第35章 可变参数模板

《现代C++语言核心特性解析》 谢丙堃

可变参数模板的概念和语法

- 类模板或者函数模板的形参个数可变。
- 典型的应用包括:
  - bind
  - tuple

可变参数模板的概念和语法

- 可变参数模板的语法和可变参数函数有一点相似之处就是都用到了省略号...

```
template<class ...Args>
void foo(Args ...args) {}

template<class ...Args>
class bar {
public:
    bar(Args ...args) {
        foo(args...);
    }
};
```

可变参数模板的概念和语法

语法	含义
class ...Args	类型模板形参包
Args ...args	函数形参包
args...	形参包展开
模式	实参展开的方法

形参包展开

- 包展开的场景包括：
  1. 表达式列表
  2. 初始化列表
  3. 基类描述
  4. 成员初始化列表
  5. 函数参数列表
  6. 模板参数列表
  7. 动态异常列表（C++17已经不在使用）
  8. lambda表达式捕获列表
  9. sizeof... 运算符
  10. 对齐运算符
  11. 属性列表

形参包展开

- 例子

```
template<class T, class U>
T baz(T t, U u) {
    std::cout << t << ":" << u << std::endl;
    return t;
}
template<class ...Args> void foo(Args ...args) {}
template<class ...Args>
class bar {
public:
    bar(Args ...args) {
        foo(baz(&args, args)...);
    }
};
int main() {
    bar<int, double, unsigned int> b(1, 5.0, 8);
}
```

形参包展开

- 模板实例化后相当于

```
class bar {
public:
    bar(int a1, double a2, unsigned int a3)
    {
        foo(baz(&a1, a1), baz(&a2, a2), baz(&a3, a3));
    }
};
```

sizeof...运算符

- 获取形参包中形参的个数，返回的类型是std::size_t

```
template <class... Args>
void foo(Args... args) {
    std::cout << "foo sizeof...(args) = "
              << sizeof...(args) << std::endl;
}
template <class... Args>
class bar {
public:
    bar() {
        std::cout << "bar sizeof...(Args) = "
                  << sizeof...(Args) << std::endl;
    }
};
foo();
foo(1, 2, 3);
bar<> b1;
bar<int, double> b2;
```

可变参数模板的递归计算

- 例子

```
template<class T>
T sum(T arg) {
    return arg;
}

template<class T1, class... Args>
auto sum(T1 arg1, Args ...args) {
    return arg1 + sum(args...);
}

int main() {
    std::cout << sum(1, 5.0, 11.7) << std::endl;
}
```

折叠表达式

- 使用折叠表达式简化递归计算

```
template<class... Args>
auto sum(Args ...args)
{
    return (args + ...);
}

int main()
{
    std::cout << sum(1, 5.0, 11.7) << std::endl;
}
```

折叠表达式

- 4种展开规则：一元向左折叠，一元向右折叠，二元向左折叠和二元向右折叠

表达式（*其中op是二元运算符）	折叠为
(args op ...)	(arg0 op (arg1 op . . . (argN-1 op argN)))
(... op args)	((((arg0 op arg1) op arg2) op . . .) op argN)
(args op ... op init)	(arg0 op (arg1 op . . . (argN-1 op (argN op init)))
(init op ... op args)	(((((init op arg0) op arg1) op arg2) op . . .) op argN)

折叠表达式

- 需要注意的结合顺序

```cpp
template <class... Args>
auto sum(Args... args)
{
    return (args + ...);
}

int main()
{
    std::cout
        << sum(
            std::string("hello "),
            "c++ ",
            "world") // 编译错误
        << std::endl;
}
```

```cpp
// 相当于代码:
std::string("hello ") + ("c++ " + "world")
```

折叠表达式

- 正确的写法

```
template<class... Args>
auto sum(Args ...args)
{
    return (... + args);
}
```

```
// 相当于代码:
(std::string("hello ") + "c++ ") + "world"
```

一元折叠表达式中空参数包的特殊处理

- 编译器很难确定折叠表达式最终的求值类型
- 特殊规则：
  1. 只有&&、||和,运算符能够在空参数包的一元折叠表达式中使用。
  2. 其中&&的求值结果一定为true,
  3. ||的求值结果一定为false,
  4. 而,的求值结果为void(),
  5. 其他运算符都是非法的。

using声明中的包展开

- 用于可变参数类模板派生于形参包的情况

```
template<class T> class base {
public:
    base() {}
    base(T t) : t_(t) {}
private:
    T t_;
};
template<class... Args>
class derived : public base<Args>... {
public:
    using base<Args>::base...;
};
derived<int, std::string, bool> d1 = 11;
derived<int, std::string, bool> d2 = std::string("hello");
derived<int, std::string, bool> d3 = true;
```

lambda表达式初始化捕获的包展开

- 捕获值，发生拷贝，存在效率问题

```
template<class F, class... Args>
auto delay_invoke(F f, Args... args) {
    return [f, args...]() -> decltype(auto) {
        return std::invoke(f, args...);
    };
}
```

lambda表达式初始化捕获的包展开

- 使用移动语义，代码复杂不易理解

```
template<class F, class... Args>
auto delay_invoke(F f, Args... args) {
    return[f = std::move(f), tup = std::make_tuple(std::move(args)...)]()
        -> decltype(auto) {
        return std::apply(f, tup);
    };
}
```

lambda表达式初始化捕获的包展开

- 使用lambda表达式初始化捕获的包展开

```
template<class F, class... Args>
auto delay_invoke(F f, Args... args) {
    return[f = std::move(f), ...args = std::move(args)]() -> decltype(auto) {
        return std::invoke(f, args...);
    };
}
```

感谢您的观看

欢迎关注