Chapter22 并行STL算法

为了从现代的多核体系中受益，C++17标准库引入了并行STL算法来使用多个线程并行处理元素。

许多算法扩展了一个新的参数来指明是否要并行运行算法（当然，没有这个参数的旧版本仍然受支持）。另外，还多了一些专为并行编程补充的新算法。

一个简单的计时器辅助类

对于这一章中的示例，有时我们需要一个计时器来测量算法的速度。因此，我们使用了一个简单的辅助类，它会初始化一个计时器并提供 `printDiff()` 来打印出消耗的毫秒数并重新初始化计时器：

```
#ifndef TIMER_HPP
#define TIMER_HPP

#include <iostream>
#include <string>
#include <chrono>

/*****************************************
* timer to print elapsed time
```

```
*****************************************/

class Timer {
private:
    std::chrono::steady_clock::time_point last;
public:
    Timer() : last{std::chrono::steady_clock::now()} {
    }
    void printDiff(const std::string& msg = "Timer diff: ") {
        auto now{std::chrono::steady_clock::now()};
        std::chrono::duration<double, std::milli> diff{now - last};
        std::cout << msg << diff.count() << "ms\n";
        last = std::chrono::steady_clock::now();
    }
};

#endif // TIMER_HPP
```

22.1 使用并行算法

让我们从一些示例程序开始，这些程序演示了怎么让现有算法并行运行和使用新的并行算法。

22.1.1 使用并行 for_each()

这是第一个例子，简单地演示了并行运行标准算法 `for_each()`：

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <numeric>
#include <execution>    // for 执行策略
#include <cmath>        // for sqrt()
#include "timer.hpp"

int main()
{
    int numElems = 1000;

    struct Data {
        double value;   // 初始值
        double sqrt;    // 并行计算平方根
    };

    // 初始化numElems个还没有计算平方根的值
```

```
    std::vector<Data> coll;
    coll.reserve(numElems);
    for (int i = 0; i < numElems; ++i) {
        coll.push_back(Data{i * 4.37, 0});
    }

    // 并行计算平方根
    for_each(std::execution::par,
             coll.begin(), coll.end(),
             [] (auto& val) {
                 val.sqrt = std::sqrt(val.value);
             });
}
```

如你所见，使用并行算法是很简单的：

- 包含头文件 `<execution>`
- 像通常调用算法一样进行调用，只不过需要添加一个第一个参数 `std::execution::par` ，这样我们就可以要求算法在并行模式下运行：

```
#include <algorithm>
#include <execution>
...
for_each(std::execution::par,
         coll.begin(), coll.end(),
         [] (auto& val) {
             val.sqrt = std::sqrt(val.value);
         });
```

像通常一样，这里的 `coll` 可以是任何范围。然而，注意所有的并行算法要求迭代器至少是前向迭代器（我们会在不同的线程中迭代范围内的元素，如果两个迭代器不能迭代相同的值这将毫无意义）。

并行算法实际运行的方式是实现特定的。当然，使用多线程不一定能加快速度，因为启动和控制多线程也会消耗时间。

性能提升

为了查明是否应该使用和应该在什么时候使用并行算法，让我们把示例修改为下面这样：

```
#include <vector>
#include <iostream>
```

```cpp
#include <algorithm>
#include <numeric>
#include <execution>    // for 执行策略
#include <cstdlib>      // for atoi()
#include "timer.hpp"

int main(int argc, char* argv[])
{
    // 从命令行读取numElems（默认值：1000）
    int numElems = 1000;
    if (argc > 1) {
        numElems = std::atoi(argv[1]);
    }

    struct Data {
        double value;   // 初始值
        double sqrt;    // 并行计算平方根
    };

    // 初始化numElems个还没有计算平方根的值：
    std::vector<Data> coll;
    coll.reserve(numElems);
    for (int i = 0; i < numElems; ++i) {
        coll.push_back(Data{i * 4.37, 0});
    }

    // 循环来重复测量
    for (int i{0}; i < 5; ++i) {
        Timer t;
        // 顺序执行：
        for_each(std::execution::seq,
                 coll.begin(), coll.end(),
                 [] (auto& val) {
                     val.sqrt = std::sqrt(val.value);
                 });
        t.printDiff("sequential: ");

        // 并行执行
        for_each(std::execution::par,
                 coll.begin(), coll.end(),
                 [] (auto& val) {
                     val.sqrt = std::sqrt(val.value);
                 });
        t.printDiff("parallel: ");
        std::cout << '\n';
    }
}
```

关键的修改点在于：

- 通过命令行参数，我们可以传递要操作的元素的数量。
- 我们使用了类 `Timer` 来测量算法运行的时间。
- 我们在循环中重复了多次测量，可以让结果更准确。

结果主要依赖于硬件、使用的C++编译器、使用的C++库。在我的笔记本上（使用Visual C++，CPU是2核心超线程的Intel i7），可以得到如下结果：

- 只有100个元素时，串行算法明显快的多（快10倍以上）。这是因为启动和管理线程占用了太多时间，对于这么少的元素来说完全不值得。
- 10,000个元素时基本相当。
- 1,000,000个元素时并行执行快了接近3倍。

再强调一次，没有通用的方法来判断什么场景什么时间值得使用并行算法，但这个示例演示了即使只是非平凡的数字操作，也可能值得使用它们。

值得使用并行算法的关键在于：

- 操作很长（复杂）
- 有很多很多元素

例如，使用算法 `count_if()` 的并行版本来统计vector中偶数的数量永远都不值得，即使有1,000,000,000个元素：

```
auto num = std::count_if(std::execution::par,       // 执行策略
                         coll.begin(), coll.end(),  // 范围
                         [] (int elem) {            // 判断准则
                             return elem % 2 == 0;
                         });
```

事实上，对于一个只有快速的判断式的简单算法（比如这个例子），并行执行永远会更慢。适合使用并行算法的场景应该是：对每个元素的处理需要消耗很多的时间并且处理过程需要独立于其他元素的处理。

然而，你并不能事先想好一切，因为什么时候如何使用并行线程取决于C++标准库的实现。事实上，你不能控制使用多少个线程，具体的实现也可能只对一定数量的元素使用多线程。

请在你的目标平台上自行测试适合的场景。

22.1.2 使用并行 `sort()`

排序是另一个使用并行算法的例子。因为排序准则对每个元素都会调用不止一次，所以你可以节省很多时间。

考虑像下面这样初始化一个string的vector：

```
std::vector<std::string> coll;
for (int i = 0; i < numElems / 2; ++i) {
    coll.emplace_back("id" + std::to_string(i));
    coll.emplace_back("ID" + std::to_string(i));
}
```

也就是说，我们创建了一个vector，其元素为以 `"id"` 或 `"ID"` 开头之后紧跟一个整数的字符串：

```
id0 ID0 id1 ID1 id2 ID2 id3 ... id99 ID99 id100 ID100 ...
```

像往常一样，我们可以顺序排序元素：

```
sort(coll.begin(), coll.end());
```

现在也可以显式传递一个“顺序”执行策略：

```
sort(std::execution::seq,
     coll.begin(), coll.end());
```

传递顺序执行策略的好处是，如果要在运行时决定是顺序执行还是并行执行的话你不需要再修改调用方式。

想要并行排序也很简单：

```
sort(std::execution::par,
     coll.begin(), coll.end());
```

注意还有另一个并行执行策略：

```
sort(std::execution::par_unseq,
     coll.begin(), coll.end());
```

我会在后文解释其中的不同。

因此，问题又一次变成了（什么时候）使用并行排序会更好？在我的笔记本上10,000个字符串时并行排序只需要顺序排序一半的时间。即使只排序1000个字符串时并行排序也稍微更快一点。

结合其他的改进

注意还可以进一步修改来提升性能。例如，如果我们只根据数字进行排序，那么我们可以在排序准则中获取不包含前两个字母的子字符串来进行比较，这样可以再一次看到使用并行算法的双重改进：

```
sort(std::execution::par,
     coll.begin(), coll.end(),
     [] (const auto& a, const auto& b) {
         return a.substr(2) < b.substr(2);
     });
```

然而， `substr()` 是一个开销很大的成员函数，因为它会创建并返回一个新的临时字符串。通过使用字符串视图，即使顺序执行也能看到三重改进：

```
sort(coll.begin(), coll.end(),
     [] (const auto& a, const auto& b) {
         return std::string_view{a}.substr(2) < std::string_view{b}.substr(2);
     });
```

我们也可以轻松的把并行算法和字符串视图结合起来：

```
sort(std::execution::par,
     coll.begin(), coll.end(),
     [] (const auto& a, const auto& b) {
         return std::string_view{a}.substr(2) < std::string_view{b}.substr(2);
     });
```

结果显示这种方式可能比如下直接使用 `string` 的 `substr()` 成员然后顺序执行快10倍：

```
sort(coll.begin(), coll.end(),
     [] (const auto& a, const auto& b) {
         return a.substr(2) < b.substr(2);
     });
```

22.2 执行策略

你也可以向并行STL算法传递不同的 *执行策略(execution policies)* 作为第一个参数。它们定义在头文件 `<execution>` 中。表执行策略列出了所有标准的 *执行策略* 。

策略	含义
`std::execution::seq`	顺序执行
`std::execution::par`	并行化顺序执行
`std::execution::par_unseq`	并行化乱序（矢量化）执行

让我们仔细讨论这些执行策略：

- 使用 `seq` 指定 **顺序执行**

这意味着，像非并行化算法一样，当前线程会一个一个的对所有元素执行操作。使用这个选项和使用不接受执行策略参数的非并行化版本的效果类似，然而，这种形式将比非并行化版本多一些约束条件，例如 `for_each()` 不能返回值，所有的迭代器必须至少是前向迭代器。

提供这个策略的目的是让你可以只修改一个参数来要求顺序执行，而不是换用一个签名不同的函数来做到这一点。然而，注意使用这个策略的并行算法的行为和相应的非并行版本可能有些细微的不同。

- 使用 `par` 指定 **并行化顺序执行**

这意味着多个线程将会顺序地对元素执行操作。当某个线程对一个新的元素进行操作之前，它会先处理完它之前处理过的其他元素。

与 `par_unseq` 不同，这个策略可以保证在以下情况中不会出现问题或者死锁：执行了某个元素的第一步处理后必须在执行另一个元素的第一步处理之前执行这个元素接下来的处理步骤。

- 使用 `par_unseq` 指定 **并行化乱序执行**

这意味着多个线程执行时不需要保证某一个线程在执行完某一个元素的处理之前不会切换到其他的元素。特别地，这允许矢量化执行，一个线程可以先执行完多个元素的第一步处理，然后再执行下一步处理。

并行化乱序执行需要编译器/硬件的特殊支持来检测哪些操作如何矢量化。

所有的执行策略都是定义在命名空间 `std` 中的新类（ `sequenced_policy` 、 `parallel_policy` 、 `parallel_unsequenced_policy` ）的 `constexpr` 对象。还提供了新的类型特征 `std::is_execution_policy<>` 来在泛型编程中检查模板参数是否是执行策略。

22.3 异常处理

当处理元素的函数因为未捕获的异常而退出时所有的并行算法会调用 `std::terminate()` 。

注意即使选择了顺序执行策略也会这样。如果觉得这样不能接受，使用非并行化版本的算法将是更好的选择。

注意并行算法本身也可能抛出异常。如果它们申请并行执行所需的临时内存资源时失败了，可能会抛出 `std::bad_alloc` 异常。然而，不会有别的异常被抛出。

22.4 不使用并行算法的优势

既然已经有了并行算法并且还可以给并行算法指定顺序执行的策略，那么我们是否还需要原本的非并行算法呢?

事实上，除了向后的兼容性之外，使用非并行算法还可以提供以下好处:

- 可以使用输入和输出迭代器。
- 算法不会在遇到异常时 `terminate()` 。
- 算法可以避免因为意外使用元素导致的副作用。
- 算法可以提供额外的功能，例如 `for_each()` 会返回传入的可调用对象，我们可能会需要该对象最终的状态。

22.5 并行算法概述

表t22.2列出了标准中支持的所有不需要修改就可以并行运行的算法。

无限制的并行算法
`find_end()`, `adjacent_find()`
`search()`, `search_n()` （和"搜索器"一起使用时除外）
`swap_ranges()`
`replace()`, `replace_if()`

无限制的并行算法
fill()
generate()
remove(), remove_if(), unique()
reverse(), rotate()
partition(), stable_partition()
sort(), stable_sort(), partial_sort()
is_sorted(), is_sorted_until()
nth_element()
inplace_merge()
is_heap(), is_heap_until()
min_element(), max_element(), min_max_element()

表t22.3列出了还不支持并行运行的算法：

无并行版本的算法
accumulate()
partial_sum()
inner_product()
search() （和“搜索器”一起使用时）
copy_backward(), move_backward()
sample(), shuffle()
partition_point()
lower_bound(), upper_bound(), equal_range()
binary_search()
is_permutation()
next_permutation(), prev_permutation()

无并行版本的算法
push_heap(), pop_heap(), make_heap(), sort_heap()

注意对于 accumulate() 、 partial_sum() 、 inner_product() ，提供了新的要求更宽松的并行算法来代替（如下所示）：

- 为了并行地运行 accumulate() ，使用 reduce() 或者 transform_reduce() 。
- 为了并行地运行 partial_sum() ，使用 ...scan() 算法。
- 为了并行地运行 inner_product() ，使用 transform_reduce() 。

表t22.4列出了标准支持的所有只需要进行一些修改就可以并行运行的算法。

算法	限制
for_each()	返回类型 void 、前向迭代器
for_each_n()	前向迭代器（新）
all_of(), and_of(), none_of()	前向迭代器
find(), find_if(), find_if_not()	前向迭代器
find_first_of()	前向迭代器
count(), count_if()	前向迭代器
mismatch()	前向迭代器
equal()	前向迭代器
is_partitioned()	前向迭代器
partial_sort_copy()	前向迭代器
includes()	前向迭代器
lexicographical_compare()	前向迭代器
fill_n()	前向迭代器
generate_n()	前向迭代器
reverse_copy()	前向迭代器
rotate_copy()	前向迭代器

算法	限制
copy(), copy_n(), copy_if()	前向迭代器
move()	前向迭代器
transform()	前向迭代器
replace_copy(), replace_copy_if()	前向迭代器
remove_copy(), remove_copy_if()	前向迭代器
unique_copy()	前向迭代器
partition_copy()	前向迭代器
merge()	前向迭代器
set_union(), set_intersection()	前向迭代器
set_differrnce(), set_symmetric_difference()	前向迭代器
inclusive_scan(), exclusive_scan()	前向迭代器（新）
transform_inclusive_scan(), transform_exclusive_scan()	前向迭代器（新）

22.6 并行编程的新算法的动机

C++17还引入了一些补充的算法来实现那些从C++98就可用的标准算法的并行执行。

22.6.1 reduce()

例如， reduce() 作为 accumulate() 的并行版本引入，后者会"累积"所有的元素（你可以自定义"累积"的具体行为）。例如，考虑下面对 accumulate() 的使用：

```
#include <iostream>
#include <vector>
#include <numeric>  // for accumulate()

void printSum(long num)
{
    // 用数字序列1 2 3 4创建coll:
    std::vector<long> coll;
    coll.reserve(num * 4);
    for (long i = 0; i < num; ++i) {
```

```
        coll.insert(coll.end(), {1, 2, 3, 4});
    }

    auto sum = std::accumulate(coll.begin(), coll.end(), 0L);
    std::cout << "accumulate(): " << sum << '\n';
}

int main()
{
    printSum(1);
    printSum(1000);
    printSum(1000000);
    printSum(10000000);
}
```

我们计算出了所有元素的和，输出为：

```
accumulate(): 10
accumulate(): 10000
accumulate(): 10000000
accumulate(): 100000000
```

可结合可交换操作的并行化

上文的示例程序可以替换成 `reduce()` 来实现并行化：

```
#include <iostream>
#include <vector>
#include <numeric>  // for reduce()
#include <execution>

void printSum(long num)
{
    // 用数字序列1 2 3 4创建coll:
    std::vector<long> coll;
    coll.reserve(num * 4);
    for (long i = 0; i < num; ++i) {
        coll.insert(coll.end(), {1, 2, 3, 4});
    }

    auto sum = std::reduce(std::execution::par, coll.begin(), coll.end(), 0L);
    std::cout << "reduce():     " << sum << '\n';
}
```

```
int main()
{
    printSum(1);
    printSum(1000);
    printSum(1000000);
    printSum(10000000);
}
```

输出基本相同，程序可能会变得更快也可能会变得更慢（根据是否支持启动多个线程、启动线程的时间大于还是小于并行运行节省的时间）。

这里使用的操作是 + ，这个操作是可结合可交换的，因为整型元素相加的顺序并不会影响结果。

不可交换操作的并行化

然而，对于浮点数来说相加的顺序不同结果也可能不同，像下面的程序演示的一样：

```
#include <iostream>
#include <vector>
#include <numeric>
#include <execution>

void printSum(long num)
{
    // 用数字序列0.1 0.3 0.0001创建coll：
    std::vector<double> coll;
    coll.reserve(num * 4);
    for (long i = 0; i < num; ++i) {
        coll.insert(coll.end(), {0.1, 0.3, 0.00001});
    }

    auto sum1 = std::accumulate(coll.begin(), coll.end(), 0.0);
    std::cout << "accumulate(): " << sum1 << '\n';
    auto sum2 = std::reduce(std::execution::par, coll.begin(), coll.end(), 0.0);
    std::cout << "reduce():     " << sum2 << '\n';
    std::cout << (sum1==sum2 ? "equal\n" : "differ\n");
}

#include <iomanip>

int main()
{
    std::cout << std::setprecision(5);
    // 译者注：此处原文是
    // std::cout << std::setprecision(20);
```

```
    // 应是作者笔误

    printSum(1);
    printSum(1000);
    printSum(1000000);
    printSum(10000000);
}
```

这里我们同时使用了 `accumulate()` 和 `reduce()` 来计算结果。一个可能的输出是：

```
accumulate(): 0.40001
reduce():     0.40001
equal
accumulate(): 400.01
reduce():     400.01
differ
accumulate(): 400010
reduce():     400010
differ
accumulate(): 4.0001e+06
reduce():     4.0001e+06
differ
```

尽管结果看起来一样，但实际上可能是不同的。用不同顺序相加浮点数就可能会导致这样的结果。

如果我们改变输出的浮点数精度：

```
std::cout << std::setprecision(20);
```

我们可以看到下面的结果有一些不同：

```
accumulate(): 0.40001000000000003221
reduce():     0.40001000000000003221
equal
accumulate(): 400.01000000000533419
reduce():     400.01000000000010459
differ
accumulate(): 400009.99999085225863
reduce():     400009.9999999878346
differ
accumulate(): 4000100.0004483023658
reduce():     4000100.0000019222498
differ
```

因为没有是否使用、何时使用、怎么使用并行算法的保证，某些平台上的结果可能看起来是相同的（当元素数量不超过一定值时）。

要想了解更多关于 `reduce()` 的细节，见参考小节。

不可结合操作的并行化

让我们把累积操作改为加上每个值的平方：

```
#include <iostream>
#include <vector>
#include <numeric>  // for accumulate()

void printSum(long num)
{
    // 用数字序列1 2 3 4创建coll
    std::vector<long> coll;
    coll.reserve(num * 4);
    for (long i = 0; i < num; ++i) {
        coll.insert(coll.end(), {1, 2, 3, 4});
    }

    auto squaredSum = [] (auto sum, auto val) {
                          return sum + val * val;
                      };

    auto sum = std::accumulate(coll.begin(), coll.end(), 0L, squaredSum);
    std::cout << "accumulate(): " << sum << '\n';
}

int main()
{
```

```
    printSum(1);
    printSum(1000);
    printSum(1000000);
    printSum(10000000);
}
```

这里，我们传递了一个lambda，把没新的元素的值的平方加到当前的和上：

```
auto squaredSum = [] (auto sum, auto val) {
                      return sum + val * val;
                  };
```

使用 `accumulate()` 的输出将是：

```
accumulate(): 30
accumulate(): 30000
accumulate(): 30000000
accumulate(): 300000000
```

然而，让我们切换到 `reduce()` 并行执行：

```
#include <iostream>
#include <vector>
#include <numeric>  // for reduce()
#include <execution>

void printSum(long num)
{
    // 用数字序列1 2 3 4创建coll:
    std::vector<long> coll;
    coll.reserve(num * 4);
    for (long i = 0; i < num; ++i) {
        coll.insert(coll.end(), {1, 2, 3, 4});
    }

    auto squaredSum = [] (auto sum, auto val) {
                          return sum + val * val;
                      };

    auto sum = std::reduce(std::execution::par, coll.begin(), coll.end(), 0L, squ
    std::cout << "reduce():     " << sum << '\n';
}

int main()
```

```
{
    printSum(1);
    printSum(1000);
    printSum(1000000);
    printSum(10000000);
}
```

输出可能是这样的：

```
reduce():     30
reduce():     30000
reduce():     -425251612
reduce():     705991074
```

没错，结果 *有时* 有可能是错的。问题在于这个操作是不可结合的。如果我们并行的将这个操作应用于元素1、2、3，那么我们可能会首先计算 `0+1*1` 和 `2+3*3` ，但是当我们组合中间结果时，我们把后者作为了第二个参数，实质上是计算了：

```
(0+1*1) + (2+3*3) * (2+3*3)
```

但是为什么这里的结果有时候是正确的呢？好吧，看起来是在这个平台上，只有当元素达到一定数量时 `reduce()` 才会并行运行。而这个行为是完全符合标准的。因此，需要像这里一样使用足够多的元素作为测试用例。

解决这个问题的方法是使用另一个新算法 `transform_reduce()` 。它把我们对每一个元素的操作（这个过程可以并行化）和可交换的结果的累加（这个过程也可以并行化）分离开来。

```
#include <iostream>
#include <vector>
#include <numeric>  // for transform_reduce()
#include <execution>
#include <functional>

void printSum(long num)
{
   // 用数字序列1 2 3 4创建coll：
   std::vector<long> coll;
   coll.reserve(num * 4);
   for (long i = 0; i < num; ++i) {
       coll.insert(coll.end(), {1, 2, 3, 4});
   }
```

```
    auto sum = std::transform_reduce(std::execution::par, coll.begin(), coll.end()
                                     0L, std::plus{},
                                     [] (auto val) {
                                         return val * val;
                                     });
    std::cout << "transform_reduce(): " << sum << '\n';
}

int main()
{
    printSum(1);
    printSum(1000);
    printSum(1000000);
    printSum(10000000);
}
```

当调用 `transform_reduce()` 时，我们传递了

- 允许并行执行的执行策略
- 要处理的值范围
- 外层累积的初始值 `0L`
- 外层累积的操作 `+`
- 在累加之前处理每个值的lambda

`transform_reduce()` 可能是到目前为止最重要的并行算法，因为我们经常在组合值之前先修改它们（也被称为 *map reduce* 原理）。

更多关于 `transform_reduce()` 的细节，见参考小节。

使用 `transform_reduce()` 进行文件系统操作

这里有另一个并行运行 `transform_reduce()` 的例子：

```
#include <vector>
#include <iostream>
#include <numeric>      // for transform_reduce()
#include <execution>    // for 执行策略
#include <filesystem>   // 文件系统库
#include <cstdlib>      // for EXIT_FAILURE

int main(int argc, char *argv[]) {
    // 根目录作为命令行参数传递：
    if (argc < 2) {
```

```
        std::cout << "Usage: " << argv[0] << " <path> \n";
        return EXIT_FAILURE;
    }
    std::filesystem::path root{argv[1]};

    // 初始化文件树中所有文件路径的列表：
    std::vector<std::filesystem::path> paths;
    try {
        std::filesystem::recursive_directory_iterator dirpos{root};
        std::copy(begin(dirpos), end(dirpos), std::back_inserter(paths));
    }
    catch (const std::exception& e) {
        std::cerr << "EXCEPTION: " << e.what() << std::endl;
        return EXIT_FAILURE;
    }

    // 累积所有普通文件的大小：
    auto sz = std::transform_reduce(
                    std::execution::par,                    // 并行执行
                    paths.cbegin(), paths.cend(),           // 范围
                    std::uintmax_t{0},                      // 初始值
                    std::plus<>(),                          // 累加...
                    [] (const std::filesystem::path& p) {    // 如果是普通文件返回文
                        return is_regular_file(p) ? file_size(p) : std::uintmax_t
                    });
    std::cout << "size of all " << paths.size()
              << " regular files: " << sz << '\n';
}
```

首先，我们递归收集了命令行参数给出的目录中所有的文件系统路径：

```
std::filesystem::path root{argv[1]};

std::vector<std::filesystem::path> paths;
std::filesystem::recursive_directory_iterator dirpos{root};
std::copy(begin(dirpos), end(dirpos), std::back_inserter(paths));
```

注意因为我们可能会传递无效的路径，所以可能会抛出（文件系统）异常。

之后，我们遍历文件系统路径的集合来累加普通文件的大小：

```
auto sz = std::transform_reduce(
                std::execution::par,                    // 并行执行
                paths.cbegin(), paths.cend(),           // 范围
                std::uintmax_t{0},                      // 初始值
```

```
                std::plus<>(),                          // 累加...
                [] (const std::filesystem::path& p) {    // 如果是普通文件返回文件大
                    return is_regular_file(p) ? file_size(p) : std::uintmax_t{0};
                });
```

新的标准算法 `transform_reduce()` 以如下方式执行：

- 最后一个参数会作用于每一个元素。这里，传入的lambda会对每个元素调用，如果是常规文件的话会返回它的大小。
- 倒数第二个参数用来把所有大小组合起来。因为我们想要累加大小，所以使用了标准函数对象 `std::plus<>` 。
- 倒数第三个参数是组合操作的初始值。这个值的类型也是整个算法的返回值类型。我们把 `0` 转换为 `file_size()` 的返回值类型 `std::uintmax_t` 。否则，如果我们简单的传递一个 `0` ，我们会得到窄化为 `int` 类型的结果。因此，如果路径集合是空的，那么整个调用会返回 `0` 。

注意查询一个文件的大小是一个开销很大的操作，因为它需要操作系统调用。因此，使用并行算法来通过多个线程并行调用这个转换函数（从路径到大小的转换）并计算大小的总和会快很多。之前第一个测试结果就已经显示出并行算法的明显优势了（最多能快两倍）。

注意你不能直接向并行算法传递指定目录的迭代器，因为目录迭代器是输入迭代器，而并行算法要求至少是前向迭代器。

最后，注意 `transform_reduce()` 算法在头文件 `<numeric>` 中而不是 `<algorithm>` 中定义（类似于 `accumulate()` ，它被认为是数值算法）。