TJSW
Jan 19, 2020 · 6 min read

潮．C++ | CRTP 和靜態多型

多型 (Polymorphism) 是每位學習 C++ 的同學曾經的痛。而一般提到多型，多是指動態多型，也就是執行期 (run-time) 發生的：透過動態繫結 (dynamic binding) 來依實際類別而執行相對應成員函式。

但今天我們要更痛，帶各位同學看看編譯期 (compile-time) 就能讓編譯器自動推導決定出要執行的成員函式，也就是靜態多型 (static polymorphism)。

. . .

CRTP (Curiously Recurring Template Pattern)

先繞開正題提點必備的基礎知識 CRTP。中文有人翻做奇異遞迴樣版模式 (其實叫什麼一點都不重要。

看名字就知道是跟 C++ template 有著些姻緣的一個編程技巧。他的定義如下：

1. 一個衍生類別必須繼承一個樣版類別。
2. 該樣版類別必須使用該衍生類別做為其樣版參數。

寫成 code 就會像這樣子：

```
template<typename T>
class Base
{
};

class Derived : public Base<Derived>
{
};

int main()
{
  Derived d;
}
```

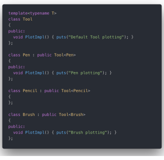
CRTP 的基本定義

. . .

靜態多型 (Static Polymorphism)

先營造一個畫畫的情境

大家對上面這 code 的模式有點印象之後，我們現在來做個鉛筆和原子筆的類比。我們來對 `Base` 這個基底類別加一些 member function，叫做 `PlotImpl()`：代表這個工具要怎麼做到「畫畫」這件事。



```
template<typename T>
class Tool
{
public:
  void PlotImpl() { puts("Default Tool plotting"); }
};

class Pen : public Tool<Pen>
{
public:
  void PlotImpl() { puts("Pen plotting"); }
};

class Pencil : public Tool<Pencil>
{
};

class Brush : public Tool<Brush>
{
public:
  void PlotImpl() { puts("Brush plotting"); }
};
```

我們做了三個類別 `Pencil`, `Pen`, `Brush`，都繼承了 `Tool` 這個基底類。並且 `Pen` 和 `Brush` 有自己的 `PlotImpl()` 方法各自實作怎麼畫畫，`Pencil` 則是繼承 `Tool` 的預設方法。

好戲上場

接下來，我們需要一個接點，讓外部使用者來使用這幾種工具「畫畫」。

我們對 `Tool` 加上一個 public 的接點：`Plot()`，讓大家統一呼叫 `Plot()` 就可以作到使用不同工具畫畫這件事。

```
template<typename T>
class Tool
{
public:
  void Plot() {
    static_cast<T*>(this)->PlotImpl();
  }
  void PlotImpl() { puts("Default Tool plotting"); }
};
```

靜態多型的精華所在

```
template<typename T>
void ToolHandler(Tool<T> &tool)
{
  tool.Plot();
}

int main()
{
  Pen pen;
  Pencil pencil;
  Brush brush;

  ToolHandler(pen);    // Pen plotting
  ToolHandler(pencil); // Default Tool plotting
  ToolHandler(brush);  // Brush plotting
}
```

而加上了 `Plot()` 這個接點之後的使用方法就像左邊這樣。`ToolHandler()` 這個函式就像我們在使用傳統的動態多型時喜歡傳入基底類別的指標一樣。不過在靜態多型的情境之下，我們傳入的會是基底類別樣版 `Tool<T>` 的參照。
就像動態多型一樣，我們一樣呼叫同樣是基底接口 `Plot()`。我們可以看到，只是因為傳入的實際類別不一樣 (`Pen`, `Pencil`, `Brush`)，同樣的一個 `Tool<T>` 的 `Plot()` 就可以做出不同的效果來。

靜態多型的關鍵在哪?

同學們一定很好奇我加上的那個 `Plot()` 為什麼可以完成這一切。道理很簡單，就是因為我們遵守了 CRTP 的定義規範。

動態多型的秘密藏在虛函式 virtual function / 虛表 VTable / 虛指標 VPtr 裡，在執行時期通過對實際的類別虛表和虛指針的查詢去執行真的所屬的 `PlotImpl()`，也因此要花更多的記憶體和執行時間達到這個目的。

靜態多型的奧妙在於由我們自己寫 code 從基底類別轉換成衍生類別呼叫到相對應的實作成員函式。

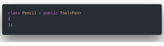
```
void Plot() {
  static_cast<T*>(this)->PlotImpl();
}
```

為什麼這是可行的呢？因為遵守 CRTP 規範之下，`Tool<T>` 的 `T`，一定是 `Tool<T>` 這個類別的衍生類。所以我們不需要使用 `dynamic_cast<T*>` 去執行期嘗試向下轉型 (down-cast)，我們確信一定可以轉下去，所以使用編譯期的 `static_cast<T*>` 告訴編譯器，我這裡就真的是 `Brush`。

如此一來，用不同的工具類別 (`Pen`, `Pencil`, `Brush`) 具現化 (template instantiation) `ToolHandler<T>` 這個函式樣板時，編譯器就能在編譯期知道具體是哪個類別呼叫的函式 `PlotImpl()`。相對於傳統多型在執行期才知道呼叫哪各函式，靜態多型在編譯時期就能知道，也因此這就是靜態多型的「靜態」由來。並且，不需要在執行期花時間去解析，相對來說快了許多。

怎麼避免手殘?

CRTP 可以帶來上面的好處，但前提是我們寫的 code 真的遵守了那個規範。要是我們手殘打錯字呢？比如這樣。



```
class Pencil : public Tool<Pen>
{
};
```

CRTP 要我們繼承的應該是 `Tool<Pencil>` 而不是手殘的 `Tool<Pen>`。如果手滑寫成這樣，在基類 `Plot()` 通過 `static_cast` 之後有可能有不預期行為發生的。

一個簡單的解法是這樣子的：因為我們的衍生類初始化時一定會先呼叫到基底類的 constructor，所以我們就將基底類的 constructor 封起來變成 private constructor，並且，利用 `friend` 裝飾限定只有繼承的子類 `T` 可以看到這個 private constructor。其它的類想要呼叫這個 private constructor 的就會在編譯期報錯。

```
template<typename T>
class Tool
{
public:
  void Plot() {
    static_cast<T*>(this)->PlotImpl();
  }
  void PlotImpl() { puts("Default Tool plotting"); }
private:
  Tool() {}
  friend T;
};
```

用左方的 code 這樣看回去解析一下我們手殘繼承的 `Tool<Pen>`。就會發現 `Tool<Pen>` 的 private constructor 只有他的 friend 也就是只有 `Pen` 看得到。

所以啦如果你想用 `Pencil` 去繼承 `Tool<Pen>`，編譯器就會直接報錯給你看。

```
error: call to implicitly-deleted default constructor
of 'Pencil'
Pencil pencil;
       ^


static_polymorphism2.cpp:22:16: note: default
constructor of 'Pencil' is implicitly deleted because
base class 'Tool<Pen>' has an inaccessible default
constructor


class Pencil : public Tool<Pen>
                      ^


1 error generated.
```

. . .

以上就是 CRTP 的簡單介紹還有一個簡單的應用啦，其實運用靜態多型還有一些變化來增加大型 C++ 程式的設計性封裝性以及擴展性，之後有緣再說。88