动态多态与静态多态

前言

今日的C++不再是个单纯的“带类的C”语言，它已经发展成为一个多种次语言所组成的语言集合，其中泛型编程与基于它的STL是C++发展中最为出彩的那部分。在面向对象C++编程中，多态是OO三大特性之一，这种多态称为运行期多态，也称为动态多态；在泛型编程中，多态基于template(模板)的具现化与函数的重载解析，这种多态在编译期进行，因此称为编译期多态或静态多态。在本文中，我们将了解：

1. 什么是运行期多态
2. 什么是编译期多态
3. 它们的优缺点在哪

运行期多态

运行期多态的设计思想要归结到类继承体系的设计上去。对于有相关功能的对象集合，我们总希望能够抽象出它们共有的功能集合，在基类中将这些功能声明为虚接口（虚函数），然后由子类继承基类去重写这些虚接口，以实现子类特有的具体功能。典型地我们会举下面这个例子：


```cpp
class Animal
{
public :
    virtual void shout() = 0;
};
class Dog :public Animal
{
public:
    virtual void shout(){ cout << "汪汪！"<<endl; }
};
class Cat :public Animal
{
public:
    virtual void shout(){ cout << "喵喵~"<<endl; }
};
```

```
class Bird : public Animal
{
public:
    virtual void shout(){ cout << "叽喳!"<<endl; }
};

int main()
{
    Animal * anim1 = new Dog;
    Animal * anim2 = new Cat;
    Animal * anim3 = new Bird;

   //藉由指针（或引用）调用的接口，在运行期确定指针（或引用）所指对象的真正类型，调用该类型对应的接口
    anim1->shout();
    anim2->shout();
    anim3->shout();

    //delete 对象
    ...
   return 0;
}
```

运行期多态的实现依赖于虚函数机制。当某个类声明了虚函数时，编译器将为该类对象安插一个虚函数表指针，并为该类设置一张唯一的虚函数表，虚函数表中存放的是该类虚函数地址。运行期间通过虚函数表指针与虚函数表去确定该类虚函数的真正实现。

运行期多态的优势还在于它使处理异质对象集合称为可能：

```
//我们有个动物园，里面有一堆动物
int main()
{
    vector<Animal*>anims;

    Animal * anim1 = new Dog;
    Animal * anim2 = new Cat;
    Animal * anim3 = new Bird;
    Animal * anim4 = new Dog;
    Animal * anim5 = new Cat;
    Animal * anim6 = new Bird;

    //处理异质类集合
    anims.push_back(anim1);
    anims.push_back(anim2);
    anims.push_back(anim3);
    anims.push_back(anim4);
    anims.push_back(anim5);
    anims.push_back(anim6);

    for (auto & i : anims)
    {
        i->shout();
    }
    //delete对象
    //...
    return 0;
}
```

总结：**运行期多态通过虚函数发生于运行期**

编译期多态

对模板参数而言，多态是通过模板具现化和函数重载解析实现的。**以不同的模板参数具现化导致调用不同的函数**，这就是所谓的编译期多态。
相比较于运行期多态，实现编译期多态的类之间并不需要成为一个继承体系，它们之间可以没有什么关系，但约束是它们都有相同的隐式接口。我们将上面的例子改写为:

```
class Animal
{
public :
    void shout() { cout << "发出动物的叫声" << endl; };
};
class Dog
{
public:
     void shout(){ cout << "汪汪！"<<endl; }
};
class Cat
{
public:
     void shout(){ cout << "喵喵~"<<endl; }
};
class Bird
{
public:
     void shout(){ cout << "叽喳!"<<endl; }
};
template <typename T>
void  animalShout(T & t)
{
    t.shout();
}
int main()
{
    Animal anim;
    Dog dog;
    Cat cat;
    Bird bird;

    animalShout(anim);
    animalShout(dog);
    animalShout(cat);
    animalShout(bird);

    getchar();
}
```

在编译之前，函数模板中t.shout()调用的是哪个接口并不确定。在编译期间，编译器推断出模板参数，因此确定调用的shout是哪个具体类型的接口。**不同的推断结果调用不同的函数**，这就是编译器多态。这类似于重载函数在编译器进行推导，以确定哪一个函数被调用。

动态多态与静态多态优缺点分析

运行期多态优点

1. OO设计中重要的特性，对客观世界直觉认识。
2. 能够处理同一个继承体系下的异质类集合。

运行期多态缺点

1. 运行期间进行虚函数绑定，提高了程序运行开销。
2. 庞大的类继承层次，对接口的修改易影响类继承层次。
3. 由于虚函数在运行期在确定，所以编译器无法对虚函数进行优化。
4. 虚表指针增大了对象体积，类也多了一张虚函数表，当然，这是理所应当值得付出的资源消耗，列为缺点有点勉强。

编译期多态优点

1. 它带来了泛型编程的概念，使得C++拥有泛型编程与STL这样的强大武器。
2. 在编译器完成多态，提高运行期效率。
3. 具有很强的适配性与松耦合性，对于特殊类型可由模板偏特化、全特化来处理。

编译期多态缺点

1. 程序可读性降低，代码调试带来困难。
2. 无法实现模板的分离编译，当工程很大时，编译时间不可小觑。
3. 无法处理异质对象集合。

关于显式接口与隐式接口

所谓的显式接口是指类继承层次中定义的接口或是某个具体类提供的接口，总而言之，我们能够在源代码中找到这个接口.显式接口以函数签名为中心，例如

```
void AnimalShot(Animal & anim)
{
    anim.shout();
}
```

我们称shout为一个显式接口。在运行期多态中的接口皆为显式接口。

而对模板参数而言，接口是隐式的，奠基于有效表达式。例如：

```
template <typename T>
void AnimalShot(T & anim)
{
    anim.shout();
}
```

对于anim来说，必须支持哪一种接口，要由模板参数执行于anim身上的操作来决定，在上面这个例子中，T必须支持shout()操作，那么shout就是T的一个隐式接口。