C++11使用using定义别名（替代typedef）

大家都知道，在 C++ 中可以通过 typedef 重定义一个类型：

```
typedef unsigned int uint_t;
```

被重定义的类型并不是一个新的类型，仅仅只是原有的类型取了一个新的名字。因此，下面这样将不是合法的函数重载：

```
void func(unsigned int);
void func(uint_t);  // error: redefinition
```

使用 typedef 重定义类型是很方便的，但它也有一些限制，比如，无法重定义一个模板。

想象下面这个场景：

```
typedef std::map<std::string, int> map_int_t;
// ...
typedef std::map<std::string, std::string> map_str_t;
// ...
```

我们需要的其实是一个固定以 std::string 为 key 的 map，它可以映射到 int 或另一个 std::string。然而这个简单的需求仅通过 typedef 却很难办到。

因此，在 C++98/03 中往往不得不这样写：

```
01.  template <typename Val>
02.  struct str_map
03.  {
04.      typedef std::map<std::string, Val> type;
05.  };
06.  // ...
07.  str_map<int>::type map1;
08.  // ...
```

一个虽然简单但却略显烦琐的 str_map 外敷类是必要的。这明显让我们在复用某些泛型代码时非常难受。

现在，在 C++11 中终于出现了可以重定义一个模板的语法。请看下面的示例：

```
template <typename Val>
using str_map_t = std::map<std::string, Val>;
// ...
str_map_t<int> map1;
```

这里使用新的 using 别名语法定义了 std::map 的模板别名 str_map_t。比起前面使用外敷模板加 typedef 构建的 str_map，它完全就像是一个新的 map 类模板，因此，简洁了很多。

实际上，using 的别名语法覆盖了 typedef 的全部功能。先来看看对普通类型的重定义示例，将这两种语法对比一下：

```
// 重定义unsigned int
typedef unsigned int uint_t;
using uint_t = unsigned int;
// 重定义std::map
typedef std::map<std::string, int> map_int_t;
using map_int_t = std::map<std::string, int>;
```

可以看到，在重定义普通类型上，两种使用方法的效果是等价的，唯一不同的是定义语法。

typedef 的定义方法和变量的声明类似：像声明一个变量一样，声明一个重定义类型，之后在声明之前加上 typedef 即可。这种写法凸显了 C/C++ 中的语法一致性，但有时却会增加代码的阅读难度。比如重定义一个函数指针时：

```
typedef void (*func_t)(int, int);
```

与之相比，using 后面总是立即跟随新标识符（Identifier），之后使用类似赋值的语法，把现有的类型（type-id）赋给新类型：

```
using func_t = void (*)(int, int);
```

从上面的对比中可以发现，C++11 的 using 别名语法比 typedef 更加清晰。因为 typedef 的别名语法本质上类似一种解方程的思路。而 using 语法通过赋值来定义别名，和我们平时的思考方式一致。

下面再通过一个对比示例，看看新的 using 语法是如何定义模板别名的。

```
/* C++98/03 */
template <typename T>
struct func_t
{
    typedef void (*type)(T, T);
};
```

```
07.  // 使用 func_t 模板
08.  func_t<int>::type xx_1;
09.  /* C++11 */
10.  template <typename T>
11.  using func_t = void (*)(T, T);
12.  // 使用 func_t 模板
13.  func_t<int> xx_2;
```

从示例中可以看出，通过 using 定义模板别名的语法，只是在普通类型别名语法的基础上增加 template 的参数列表。使用 using 可以轻松地创建一个新的模板别名，而不需要像 C++98/03 那样使用烦琐的外敷模板。

需要注意的是，using 语法和 typedef 一样，并不会创造新的类型。也就是说，上面示例中 C++11 的 using 写法只是 typedef 的等价物。虽然 using 重定义的 func_t 是一个模板，但 func_t<int> 定义的 xx_2 并不是一个由类模板实例化后的类，而是 void(*)(int, int) 的别名。

因此，下面这样写：

```
void foo(void (*func_call)(int, int));
void foo(func_t<int> func_call);  // error: redefinition
```

同样是无法实现重载的，func_t<int> 只是 void(*)(int, int) 类型的等价物。

细心的读者可以发现，using 重定义的 func_t 是一个模板，但它既不是类模板也不是函数模板（函数模板实例化后是一个函数），而是一种新的模板形式：模板别名（alias template）。

其实，通过 using 可以轻松定义任意类型的模板表达方式。比如下面这样：

```
template <typename T>
using type_t = T;
// ...
type_t<int> i;
```

type_t 实例化后的类型和它的模板参数类型等价。这里，type_t<int> 将等价于 int。