C++ STL萃取机制

CPP开发者 2020-12-06 19:40

(给CPP开发者加星标，提升C/C++技能)

> 来源：
> https://www.cnblogs.com/single-dont/p/11403807.html

我将从定义、技术实现、设问形式、实例总结来阐述我对于萃取机制的理解。

1.定义

traits中文意思是特性，它通过提取不同类的共性，使得可以统一处理。

2.技术实现

traits运用显式模板特殊化将代码中因为类型不同而发生变化的片段提取出来，用统一的接口来包装，并通过traits模板类公开的接口间接访问相应的类。

3.设问形式

问题1：什么是显式模板特殊化呢?

答：模板特殊化又分了一个偏特化（意思就是没有完全的特化）我们来看一段代码，

```
template<class T,class U>
// 基础模板类
class NumTraits
{};
//模板特化的格式
template<class T>
//偏特殊化
class NumTraits<IntArray>
{
public:
    typedef int resulttype;
```

```
    typedef int inputargtype;
};
template<class T>
class NumTraits
{};
//模板特化的格式
template<>
//特殊化
class NumTraits<IntArray>
{
public:
    typedef int resulttype;
    typedef int inputargtype;
};
```

问题2：用实例来展示一下为什么会使用萃取机制?

答：我会用（3步走）的代码来解释为什么需要使用萃取。

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

//①基本类写法
class IntArray
{
public:
    IntArray()
    {
        a = new int[10];
        for (int i = 0; i < 10; ++i)
        {
            a[i] = i + 1;
        }
    }
    ~IntArray()
    {
        delete[] a;
```

```
19      }
20
21      int GetSum(int times)
22  {
23          int sum = 0;
24          for (int i = 0; i < 10; ++i)
25              sum += a[i];
26          cout << "int sum=" << sum << endl;
27          return sum * times;
28      }
29  private:
30      int *a;
31  };
32  class FloatArray
33  {
34  public:
35      FloatArray()
36      {
37          f = new float[10];
38          for (int i = 1; i <= 10; ++i)
39          {
40              f[i - 1] = 1.0f / i;
41          }
42      }
43      ~FloatArray()
44      {
45          delete[] f;
46      }
47      float GetSum(float times)
48  {
49          float sum = 0.0f;
50          for (int i = 0; i < 10; i++)
51              sum += f[i];
52          cout << "float sum=" << sum << endl;
53          return sum * times;
54      }
55  private:
```

```
    float* f;
};
//②模板写法
template<class T>
class Apply
{
public:
    float GetSum(T& t, float inarg)
{
        return t.GetSum(inarg);
    }
};

//以上方法不能完全解决我们的问题(函数返回值固定，就会导致异常)
//③采用萃取机制：模板特化
template<class T>
class NumTraits
{};
//模板特化的格式
template<>
class NumTraits<IntArray>
{
public:
    typedef int resulttype;
    typedef int inputargtype;
};
template<>
class NumTraits<FloatArray>
{
public:
    typedef float resulttype;
    typedef float inputargtype;
};
template<class T>
class Apply2
{
public:
```

```
93      NumTraits<T>::resulttype GetSum(T& obj, NumTraits<T>::inputargtype i
94  {
95          return obj.GetSum(inputarg);
96      }
97  };
98  int main()
99  {
100     IntArray intary;
101     FloatArray floatary;
102     Apply<IntArray> ai;  //采用模板
103     Apply<FloatArray> af;  //采用模板
104     cout << "1整型数组的和3倍：" << ai.GetSum(intary, 3) << endl;
105     cout << "1浮点数组的和3.2倍：" << af.GetSum(floatary, 3.2f) << endl;
106     cout<<endl;
107     cout<<endl;
108     Apply2<IntArray> ai2;  //采用萃取
109     Apply2<FloatArray> af2; //采用萃取
110     cout << "2整型数组的和3倍：" <<ai2.GetSum(intary,3) << endl;
111     cout << "2浮点数组的和3.2倍：" << af2.GetSum(floatary,3.2f) << endl;
112     return 0;
113 }
```

4.实例总结

第①步：我们会发现代码冗余度很高，所以采用了第二种；

第②步：我们会发现在运用模板后，代码量是减少了，但是其类内部函数定义出现了固定形式的类型。若遇到复杂的问题，会导致数据的错误。

第③步：我们运用了traits机制，将根据不同类类型特化出相应的函数参数类型和返回值类型，这样就可以通过统一的接口，来实现不同的实例。

由此，萃取机制对我们编码的复用性，帮助还是很大的！！！！！！

参考书籍：《C++STL基础及应用》