆函数，我们可以进行任何相关的操作，增强了可扩展性。

我们将函数（操作）和字典（数据）分开了，这就是所谓的面向过程思想。所谓过程，就是操作数据的过程。我们在操作数据的时候，看重的是操作，数据只是操作的一个参数。

面向对象

还有人想，能不能将操作也放到结构体当中呢？这样的话，我们将操作视为一种特殊的数据，同时看重操作和数据。这就是所谓的面向对象的思想，如以下代码所示：

```
#include <stdio.h>
#include <stdlib.h>

struct pet{
```

```
    char *name;
    char *kind;
    int age;
    void (*spend)(Pet);
    void (*greet)(Pet);

};
typedef struct pet *Pet;

void spend_a_year(Pet);
void greet_to_pet(Pet);

Pet init_pet(char *name, char *kind, int age){
    Pet pet = (Pet)malloc(sizeof(struct pet));
    if(pet == NULL)return NULL;
    pet->name = name;
    pet->kind = kind;
    pet->age = age;
    pet->spend = spend_a_year;
    pet->greet = greet_to_pet;
    return pet;
}

void spend_a_year(Pet pet){
    pet->age++;
    return;
}

void greet_to_pet(Pet pet){
    printf("Hello, my  %d-year-old %s, %s\n", pet->age, pet->kind,
pet->name);
    return;
}

void free_pet(Pet pet){
    free(pet);
    return;
}

int main(void){
    Pet jinmao = init_pet("jinmao", "dog", 10);
    jinmao->greet(jinmao);
    jinmao->spend(jinmao);
```

```
    jinmao->greet(jinmao);
    free_pet(jinmao);
    return 0;
}
```

在这里面，我们向pet结构体加入了spend和greet函数指针，在初始化函数上设置了这些函数指针指向的函数，这样在main函数使用的时候就可以直接调用了。

我们可以发现，这样写的好处在于当我们使用这个pet结构体时，可以直接从pet内部调用函数，而不用关心这实际上调用了什么函数，这样可以提升其封装性。但面向对象的程序每创建一个结构体就会多几个函数指针，提高了空间占用。

我们称这样的结构体叫类，这样的结构体变量叫对象，结构体当中的函数指针叫方法，结构体当中的普通数据叫字段。

在现代面向对象语言中，`init_pet`中的设置函数指针部分、申请地址部分和`free_pet`中释放地址部分对应的操作均被编译器或解释器写出，用户无需显式地写出这些代码（`init_pet`的其他部分组成构造函数（对象创建之前执行），`free_pet`的其他部分组成析构函数（对象创建之后执行））。而且，为了提高其封装性，防止用户修改开发者不希望修改的东西，现代面向对象语言常提供`private`和`public`关键字，分别表示外部不能访问和外部可以访问。另外，用户不需要显式地指定其操作的对象，即不需要显式地将对象传递到函数里。例如，在C++语言中，定义这样一个类的代码如下：

```
#include <iostream>
using namespace std;

class Pet{
private:
    char *name;
    char *kind;
    int age;
public:
    Pet(char *n, char *k, int a):name(n), kind(k), age(a){}
    void spend(void){
        age++;
        return;
    }
    void greet(void){
        cout << "Hello, my " << age << "-year-old " << kind << ", "
<< name << "." << endl;
    }
};
```

```
int main(void){
    Pet jinmao("jinmao", "dog", 10);
    jinmao.greet();
    jinmao.spend();
    jinmao.greet();
    return 0;
}
```

继承

面向对象还可以提高程序的可扩展性，这就体现在继承上了。

继承，就是子类（又叫派生类）继承父类（又叫基类）的全部字段和方法，但可以新增一些字段和方法，简单来说，父类是子类的子集。

从另一个角度来说，如果对于每一个类型为`T1`的对象`O1`，都有类型为`T2`的`O2`，使得将程序中所有的`O1`替换为`O2`后，程序的行为没有任何变化，则称`T1`是`T2`的父类，这也符合我们之前说的定义。

如果我们想扩展这个宠物管理系统，让其可以管理鸟类，而鸟类可以飞行，我们需要加一个飞行功能（打印：“<宠物名字> is flying...”），可以修改代码如下：

```
#include <stdio.h>
#include <stdlib.h>

struct pet{
    char *name;
    char *kind;
    int age;
    void (*spend)(Pet);
    void (*greet)(Pet);

};
typedef struct pet *Pet;

struct bird{
    // pet的全部成员
    char *name;
    char *kind;
    int age;
    void (*spend)(Pet);
```

```
    void (*greet)(Pet);
    // 根据结构体在地址空间中的分布，这里必须将扩展的成员写在后面
    void (*fly)(Bird);
};
typedef struct bird *Bird;

void spend_a_year(Pet);
void greet_to_pet(Pet);
void fly_bird(Bird);

Pet init_pet(char *name, char *kind, int age){
    Pet pet = (Pet)malloc(sizeof(struct pet));
    if(pet == NULL)return NULL;
    pet->name = name;
    pet->kind = kind;
    pet->age = age;
    pet->spend = spend_a_year;
    pet->greet = greet_to_pet;
    return pet;
}

Bird init_bird(char *name, char *kind, int age){
    Pet child = init_pet(name, kind, age);
    Bird bird = (Bird)realloc(child, sizeof(struct bird));
    bird->fly = fly_bird; // 新增成员
    return bird;
}

void spend_a_year(Pet pet){
    pet->age++;
    return;
}

void greet_to_pet(Pet pet){
    printf("Hello, my  %d-year-old %s, %s\n", pet->age, pet->kind,
pet->name);
    return;
}

void free_pet(Pet pet){
    free(pet);
    return;
}
```

```
void fly_bird(Bird bird){
    printf("%s is flying\n", bird->name);
    return;
}

int main(void){
    Bird white = init_bird("xiaobai", "bird", 10);
    white->greet(white);
    white->spend(white);
    white->greet(white);
    white->fly(white);
    free_pet(white);
    return 0;
}
```

在这里我们可以发现，bird作为pet的子类，增加了`fly`这一方法，而原有的方法仍然可以使用，这就是说，bird的对象可以使用pet对象的方法，当程序需要bird对象按pet的使用方法使用时，就会将其视为pet对象。这体现了面向对象编程的可扩展性。

在现代面向对象编程语言中，继承操作更为简单，且成为了一种常见操作。这里不展开讲解。

重写

子类可以重新实现父类的方法，但是实现时函数指针的类型不变，即返回值类型、参数列表不变。

如我们想让给鸟打招呼时，说："My bird is <宠物名字>"，而不是正常宠物的输出，就可以重写`greet`方法，具体代码如下：

```
#include <stdio.h>
#include <stdlib.h>

struct pet{
    char *name;
    char *kind;
    int age;
    void (*spend)(Pet);
    void (*greet)(Pet);
```

```
};
typedef struct pet *Pet;

struct bird{
    // pet的全部成员
    char *name;
    char *kind;
    int age;
    void (*spend)(Pet);
    void (*greet)(Pet);
    // 根据结构体在地址空间中的分布，这里必须将扩展的成员写在后面
    void (*fly)(Bird);
};
typedef struct bird *Bird;

void spend_a_year(Pet);
void greet_to_pet(Pet);
void fly_bird(Bird);
void greet_bird(Bird);

Pet init_pet(char *name, char *kind, int age){
    Pet pet = (Pet)malloc(sizeof(struct pet));
    if(pet == NULL)return NULL;
    pet->name = name;
    pet->kind = kind;
    pet->age = age;
    pet->spend = spend_a_year;
    pet->greet = greet_to_pet;
    return pet;
}

Bird init_bird(char *name, char *kind, int age){
    Pet child = init_pet(name, kind, age);
    Bird bird = (Bird)realloc(child, sizeof(struct bird));
    bird->fly = fly_bird; // 新增成员
    bird->greet = greet_bird;  // 重写greet方法
    return bird;
}

void spend_a_year(Pet pet){
    pet->age++;
    return;
}
```

```
void greet_to_pet(Pet pet){
    printf("Hello, my  %d-year-old %s, %s\n", pet->age, pet->kind,
pet->name);
    return;
}

void free_pet(Pet pet){
    free(pet);
    return;
}

void greet_bird(Bird bird){
    printf("My bird is %s\n", bird->name);
    return;
}

void fly_bird(Bird bird){
    printf("%s is flying\n", bird->name);
    return;
}

int main(void){
    Bird white = init_bird("xiaobai", "bird", 10);
    white->greet(white);
    white->spend(white);
    white->greet(white);
    white->fly(white);
    free_pet(white);
    return 0;
}
```

在这里面，我们将`bird`类型的greet方法的实现函数改为`greet_bird`，实现了重写。

在现代面向对象语言中，重写的写法也更为简便，而且，为了保证封装性，子类只能通过super关键字调用父类的原方法。

抽象

如果我们在父类中不实现方法，而交给子类实现不同的方法，这样的方法就是抽象方法。如我们可以给宠物类抽象一个进食方法，具体实现交给子类来做，如鸟吃虫，狗吃肉。具体代码如下：

```
#include <stdio.h>
#include <stdlib.h>

struct pet{
    char *name;
    char *kind;
    int age;
    void (*spend)(Pet);
    void (*greet)(Pet);
    void (*eat)(Pet);

};
typedef struct pet *Pet;

struct bird{
    // pet的全部成员
    char *name;
    char *kind;
    int age;
    void (*spend)(Pet);
    void (*greet)(Pet);
    void (*eat)(Pet);
    // 根据结构体在地址空间中的分布，这里必须将扩展的成员写在后面
    void (*fly)(Bird);
};
typedef struct bird *Bird;

struct dog{
    // pet的全部成员
    char *name;
    char *kind;
    int age;
    void (*spend)(Pet);
    void (*greet)(Pet);
    void (*eat)(Pet);
};
typedef struct dog *Dog;
```

```c
void spend_a_year(Pet);
void greet_to_pet(Pet);
void fly_bird(Bird);
void greet_bird(Bird);
void eat_bird(Bird);
void eat_dog(Dog);

Pet init_pet(char *name, char *kind, int age){
    Pet pet = (Pet)malloc(sizeof(struct pet));
    if(pet == NULL)return NULL;
    pet->name = name;
    pet->kind = kind;
    pet->age = age;
    pet->spend = spend_a_year;
    pet->greet = greet_to_pet;
    pet->eat = NULL; // 抽象方法没有实现
    return pet;
}

Bird init_bird(char *name, char *kind, int age){
    Pet child = init_pet(name, kind, age);
    Bird bird = (Bird)realloc(child, sizeof(struct bird));
    bird->fly = fly_bird; // 新增成员
    bird->greet = greet_bird;  // 重写greet方法
    bird->eat = eat_bird;  // 实现抽象方法
    return bird;
}

Dog init_dog(char *name, char *kind, int age){
    Dog dog = (Dog)init_pet(name, kind, age);
    dog->eat = eat_dog;
    return dog;
}

void spend_a_year(Pet pet){
    pet->age++;
    return;
}

void greet_to_pet(Pet pet){
    printf("Hello, my  %d-year-old %s, %s\n", pet->age, pet->kind,
pet->name);
```

```
    return;
}

void free_pet(Pet pet){
    free(pet);
    return;
}

void greet_bird(Bird bird){
    printf("My bird is %s\n", bird->name);
    return;
}

void fly_bird(Bird bird){
    printf("%s is flying\n", bird->name);
    return;
}

void eat_bird(Bird bird){
    printf("eating bugs...\n");
    return;
}

void eat_dog(Dog dog){
    printf("eating meat...\n");
    return;
}

int main(void){
    Bird white = init_bird("xiaobai", "bird", 10);
    white->eat(white);
    Dog dog = init_dog("jinmao", "dog", 10);
    dog->eat(dog);
    free_pet(white);
    return 0;
}
```

在代码中，父类Pet的方法eat是抽象方法，父类中没有实现，而子类进行的实现。

在一些现代面向对象语言中，子类必须实现父类全部的抽象方法。

如果父类中的所有方法全是抽象方法，这个父类可以被视为接口。当外界需要用一个对象，而不知道这个对象具体是什么，只知道它们继承自同一个接口，那它就可以根据这个接口的方法进行使用。

写在最后

本文通过C语言带读者从底层认识面向过程和面向对象，而不是通过已有的面向对象语言实现，希望可以增强读者对面向对象的认识。