

08 | 易用性改进 I：自动类型推断和初始化

2019-12-13 吴咏炜

现代C++实战30讲

[进入课程 >](#)



讲述：吴咏炜

时长 15:26 大小 10.61M



你好，我是吴咏炜。

在之前的几讲里，我们已经多多少少接触到了一些 C++11 以来增加的新特性。下面的两讲，我会重点讲一下现代 C++（C++11/14/17）带来的易用性改进。

就像我们 [🔗\[开篇词\]](#) 中说的，我们主要是介绍 C++ 里好用的特性，而非让你死记规则。因此，这里讲到的内容，有时是一种简化的说法。对于日常使用，本讲介绍的应该能满足大部分的需求。对于复杂用法和边角情况，你可能还是需要查阅参考资料里的明细规则。

自动类型推断

如果要挑选 C++11 带来的最重大改变的话，自动类型推断肯定排名前三。如果只看易用性或表达能力的改进的话，那它就是“舍我其谁”的第一了。

auto

自动类型推断，顾名思义，就是编译器能够根据表达式的类型，自动决定变量的类型（从 C++14 开始，还有函数的返回类型），不再需要程序员手工声明 ([1])。但需要说明的是，auto 并没有改变 C++ 是静态类型语言这一事实——使用 auto 的变量（或函数返回值）的类型仍然是编译时就确定了，只不过编译器能自动帮你填充而已。

自动类型推断使得像下面这样累赘的表达式成为历史：

```
1 // vector<int> v;
2 for (vector<int>::iterator
3     it = v.begin(),
4     end = v.end();
5     it != end; ++it) {
6     // 循环体
7 }
```

 复制代码

现在我们可以直接写（当然，是不使用基于范围的 for 循环的情况）：

```
1 for (auto it = v.begin(), end = v.end();
2     it != end; ++it) {
3     // 循环体
4 }
```

 复制代码

不使用自动类型推断时，如果容器类型未知的话，我们还需要加上 typename（注意此处 const 引用还要求我们写 const_iterator 作为迭代器的类型）：

```
1 template <typename T>
2 void foo(const T& container)
3 {
4     for (typename T::const_iterator
5         it = v.begin(),
```

 复制代码

```
6     ...
7 }
```

如果 `begin` 返回的类型不是该类型的 `const_iterator` 嵌套类型的话，那实际上不用自动类型推断就没法表达了。这还真不是假设。比如，如果我们的遍历函数要求支持 C 数组的话，不用自动类型推断的话，就只能使用两个不同的重载：

 复制代码

```
1  template <typename T, std::size_t N>
2  void foo(const T (&a)[N])
3  {
4      typedef const T* ptr_t;
5      for (ptr_t it = a, end = a + N;
6           it != end; ++it) {
7          // 循环体
8      }
9  }
10
11 template <typename T>
12 void foo(const T& c)
13 {
14     for (typename T::const_iterator
15          it = c.begin(),
16          end = c.end();
17          it != end; ++it) {
18         // 循环体
19     }
20 }
```

如果使用自动类型推断的话，再加上 C++11 提供的全局 `begin` 和 `end` 函数，上面的代码可以统一成：

 复制代码

```
1  template <typename T>
2  void foo(const T& c)
3  {
4      using std::begin;
5      using std::end;
6      // 使用依赖参数查找 (ADL) ; 见 [2]
7      for (auto it = begin(c),
8           ite = end(c);
9           it != ite; ++it) {
10         // 循环体
11     }
```

```
11     }  
12 }  
13 </span class="orange">
```

从这个例子可见，自动类型推断不仅降低了代码的啰嗦程度，也提高了代码的抽象性，使我们可以用更少的代码写出通用的功能。

`auto` 实际使用的规则类似于函数模板参数的推导规则（[\[3\]](#)）。当你写了一个含 `auto` 的表达式时，相当于把 `auto` 替换为模板参数的结果。举具体的例子：

`auto a = expr;` 意味着用 `expr` 去匹配一个假想的 `template <typename T> f(T)` 函数模板，结果为值类型。

`const auto& a = expr;` 意味着用 `expr` 去匹配一个假想的 `template <typename T> f(const T&)` 函数模板，结果为常左值引用类型。

`auto&& a = expr;` 意味着用 `expr` 去匹配一个假想的 `template <typename T> f(T&&)` 函数模板，根据 [🔗\[第 3 讲\]](#) 中我们讨论过的转发引用和引用坍缩规则，结果是一个跟 `expr` 值类别相同的引用类型。

decltype

`decltype` 的用途是获得一个表达式的类型，结果可以跟类型一样使用。它有两个基本用法：

`decltype(变量名)` 可以获得变量的精确类型。

`decltype(表达式)`（表达式不是变量名，但包括 `decltype((变量名))` 的情况）可以获得表达式的引用类型；除非表达式的结果是个纯右值（prvalue），此时结果仍然是值类型。

如果我们有 `int a;`，那么：

`decltype(a)` 会获得 `int`（因为 `a` 是 `int`）。

`decltype((a))` 会获得 `int&`（因为 `a` 是 `lvalue`）。

`decltype(a + a)` 会获得 `int` (因为 `a + a` 是 `prvalue`) 。

decltype(auto)


通常情况下，能写 `auto` 来声明变量肯定是件比较轻松的事。但这儿有个限制，你需要在写下 `auto` 时就决定你写下的是个引用类型还是值类型。根据类型推导规则，`auto` 是值类型，`auto&` 是左值引用类型，`auto&&` 是转发引用（可以是左值引用，也可以是右值引用）。使用 `auto` 不能通用地根据表达式类型来决定返回值的类型。不过，`decltype(expr)` 既可以是值类型，也可以是引用类型。因此，我们可以这么写：

```
1 decltype(expr) a = expr;
```

 复制代码

这种写法明显不能让人满意，特别是表达式很长的情况（而且，任何代码重复都是潜在的问题）。为此，C++14 引入了 `decltype(auto)` 语法。对于上面的情况，我们只需要像下面这样写就行了。

```
1 decltype(auto) a = expr;
```

 复制代码

这种代码主要用在通用的转发函数模板中：你可能根本不知道你调用的函数是不是会返回一个引用。这时使用这种语法就会方便很多。

函数返回值类型推断

从 C++14 开始，函数的返回值也可以用 `auto` 或 `decltype(auto)` 来声明了。同样的，用 `auto` 可以得到值类型，用 `auto&` 或 `auto&&` 可以得到引用类型；而用 `decltype(auto)` 可以根据返回表达式通用地决定返回的是值类型还是引用类型。

和这个形式相关的有另外一个语法，后置返回值类型声明。严格来说，这不算“类型推断”，不过我们也放在一起讲吧。它的形式是这个样子：

```
1 auto foo(参数) -> 返回值类型声明
```

 复制代码

```
2 {  
3     // 函数体  
4 }
```

通常，在返回类型比较复杂、特别是返回类型跟参数类型有某种推导关系时会使用这种语法。以后我们会讲到一些实例。今天暂时不多讲了。

类模板的模板参数推导


如果你用过 `pair` 的话，一般都不会使用下面这种形式：

```
1 pair<int, int> pr{1, 42};
```

 复制代码

使用 `make_pair` 显然更容易一些：


```
1 auto pr = make_pair(1, 42);
```

 复制代码

这是因为函数模板有模板参数推导，使得调用者不必手工指定参数类型；但 C++17 之前的类模板却没有这个功能，也因而催生了像 `make_pair` 这样的工具函数。

在进入了 C++17 的世界后，这类函数变得不必要了。现在我们可以直接写：

```
1 pair pr{1, 42};
```

 复制代码

生活一下子变得简单多了！

在初次见到 `array` 时，我觉得它的主要缺点就是不能像 C 数组一样自动从初始化列表来推断数组的大小了：

[复制代码](#)

```
1 int a1[] = {1, 2, 3};
2 array<int, 3> a2{1, 2, 3}; // 啰嗦
3 // array<int> a3{1, 2, 3}; 不行
```

这个问题在 C++17 里也是基本不存在的。虽然不能只提供一个模板参数，但你可以两个参数全都不写 🤖：

[复制代码](#)

```
1 array a{1, 2, 3};
2 // 得到 array<int, 3>
```

这种自动推导机制，可以是编译器根据构造函数来自动生成：

[复制代码](#)

```
1 template <typename T>
2 struct MyObj {
3     MyObj(T value);
4     ...
5 };
6
7 MyObj obj1{string("hello")};
8 // 得到 MyObj<string>
9 MyObj obj2{"hello"};
10 // 得到 MyObj<const char*>
```

也可以是手工提供一个推导向导，达到自己需要的效果：

[复制代码](#)


```
1 template <typename T>
2 struct MyObj {
3     MyObj(T value);
4     ...
5 };
6
7 MyObj(const char*) -> MyObj<string>;
8
9 MyObj obj{"hello"};
10 // 得到 MyObj<string>
```


更多的技术细节请参见参考资料 [4]。

结构化绑定


在讲关联容器的时候我们有过这样一个例子：

```
1 multimap<string, int>::iterator
2   lower, upper;
3 std::tie(lower, upper) =
4   mmp.equal_range("four");
```

 复制代码

这个例子里，返回值是个 `pair`，我们希望用两个变量来接收数值，就不得不声明了两个变量，然后使用 `tie` 来接收结果。在 C++11/14 里，这里是没法使用 `auto` 的。好在 C++17 引入了一个新语法，解决了这个问题。目前，我们可以把上面的代码简化为：

```
1 auto [lower, upper] =
2   mmp.equal_range("four");
```

 复制代码

这个语法使得我们可以用 `auto` 声明变量来分别获取 `pair` 或 `tuple` 返回值里各个子项，可以让代码的可读性更好。

关于这个语法的更多技术说明，请参见参考资料 [5]。

列表初始化

在 C++98 里，标准容器比起 C 风格数组至少有一个明显的劣势：不能在代码里方便地初始化容器的内容。比如，对于数组你可以写：

```
1 int a[] = {1, 2, 3, 4, 5};
```

 复制代码

而对于 `vector` 你却得写：

[复制代码](#)

```
1 vector<int> v;  
2 v.push(1);  
3 v.push(2);  
4 v.push(3);  
5 v.push(4);  
6 v.push(5);
```

这样真是又啰嗦，性能又差，显然无法让人满意。于是，C++ 标准委员会引入了列表初始化，允许以更简单的方式来初始化对象。现在我们初始化容器也可以和初始化数组一样简单了：

[复制代码](#)

```
1 vector<int> v{1, 2, 3, 4, 5};
```

同样重要的是，这不是对标准库容器的特殊魔法，而是一个通用的、可以用于各种类的方法。从技术角度，编译器的魔法只是对 {1, 2, 3} 这样的表达式自动生成一个初始化列表，在这个例子里其类型是 `initializer_list<int>`。程序员只需要声明一个接受 `initializer_list` 的构造函数即可使用。从效率的角度，至少在动态对象的情况下，容器和数组也并无二致，都是通过拷贝（构造）进行初始化。

对于初始化列表在构造函数外的用法和更多的技术细节，请参见参考资料 [\[6\]](#)。

统一初始化

你可能已经注意到了，我在代码里使用了大括号 {} 来进行对象的初始化。这当然也是 C++11 引入的新语法，能够代替很多小括号 () 在变量初始化时使用。这被称为统一初始化 (uniform initialization)。

大括号对于构造一个对象而言，最大的好处是避免了 C++ 里“最令人恼火的语法分析” (the most vexing parse)。我也遇到过。假设你有一个类，原型如下：


[复制代码](#)

```
1 class utf8_to_wstring {  
2 public:  
3     utf8_to_wstring(const char*);
```

```
4 operator wchar_t*();  
5 };
```


然后你在 Windows 下想使用这个类来帮助转换文件名，打开文件：

```
1 ifstream ifs(  
2     utf8_to_wstring(filename));
```

 复制代码

你随后就会发现，`ifs` 的行为无论如何都不正常。最后，要么你自己查到，要么有人告诉你，上面这个写法会被编译器认为是和下面的写法等价的：

```
1 ifstream ifs(  
2     utf8_to_wstring filename);
```

 复制代码

换句话说，编译器认为你是声明了一个叫 `ifs` 的函数，而不是对象！

如果你把任何一对小括号替换成大括号（或者都替换，如下），则可以避免此类问题：

```
1 ifstream ifs{  
2     utf8_to_wstring{filename}};
```

 复制代码

推而广之，你几乎可以在所有初始化对象的地方使用大括号而不是小括号。它还有一个附带的特点：当一个构造函数没有标成 `explicit` 时，你可以使用大括号不写类名来进行构造，如果调用上下文要求那类对象的话。如：

```
1 Obj getObj()  
2 {  
3     return {1.0};  
4 }
```

 复制代码

如果 Obj 类可以使用浮点数进行构造的话，上面的写法就是合法的。如果有无参数、多参数的构造函数，也可以使用这个形式。除了形式上的区别，它跟 `Obj(1.0)` 的主要区别是，后者可以用来调用 `Obj(int)`，而使用大括号时编译器会拒绝“窄”转换，不接受以 `{1.0}` 或 `Obj{1.0}` 的形式调用构造函数 `Obj(int)`。

这个语法主要的限制是，如果一个构造函数既有使用初始化列表的构造函数，又有不使用初始化列表的构造函数，那编译器会**千方百计**地试图调用使用初始化列表的构造函数，导致各种意外。所以，如果给一个推荐的话，那就是：

如果一个类没有使用初始化列表的构造函数时，初始化该类对象可全部使用统一初始化语法。


如果一个类有使用初始化列表的构造函数时，则只应用在初始化列表构造的情况。

关于这个语法的更多详细用法讨论，请参见参考资料 [\[7\]](#)。

类数据成员的默认初始化

按照 C++98 的语法，数据成员可以在构造函数里进行初始化。这本身不是问题，但实践中，如果数据成员比较多、构造函数又有多个的话，逐个去初始化是个累赘，并且很容易在增加数据成员时漏掉在某个构造函数中进行初始化。为此，C++11 增加了一个语法，允许在声明数据成员时直接给予一个初始化表达式。这样，当且仅当构造函数的初始化列表中不包含该数据成员时，这个数据成员就会自动使用初始化表达式进行初始化。

这个句子有点长。我们看个例子：

 复制代码

```
1 class Complex {
2 public:
3     Complex()
4         : re_(0) , im_(0) {}
5     Complex(float re)
6         : re_(re), im_(0) {}
7     Complex(float re, float im)
8         : re_(re) , im_(im) {}
9     ...
10
11 private:
12     float re_;
```

```
13     float im_;  
14 };
```

假设由于某种原因，我们不能使用缺省参数来简化构造函数，我们可以用什么方式来优化上面这个代码呢？

使用数据成员的默认初始化的话，我们就可以这么写：

```
1 class Complex {  
2 public:  
3     Complex() {}  
4     Complex(float re) : re_(re) {}  
5     Complex(float re, float im)  
6         : re_(re) , im_(im) {}  
7  
8 private:  
9     float re_{0};  
10    float im_{0};  
11 };
```

 复制代码

第一个构造函数没有任何初始化列表，所以类数据成员的初始化全部由默认初始化完成，`re_` 和 `im_` 都是 0。第二个构造函数提供了 `re_` 的初始化，`im_` 仍由默认初始化完成。第三个构造函数则完全不使用默认初始化。

内容小结

在本讲中，我们介绍了现代 C++ 引入的几个易用性改进：自动类型推断，初始化列表，及类数据成员的默认初始化。使用这些特性非常简单，可以立即简化你的 C++ 代码，而不会引入额外的开销。唯一的要求只是你不要再使用那些上古时代的老掉牙编译器了.....

课后思考

你使用过现代 C++ 的这些特性了吗？如果还没有的话，哪些特性你打算在下一个项目里开始使用？

欢迎留言来分享你的看法。

参考资料

[1] cppreference.com, “Placeholder type specifiers” .

[🔗 https://en.cppreference.com/w/cpp/language/auto](https://en.cppreference.com/w/cpp/language/auto)

[1a] cppreference.com, “占位符类型说明符” .

[🔗 https://zh.cppreference.com/w/cpp/language/auto](https://zh.cppreference.com/w/cpp/language/auto)

[2] Wikipedia, “Argument-dependent name lookup” .

[🔗 https://en.wikipedia.org/wiki/Argument-dependent_name_lookup](https://en.wikipedia.org/wiki/Argument-dependent_name_lookup)

[2a] 维基百科, “依赖于实参的名字查找” . [🔗 https://zh.wikipedia.org/zh-cn/ 依赖于实参的名字查找](https://zh.wikipedia.org/zh-cn/依赖于实参的名字查找)

[3] cppreference.com, “Template argument deduction” .

[🔗 https://en.cppreference.com/w/cpp/language/template_argument_deduction](https://en.cppreference.com/w/cpp/language/template_argument_deduction)

[3a] cppreference.com, “模板实参推导” .

[🔗 https://zh.cppreference.com/w/cpp/language/template_argument_deduction](https://zh.cppreference.com/w/cpp/language/template_argument_deduction)

[4] cppreference.com, “Class template argument deduction” .

[🔗 https://en.cppreference.com/w/cpp/language/class_template_argument_deduction](https://en.cppreference.com/w/cpp/language/class_template_argument_deduction)

[4a] cppreference.com, “类模板实参推导” .

[🔗 https://zh.cppreference.com/w/cpp/language/class_template_argument_deduction](https://zh.cppreference.com/w/cpp/language/class_template_argument_deduction)

[5] cppreference.com, “Structured binding declaration” .

[🔗 https://en.cppreference.com/w/cpp/language/structured_binding](https://en.cppreference.com/w/cpp/language/structured_binding)

[5a] cppreference.com, “结构化绑定声明” .

[🔗 https://zh.cppreference.com/w/cpp/language/structured_binding](https://zh.cppreference.com/w/cpp/language/structured_binding)

[6] cppreference.com, "std::initializer_list" .

https://en.cppreference.com/w/cpp/utility/initializer_list

[6a] cppreference.com, "std::initializer_list" .

https://en.cppreference.com/w/cpp/utility/initializer_list

[7] Scott Meyers, Effective Modern C++, item 7. O' Reilly Media, 2014. 有中文版 (高博译, 中国电力出版社, 2018 年)

点击查看 

打卡学习 C++ 拒绝从入门到放弃



PC 端用户扫码参与



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 07 | 迭代器和好用的新for循环

下一篇 09 | 易用性改进 II：字面量、静态断言和成员函数说明符

精选留言 (12)

 写留言



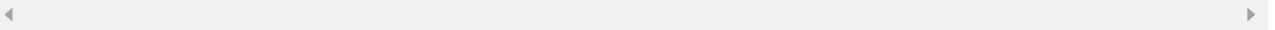
中年男子

2019-12-14

建议各位如果文章中有没看懂的地方，去看看老师在文末的参考资料，这些也都是好东西

作者回复: 识货。😊

毕竟这个专栏的篇幅是 30 讲，不是 60 讲或 100 讲啊。



👍 2



花晨少年

2019-12-14

如果一个类有使用初始化列表的构造函数时，则只应用在初始化列表构造的情况。是说{1.0}这种形式只用在初始化列表构造的情况吗？什么是初始化列表构造的情况？不明白

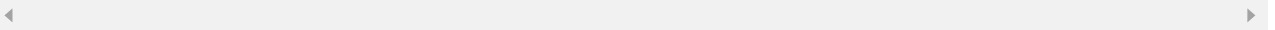
作者回复: 是说如果一个类Obj既有：

```
Obj(initializer_list<int>);
```

又有：

```
Obj(double);
```

那你想调用后面那个构造函数，就别用 Obj{1.0} 这种形式，而用 Obj(1.0)。



👍 1



lyfei

2019-12-18

谢谢老师上次耐心的回复。

上次问题: 就是我在编译文稿中的推导向导的时候，提示错误：class template argument deduction failed:

```
MyObj(const char*) -> MyObj<std::string>;...
```

展开 ▾

作者回复: “龟腩”而已。参考资料 [4a] 里有的：

“用户定义推导指引必须指名一个类模板，且必须在类模板的同一语义作用域（可以是命名空间或外围类）中引入，而且对于成员类模板，必须拥有同样的访问，但推导指引不成为该作用域的成员。”



**lyfei**

2019-12-17

老师您好，我对下面这两个疑惑有些不解：

1. 就是我在编译文稿中的推导向导的时候，提示错误：class template argument deduction failed:

```
MyObj(const char*) -> MyObj<std::string>;
```

2. MyObj obj2{"hello"}; 这句话编译器自动推断出来的类型是：MyObj<char const*> ...

展开 ▾

作者回复：1. 文稿中不是完整的代码。我拿下面的完整代码测试是没有问题的：

```
#include <string>
```

```
using namespace std;
```

```
template <typename T>
```

```
struct MyObj {
```

```
    MyObj(T value)
```

```
        : value_(value) {}
```

```
    T value_;
```

```
};
```

```
MyObj(const char*) -> MyObj<string>;
```

```
int main()
```

```
{
```

```
    MyObj obj{"hello"};
```

```
}
```

2. const char* 就是 char const*，没有区别，是同一个东西，都是指向常字符的指针（指针指向的内容不可更改）。如果写成 char* const，那就不一样了——那是指向字符的常指针（指针本身不可更改，指向的内容可更改）。

**lyfei**

2019-12-17

老师您好，就是您文稿中的代码：

```
template <typename T>void foo(const T& c){ using std::begin; using std::end; // 使
```

用依赖参数查找 (ADL) ; 见 `[2] for (auto it = begin(c), ite = end(c); it != ite; ++it) { // 循环体 }`

我这里有疑惑，就是这里哪一句可以体现出ADL呀？（ADL我理解的是：编译器根据传...
展开

作者回复: 就是 `begin` 和 `end`。对象 `c` 所属类型所在的名空间里的这两个函数将被优先使用。



墨梵

2019-12-16

吴老有没有打算在网络编程和多线程这几个点上做一个剖析呢？

作者回复: 内容太多，这两个话题都会讨论到，但可能不会太深。具体参见目录。第 19、20、27 讲。



海生

2019-12-14

目前的话，`C++11`用的比较多，`C++17`估计大多数以前的老代码都是不支持的。`bind` 和 `functional` 实现类似Java的面向interface编程的方式比`auto`应用影响更大吧，毕竟`C++`是强语言，类型声明是应该的义务。老师后续能不能讲讲进程编程和多线程，`CAS`, `disruptor`类的。`algorithm` 库里面的东西也很多，值得讲讲。

展开

作者回复: `C++`，不是 `C`。这是两种不同的语言。

这个专栏讲的内容是比较确定的，你可以看目录。后面我会讲到函数式和多线程，`CAS` 可以稍微讲一下。其他内容大概不会覆盖到了.....

算法本身很零散，又不算难理解。在我讲到的个别算法之外，其他大家自己看应该不会很复杂。



小一日一

2019-12-13

由于维护优化的是公司10年前的老代码，`gcc`版本停留在了古老的4.8.5，我在写新项目和特性时只能使用`C++11`特性，老师今天讲的`C++11`引入的所有特性我都在使用，如数据

成员的默认初始化，统一初始化，列表初始化，后置返回值类型，decltype，auto，而C++14和17引入的结构化绑定，类模板的模板参数推导，decltype(auto)无法使用，只有望洋兴叹，留口水的份。...

展开 ▾

作者回复: 先升级编译器，解决任何编译问题，再用测试来确保没有问题。

编译期和语言对向后兼容性一直保持得很好的，原则上不应该有问题。不能太保守了。（但也不要激进地每个新版本都升。）

◀ ▶

💬 4



皓首不倦

2019-12-13

老师您好 我记得以前自己对auto的推导进行学习的时候 想看推导出的到底是什么类型 需要用boost库的一些特殊api 才行 auto推出来到底什么时候带引用 什么时候不带引用有时记不清楚 希望能直接把auto推出来的类型名字包括带不带引用符号打出来看下 请问下只用标准库的api 的话 有什么方便的方法能把一个变量的完整类型信息打印出来看吗

展开 ▾

作者回复: Boost 也没什么特别神秘的方法吧。不用 Boost，方法也应该相似的。

我个人一般用 Scott Meyers 教的一个办法：

```
#define TYPE_DISPLAY(var) \
    static type_displayer<decltype(var)> type_display_test
```

```
template <typename T> // declaration only for type_displayer;
class type_displayer;
```

用的时候，就写 TYPE_DISPLAY(变量名字);。

◀ ▶

💬 2



Cheng

2019-12-13

```
for(auto &it : list)
{
```

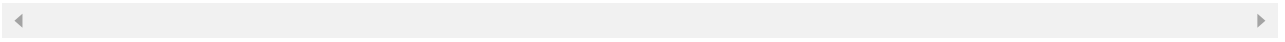
```
}
```

这个用法不知后面是否有讲到?

展开 ▾

作者回复: 这个已经讲到过了。不会再单独讲。

你上面的变量命名有问题, 会让人误以为 `it` 是个迭代器。它只是元素的运用, 并不是迭代器。



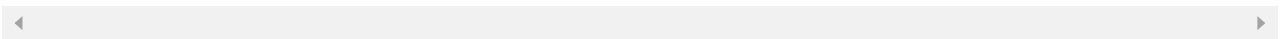
hello world

2019-12-13

初始化列表那里还没怎么看明白唉, 还是得多补习补习

展开 ▾

作者回复: 多自己试验例子来体会一下。



我叫bug谁找我

2019-12-13

想知道`auto`到底什么情况下用, 什么情况下不要用`auto`, 用多了会不会造成阅读困难

作者回复: 代码怎么看起来好看怎么用。 😊

