Contents

Part I

基本语言特性

这一部分介绍了C++17中新的核心语言特性，但不包括那些专为泛型编程（即template）设计的特性。这些新增的特性对于应用程序员的日常编程非常有用，因此每一个使用C++17的C++程序员都应该了解它们。

专为模板编程设计的新的核心语言特性在Part II中介绍。

Part II

模板特性

这一部分介绍了 C++17 为泛型编程（即 template）提供的新的语言特性。

我们首先从类模板参数推导开始，这一特性只影响模板的使用。之后的章节会介绍为编写泛型代码（函数模板，类模板，泛型库等）的程序员提供的新特性。

Part III

新的标准库组件

这一部分介绍 C++17 中新的标准库组件。

Chapter 20

文件系统库

直到C++17，Boost.Filesystem库终于被C++标准采纳。在这个过程中，这个库用新的语言特性进行了很多调整、改进了和其他库的一致性、进行了精简、还扩展了很多确实的功能（例如计算两个文件系统路径之间的相对路径）。

20.1 基本的例子

让我们以一些基本的例子开始。

20.1.1 打印文件系统路径类的属性

下面的程序允许我们传递一个字符串作为文件系统路径，然后根据给定路径的文件类型打印出一些信息：

filesystem/checkpath1.cpp

```
#include <iostream>
#include <filesystem>
#include <cstdlib> // for EXIT_FAILURE

int main(int argc, char* argv[])
{
    if (argc < 2) {
        std::cout << "Usage: " << argv[0] << " <path> \n";
        return EXIT_FAILURE;
    }

    std::filesystem::path p{argv[1]}; // p代表一个文件系统路径（可能不存在）
    if (is_regular_file(p)) {   // 路径p是普通文件吗？
        std::cout << p << " exists with " << file_size(p) << " bytes\n";
    }
    else if (is_directory(p)) { // 路径p是目录吗？
        std::cout << p << " is a directory containing:\n";
        for (const auto& e : std::filesystem::directory_iterator{p}) {
```

```
            std::cout << "   " << e.path() << '\n';
        }
    }
    else if (exists(p)) {       // 路径p存在吗？
        std::cout << p << " is a special file\n";
    }
    else {
        std::cout << "path " << p << " does not exist\n";
    }
}
```

我们首先把传入的命令行参数转换为了一个文件系统路径：

```
std::filesystem::path p{argv[1]};   // p代表一个文件系统路径（有可能不存在）
```

然后，我们进行了下列检查：

- 如果改路径代表一个普通文件，我们打印出它的大小：

  ```
  if (is_regular_file(p)) {   // 路径p是普通文件？
      std::cout << p << " exists with " << file_size(p) << " bytes\n";
  }
  ```

  像下面这样调用程序：

  ```
  checkpath checkpath.cpp
  ```

  将会有如下输出：

  ```
  "checkpath.cpp" exists with 907 bytes
  ```

  注意输出路径时会自动把路径名用双引号括起来输出（把路径用双引号括起来、反斜杠用另一个反斜杠转义，对 Windows 路径来说是一个问题）。

- 如果路径是一个目录，我们迭代这个目录中的所有文件并打印出这些文件的路径：

  ```
  if (is_directory(p)) {      // 路径p是目录？
      std::cout << p << " is a directory containing:\n";
      for (auto& e : std::filesystem::directory_iterator{p}) {
          std::cout << "   " << e.path() << '\n';
      }
  }
  ```

  这里，我们使用了 `directory_iterator`，它提供了 `begin()` 和 `end()`，所以我们可以使用范围 for 循环来迭代 `directory_entry` 元素。在这里，我们使用了 `directory_entry` 的成员函数 `path()`，返回该目录项的文件系统路径。像下面这样调用程序：

  ```
  checkpath .
  ```

  输出将是：

  ```
  "." is a directory containing:
    "./checkpath.cpp"
    "./checkpath.exe"
    ...
  ```

- 最后，我们检查传入文件系统路径是否不存在：

```
if (exists(p)) {        // 路径p存在吗？
    ...
}
```

注意根据参数依赖查找 *(argument dependent lookup)(ADL)*，你不需要使用完全限定的名称来调用 `is_regular_file()`、`file_size()`、`is_directory()`、`exists()` 等函数。它们都属于命名空间 `std::filesystem`，但是因为它们的参数也属于这个命名空间，所以调用它们时会自动在这个命名空间中进行查找。

在 Windows 下处理路径

（译者注：可能是水平有限，完全看不懂作者在这一小节的逻辑，所以只能按照自己的理解胡乱翻译，如有错误请见谅。）

默认情况下，输出路径时用双引号括起来并用反斜杠转义反斜杠在 Windows 下会导致一个问题。在 Windows 下以如下方式调用程序：

```
checkpath C:\
```

将会有如下输出：

```
"C:" is a directory containing:
  ...
  "C:Users"
  "C:Windows"
```

用双引号括起来输出路径可以确保输出的文件名可以被直接复制粘贴到其他程序里，并且经过转义之后还会恢复为原本的文件名。然而，终端通常不接受这样的路径。

因此，一个在 Windows 下的可移植版本应该使用成员函数 `string()`，这样可以在向标准输出写入路径时避免输出双引号：

filesystem/checkpath2.cpp

```
#include <iostream>
#include <filesystem>
#include <cstdlib>  // for EXIT_FAILURE

int main(int argc, char* argv[])
{
    if (argc < 2) {
        std::cout << "Usage: " << argv[0] << " <path> \n";
        return EXIT_FAILURE;
    }
    std::filesystem::path p{argv[1]};   // p代表一个文件系统路径（可能不存在）
    if (is_regular_file(p)) {   // 路径p是普通文件吗？
        std::cout << '"' << p.string() << "\" existes with " << file_size(p) << " bytes\n";
    }
    else if (is_directory(p)) { // 路径p是目录吗？
        std::cout << '"' << p.string() << "\" is a directory containing:\n";
```

```
        for (const auto& e : std::filesystem::directory_iterator{p}) {
            std::cout << "  \"" << e.path().string() << "\"\n";
        }
    }
    else if (exists(p)) {       // 路径p存在吗？
        std::cout << '"' << p.string() << "\" is a special file\n";
    }
    else {
        std::cout << "path \"" << p.string() << "\" does not exist\n";
    }
}
```

现在，在 Windows 上以下面方式调用程序：

```
checkpath C:\
```

将会有如下输出：

```
"C:\" is a directory containing:
  ...
  "C:\Users"
  "C:\Windows"
```

通用字符串格式还提供了其他转换来把 string 转换为本地编码。

20.1.2 用 `switch` 语句处理不同的文件系统类型

我们可以像下面这样修改并改进上面的例子：

filesystem/checkpath3.cpp

```
#include <iostream>
#include <filesystem>
#include <cstdlib>  // for EXIT_FAILURE

int main(int argc, char* argv[])
{
    if (argc < 2) {
        std::cout << "Usage: " << argv[0] << " <path> \n";
        return EXIT_FAILURE;
    }
    namespace fs = std::filesystem;

    switch (fs::path p{argv[1]}; status(p).type()) {
        case fs::file_type::not_found:
            std::cout << "path \"" << p.string() << "\" does not exist\n";
            break;
        case fs::file_type::regular:
            std::cout << '"' << p.string() << "\" exists with " << file_size(p) << " bytes\n";
            break;
```

```
        case fs::file_type::directory:
            std::cout << '"' << p.string() << "\" is a directory containing:\n";
            for (const auto& e : std::filesystem::directory_iterator{p}) {
                std::cout << "  " << e.path().string() << '\n';
            }
            break;
        default:
            std::cout << '"' << p.string() << "\" is a special file\n";
            break;
    }
}
```

命名空间 fs

首先，我们做了一个非常普遍的操作：我们定义了 fs 作为命名空间 std::filesystem 的缩写：

```
namespace fs = std::filesystem;
```

使用新初始化的命名空间的一个例子是下面 switch 语句中的路径 p：

```
fs::path p{argv[1]};
```

这里的 switch 语句使用了新的带初始化的 switch 语句特性，初始化路径的同时把路径的类型作为分支条件：

```
switch (fs::path p{argv[1]}; status(p).type()) {
    ...
}
```

表达式 status(p).type() 首先创建了一个 file_status 对象，然后该对象的 type() 方法返回了一个 file_type 类型的值。通过这种方式我们可以直接处理不同的类型，而不需要使用 is_regular_file()、is_directory() 等函数构成的 if-else 链。我们通过多个步骤（先调用 status() 再调用 type() 才得到了最后的类型，因此我们不需要为不感兴趣的其他信息付出多余的系统调用开销。

注意可能已经有特定实现的 file_type 存在。例如，Windows 就提供了特殊的文件类型 junction。然而，使用了它的代码是不可移植的。

20.1.3 创建不同类型的文件

在介绍了文件系统的只读操作之后，让我们给出首个进行修改的例子。下面的程序在一个子目录 tmp 中创建了不同类型的文件：

filesystem/createfiles.cpp

```
#include <iostream>
#include <fstream>
#include <filesystem>
#include <cstdlib> // for std::exit()和EXIT_FAILURE

int main()
{
```

```
    namespace fs = std::filesystem;
    try {
        // 创建目录tmp/test/（如果不存在的话）
        fs::path testDir{"tmp/test"};
        create_directories(testDir);

        // 创建数据文件tmp/test/data.txt：
        auto testFile = testDir / "data.txt";
        std::ofstream  dataFile{testFile};
        if (!dataFile) {
            std::cerr << "OOPS, can't open\"" << testFile.string() << "\"\n";
            std::exit(EXIT_FAILURE);    // 以失败方式结束程序
        }
        dataFile << "The answer is 42\n";

        // 创建符号链接tmp/slink/，指向tmp/test/：
        create_directory_symlink("test", testDir.parent_path() / "slink");
    }
    catch (const fs::filesystem_error& e) {
        std::cerr << "EXCEPTION: " << e.what() << '\n';
        std::cerr << "    path1: \"" << e.path1().string() << "\"\n";
    }

    // 递归列出所有文件（同时遍历符号链接）
    std::cout << fs::current_path().string() << ":\n";
    auto iterOpts{fs::directory_options::follow_directory_symlink};
    for (const auto& e : fs::recursive_directory_iterator(".", iterOpts)) {
        std::cout << "  " << e.path().lexically_normal().string() << '\n';
    }
}
```

让我们一步步来分析这段程序。

命名空间 `fs`

首先，我们又一次定义了 `fs` 作为命名空间 `std::filesystem` 的缩写：

```
namespace fs = std::filesystem;
```

之后我们使用这个命名空间为临时文件创建了一个基本的子目录：

```
fs::path testDir{"tmp/test"};
```

创建目录

当我们尝试创建子目录时：

```
create_directories(testDir);
```

通过使用 create_directories() 我们可以递归创建整个路径中所有缺少的目录（还有一个 create_directory() 只在已存在的目录中创建目录）。

当目标目录已经存在时这个调用并不会返回错误。然而，其他的问题会导致错误病抛出一个相应的异常。

如果 testDis 已经存在，create_directories() 会返回 false。因此，你可以这么写：

```
if (!create_directories(testDir)) {
    std::cout << "\"" << testDir.string() << "\" already exists\n";
}
```

创建普通文件

之后我们用一些内容创建了一个新文件 tmp/test/data.txt：

```
auto testFile = testDir / "data.txt";
std::ofstream dataFile{testFile};
if (!dataFile) {
    std::cerr << "OOPS, can't open \"" << testFile.string() << "\"\n";
    std::exit(EXIT_FAILURE);  // 失败退出程序
}
dataFile << "The answer is 42\n";
```

这里，我们使用了运算符/来扩展路径，然后传递给文件流的构造函数。如你所见，普通文件的创建可以使用现有的I/O流库来实现。然而，I/O流的构造函数多了一个以文件系统路径为参数的版本（一些函数例如open()也添加了这种重载版本）。

注意你仍然应该总是检查创建/打开文件的操作是否成功了。这里有很多种可能发生的错误（见下文）。

创建符号链接

接下来的语句尝试尝试创建符号链接 tmp/slink 指向目录 tmp/test：

```
create_directory_symlink("test", testDir.parent_path() / "slink");
```

注意第一个参数的路径是以即将创建的符号链接所在的目录为起点的相对路径。因此，你必须传递"test"而不是"tmp/test"来高效的创建链接 tmp/slink 指向 tmp/test。如果你调用：

```
std::filesystem::create_directory_symlink("tmp/test", "tmp/slink");
```

你将会高效的创建符号链接 tmp/slink，然而它会指向 tmp/tmp/test。

注意通常情况下，也可以调用 create_symlink() 代替 create_ directory_symlink() 来创建目录的符号链接。然而，一些操作系统可能对目录的符号链接有特殊处理或者当知道要创建的符号链接指向目录时会有优化，因此，当你想创建指向目录的符号链接时你应该使用 create_directory_ symlink()。

最后，注意这个调用在Windows上可能会失败并导致错误处理，因为创建符号链接可能需要管理员权限。

递归遍历目录

最后，我们递归的遍历了当前目录：

```
auto iterOpts = fs::directory_options::follow_directory_symlink;
for (auto& e : fs::recursive_directory_iterator(".", iterOpts)) {
    std::cout << "  " << e.path().lexically_normal().string() << '\n';
}
```

注意我们使用了一个递归目录迭代器并传递了选项`follow_directory_ symlink`来遍历符号链接。因此，我们在POSIX兼容系统上可能会得到类似于如下输出：

```
/home/nico:
  ...
  tmp
  tmp/slink
  tmp/slink/data.txt
  tmp/test
  tmp/test/data.txt
  ...
```

在Windows系统上有类似如下输出：

```
C:\Users\nico:
  ...
  tmp
  tmp\slink
  tmp\slink\data.txt
  tmp\test
  tmp\test\data.txt
  ...
```

注意在我们打印目录项之前调用了`lexically_normal()`。如果略过这一步，目录项的路径可能会包含一个前缀，这个前缀是创建目录迭代器时传递的实参。因此，在循环内直接打印路径：

```
auto iterOpts = fs::directory_options::follow_directory_symlink;
for (auto& e : fs::recursive_directory_iterator(".", iterOpts)) {
    std::cout << "  " << e.path() << '\n';
}
```

将会在POSIX兼容系统上有如下输出：

```
all files:
  ...
  "./testdir"
  "./testdir/data.txt"
  "./tmp"
  "./tmp/test"
  "./tmp/test/data.txt"
```

在Windows上，输出将是：

```
all files:
  ...
  ".testdir"
  ".testdirdata.txt"
```

```
".tmp"
".tmptest"
".tmptestdata.txt"
```

通过调用 `lexically_normal()` 我们可以得到正规化的路径，它移除了前导的代表当前路径的点。还有，如上文所述，通过调用 `string()` 我们避免了输出路径时用双引号括起来。这里没有调用 `string()` 的输出结果在 POSIX 兼容的系统上看起来 OK（只是路径两端有双引号），但在 Windows 上的结果看起来就很奇怪（因为每一个反斜杠都需要反斜杠转义）。

错误处理

文件系统往往是麻烦的根源。你可能因为在文件名中使用了无效的字符而导致操作失败，或者当你正在访问文件系统时它已经被其他程序修改了。因此，根据平台和权限的不同，这个程序中可能会有很多问题。

对于那些没有被返回值覆盖的情况（例如当目录已经存在时），我们捕获了相应的异常并打印了一般的信息和第一个路径：

```
try {
    ...
}
catch (const fs::filesystem_error& e) {
    std::cerr << "EXCEPTION: " << e.what() << '\n';
    std::cerr << "    path1: \"" << e.path1().string() << "\"\n";
}
```

例如，如果我们不能创建目录，将会打印出类似于如下消息：

```
EXCEPTION: filesystem error: cannot create directory: [tmp/test]
    path1: "tmp/test"
```

如果我们不能创建符号链接可能是因为它已经存在了，或者我们可能需要特殊权限，这些情况下你可能会得到如下消息：

```
EXCEPTION: create_directory_symlink: Can't create a file when it
                                     already exists: "tmp\test\
                                     data.txt", "testdir"
    path1: "tmp\test\data.txt"
```

或者：

```
EXCEPTION: create_directory_symlink: A requied privilege is not
                                     held by the client.: "test
                                     ", "tmp\slink"
    path1: "test"
```

在每一种情况下，都要注意在多用户/多进程操作系统中情况可能会在任何时候改变，这意味着你刚刚创建的目录甚至可能已经被删除、重命名、或已经被同名的文件覆盖。因此，很显然不能只根据当前的情况就保证一个预期操作一定是有效的。最好的方式就是尝试做想做的操作（例如，创建目录、打开文件）并处理抛出的异常和错误，或者验证预期的行为。

然而，有些时候文件系统操作能正常执行但不是按你预想的结果。例如，如果你想在指定目录中创建一个文件并且已经有了一个和目录同名的指向另一个目录的符号链接，那么这个文件可能在一个预料之外的地方创建或者覆写。（译者注：比如你想在当前目录下递归创建”a/b”，但已经有了一个”a” 是一个指向”c” 目录的符号链接，这时你实际会在目录”c” 下创建”b”）这种情况是有可能的（用户完全有可能会创建目录的符号链接），但是如果你想检测这种情况，在创建文件之前你需要检查文件是否存在（这可能比你一开始想的要复杂很多）。

再强调一次：文件系统并不保证进行处理之前的检查的结果直到你进行处理时仍然有效。

20.1.4 使用并行算法处理文件系统

参见dirsize.cpp 查看另一个使用并行算法计算目录树中所有文件大小之和的例子。

20.2 原则和术语

在讨论文件系统库的细节之前，我们不得不继续介绍一些设计原则和术语。这是必须的，因为标准库要覆盖不同的操作系统并把系统提供的接口映射为公共的 API。

20.2.1 通用的可移植的分隔符

C++ 标准库不仅标准化了所有操作系统的文件系统中公共的部分，在很多情况下，C++ 标准还尽可能的遵循 POSIX 标准的要求来实现。对于一些操作，只要是合理的就应该能正确执行，如果操作是不合理的，实现应该报错。这些错误可能是：

- 特殊的字符不能被用作文件名
- 文件系统的某些元素不支持创建（例如，符号链接）

不同文件系统的差异也应该纳入考虑：

- 大小写敏感：`"hello.txt"` 和`"Hello.txt"` 和`"hello.TXT"` 可能指向同一个文件（Windows 上）也可能指向三个不同的文件（POSIX 兼容系统）。
- 绝对路径和相对路径：
- 在某些系统上，”/bin” 是一个绝对路径（POSIX 兼容系统），然而在某些系统上不是（Windows）。

20.2.2 命名空间

文件系统库在 `std` 里有自己的子命名空间 `filesystem`。一个很常见的操作是定义缩写 `fs`：

```
namespace fs = std::filesystem;
```

这允许我们使用 `fs::current_path()` 代替 `std::filesystem:: current_path()`。

这一章的示例代码中将经常使用 `fs` 作为缩写。

注意你应该总是使用完全限定的函数调用，尽管不指明命名空间时通过 *参数依赖查找 (argument dependent lookup)(ADL)* 也能够工作。但如果不用命名空间限定有时可能导致意外的行为。

20.2.3 文件系统路径

文件系统库的一个关键元素是 path。它代表文件系统中某一个文件的名字。它由可选的根名称、可选的根目录、和一些以目录分隔符分隔的文件名组成。路径可以是相对的（此时文件的位置依赖于当前的工作目录）或者是绝对的。

路径可能有不同的格式：

- 通过格式，这是可移植的
- 本地格式，这是底层文件系统特定的

在 POSIX 兼容系统上通用格式和本地格式没有什么区别。在 Windows 上，通用格式 /tmp/test.txt 也是有效的本地格式，另外 \tmp\test.txt 也是有效的（/tmp/test.txt 和 \tmp\test.txt 是同一个路径的两种本地版本）。在 OPenVMS 上，相应的本地格式将是 [tmp]test.txt。

也有一些特殊的文件名：

- ”.” 代表当前 mulu
- ”..” 代表父目录

通用的路径格式如下：

```
[rootname] [rootdir] [relativepath]
```

这里：

- 可选的根名称是实现特定的（例如，在 POSIX 系统上可以是 //host，而在 Windows 上可以是 C:）
- 可选的根目录是一个目录分隔符
- 相对路径是若干目录分隔符分隔的文件名

目录分隔符由一个或多个 '/' 组成或者是实现特定的。

可移植的通用路径的例子有：

```
//host1/bin/hello.txt
.
tmp/
/a/b//.../c
```

注意在 POSIX 系统上最后一个路径和 /a/c 指向同一个位置，并且都是绝对路径。而在 Windows 上则是相对路径（因为没有指定驱动器/分区（盘））。

另一方面，C:/bin 在 Windows 上是绝对路径（在”C” 盘上的根目录 "bin"），但在 POSIX 系统上是一个相对路径（目录”C:” 下的子目录”bin”）。

在 Windows 系统上，反斜杠是实现特定的目录分隔符，因此上面的路径也可以使用反斜杠作为目录分隔符：

```
host1\bin\hello.txt
.
tmp\
\a\b\..\c
```

文件系统库提供了在本地格式和通用格式之间转换的函数。

一个 path 可能为空，这意味着没有定义路径。这种状态的含义不需要和 "." 一样。它的含义依赖于上下文。

20.2.4 正规化

路径可以进行正规化，在正规化的路径中：

- 文件名由单个推荐的目录分隔符分隔。
- 除非整个路径就是"."（代表当前目录），否则路径中不会使用"."。
- 路径中除了开头以外的地方不会包含".."（不能在路径中上下徘徊）。
- 除非整个路径就是"."或者".."，否则当路径结尾的文件名是目录时要在最后加上目录分隔符。

注意正规化之后以目录分隔符结尾的路径和不以目录分隔符结尾的路径是不同的。这是因为在某些操作系统中，当它们知道目标路径是一个目录时行为可能会发生改变（例如，有尾部的分隔符时符号链接将被解析）。

表路径正规化的效果列举了一些在POSIX系统和Windows系统上对路径进行正规化的例子。注意再重复一次，在POSIX系统上，`C:bar`和`C:`只是把冒号作为文件名的一部分的单个文件名，并没有特殊的含义，而在Windows上，它们指定了一个分区。

路径	POSIX正规化	Windows正规化
`foo/.///bar/../`	`foo/`	`foo\`
`//host/../foo.txt`	`//host/foo.txt`	`\\host\foo.txt`
`./f/../.f/`	`.f/`	`.f\`
`C:bar/../`	`.`	`C:`
`C:/bar/..`	`C:/`	`C:\`
`C:\bar\..`	`C:\bar\..`	`C:\`
`/./../data.txt`	`/data.txt`	`\data.txt`
`././`	`.`	`.`

Table 20.1: 路径正规化的效果

文件系统库同时提供了词法正规化（不访问文件系统）和依赖文件系统的正规化两种方式的相关函数。

20.2.5 成员函数VS独立函数

文件系统库提供了一些函数，有些是成员函数有些是独立函数。这么做的目的是：

- **成员函数开销较小**。这是因为它们是纯词法的操作，并不会访问实际的文件系统，这意味着它们不需要进行操作系统调用。例如：

```
mypath.is_absolute()        // 检查路径是否是绝对的
```

- **独立函数开销较大**。因为它们通常会访问实际的文件系统，这意味着需要进行操作系统调用。例如：

```
equivalent(path1, path2);   // 如果两个路径指向同一个文件则返回true
```

有时，文件系统库甚至为同一个功能既提供根据词法的版本又提供访问实际文件系统的版本：

```
std::filesystem::path fromP, toP;
...
toP.lexically_relative(fromP);  // 返回从fromP到toP的词法路径
relative(toP, fromP);           // 返回从fromP到toP的实际路径
```

得益于参数依赖查找*(ADL)*，很多情况下当调用独立函数时你不需要指明完整命名空间 `std::filesystem`，只要参数是文件系统库里定义的类型。只有当用其它类型隐式转换为参数时时你才需要给出完全限定的函数名。例如：

```
create_directory(std::filesystem::path{"tmpdir"});  // OK
remove(std::filesystem::path{"tmpdir"});            // OK
std::filesystem::create_directory("tmpdir");        // OK
std::filesystem::remove("tmpdir");                  // OK
create_directory("tmpdir");                         // ERROR
```

最后一个调用将会编译失败，因为我们并没有传递文件系统命名空间里的类型作为参数，因此也不会在该命名空间里查找符号 `create_directory`。

然而，这里有一个著名的陷阱：

```
remove("tmpdir");   // OOPS：调用C函数remove()
```

根据你包含的头文件，这个调用可能会找到C函数 `remove()`，它的行为有一些不同：它也会删除指定的文件但不会删除空目录。

因此，强烈推荐使用完全限定的文件系统库里的函数名。例如：

```
namespace fs = std::filesystem;
...
fs::remove("tmpdir");   // OK：调用C++文件系统库函数remove()
```

20.2.6 错误处理

如上文所述，文件系统是错误的根据。你必须考虑相应的文件是否存在、文件的操作是否被允许、该操作是否会违背资源限制。另外，当程序运行时其它进程可能创建、修改、或者移除了某些文件，这意味着事先检查并不能保证没有错误。

问题在于从理论上讲，你不能提前保证下一次文件系统操作能够成功。任何事先检查的结果都可能在你实际进行处理时失效。因此，最好的方法是在进行一个或多个文件系统操作时处理好相应的异常或者错误。

注意，当读写普通文件时，默认情况下I/O流并不会抛出异常或错误。当操作遇到错误时它只会什么也不做。因此，建议至少检查一下文件是否被成功打开。

因为并不是所有情况下都适合抛出异常（例如当一个文件系统调用失败时你想直接处理），所以文件系统库使用了混合的异常处理方式：

- 默认情况下，文件系统错误会作为异常处理。
- 然而，如果你想的话可以在本地处理具体的某一个错误。

因此，文件系统库通常为每个操作提供两个重载版本：

1. 默认情况下（没有额外的错误处理参数），出现错误时抛出 `filesystem_error` 异常。
2. 传递额外的输出参数时，可以得到一个错误码或错误信息，而不是异常。

注意在第二种情况下，你可能会得到一个特殊的返回值来表示特定的错误。

使用 `filesystem_error` 异常

例如，你可以尝试像下面这样创建一个目录：

```
if (!create_directory(p)) { // 发生错误时抛出异常（除非错误是该路径已经存在）
    std::cout << p << " already exists\n";  // 该路径已经存在
}
```

这里没有传递错误码参数，因此错误时通常会抛出异常。然而，注意当目录已存在时这种特殊情况是直接返回 `false`。因此，只有当其他错误例如没有权限创建目录、路径 `p` 无效、违反了文件系统限制（例如路径长度超过上限）时才会抛出异常。

可以直接或间接的用 `try-catch` 包含这段代码，然后处理 `std::filesystem::filesystem_error` 异常：

```
try {
    ...
    if (!create_directory(p)) { // 错误时抛出异常（除非错误是该路径已经存在）
        std::cout << p << " already exists\n"; // 该路径已经存在
    }
    ...
}
catch (const std::filesystem::filesystem_error& e) { // 派生自std::exception
    std::cout << "EXCEPTION: " << e.what() << '\n';
    std::cout << "     path: " << e.path1() << '\n';
}
```

如你所见，文件系统异常提供了标准异常的 `what()` 函数 API 来返回一个实现特定的错误信息。然而，API 还提供了 `path1()` 来获取错误相关的第一个路径，和 `path2()` 来获取相关的第二个路径。

使用 `error_code` 参数

另一种创建目录的方式如下所示：

```
std::error_code ec;
create_directory(p, ec);    // 发生错误时设置错误码
if (ec) {                   // 如果设置了错误码（因为发生了错误）
    std::cout << "ERROR: " << ec.message() << "\n";
}
```

之后，我们还可以检查特定的错误码：

```
if (ec == std::errc::read_only_file_system) {   // 如果设置了特定的错误码
    std::cout << "ERROR: " << p << " is read-only\n";
}
```

注意这种情况下，我们仍然必须检查 `create_directory()` 的返回值：

```
std::error_code ec;
if (!create_directory(p, ec)) { // 发生错误时设置错误码
    // 发生任何错误时
    std::cout << "can't create directory " << p << "\n";
    std::cout << "error: " << ec.message() << "\n";
}
```

然而，并不是所有的文件系统操作都提供这种能力（因为它们在正常情况下会返回一些值）。

类型 `error_code` 由 C++11 引入，它包含了一系列可移植的错误条件例如 `std::errc::read_only_filesystem`。在 POSIX 兼容的系统上这些被映射为 `errno` 的值。

20.2.7 文件类型

不同的操作系统支持不同的文件类型。标准文件系统库中也考虑到了这一点，它定义了一个枚举类型 `file_type`，标准中定义了如下的值：

```
namespace std::filesystem {
    enum class file_type {
        regular, directory, symlink,
        block, character, fifo, socket,
        ...
        none, not_found, unknown,
    };
}
```

表`file_type` 的值列出了这些值的含义。

值	含义
`regular`	普通文件
`directory`	目录文件
`symlink`	符号链接文件
`character`	字符特殊文件
`block`	块特殊文件
`fifo`	FIFO 或者管道文件
`socket`	套接字文件
…	附加的实现定义的文件类型
`none`	文件的类型未知
`unknown`	文件存在但推断不出类型
`not_found`	虚拟的表示文件不存在的类型

Table 20.2: 文件系统类型的值

操作系统平台可能会提供附加的文件类型值。然而，使用它们是不可移植的。例如，Windows 就提供了文件类型值 `junction`，它被用于 NTFS 文件系统中的 *NTFS junctions*（也被称为软链接）。它们被用作链接来访问同一台电脑上不同的子卷 (盘)。

除了普通文件和目录之外，最常见的类型是符号链接，它是一种指向另一个位置的文件。指向的为之可能有一个文件也可能没有。注意有些操作系统和/或文件系统（例如 FAT 文件系统）完全不支持符号链接。有些操作系统只支持普通文件的符号链接。注意在 Windows 上需要特殊的权限才能创建符号链接，可以用 `mklink` 命令创建。

字符特殊文件、块特殊文件、FIFO、套接字都来自于 UNIX 文件系统。目前，Visual C++ 并没有使用这四种类型中的任何一个。[1]

如你所见，有一些特殊的值来表示文件不存在或者类型未知或者无法探测出类型。

在这一章的剩余部分我将使用两种广义的类型来代表相应的若干文件类型：

- *其他文件*：除了普通文件、目录、符号链接之外的所有类型的文件。库函数 `is_other()` 和这个术语相匹配。
- *特殊文件*：下列类型的文件：字符特殊文件、块特殊文件、FIFO、套接字。

另外，*特殊*文件类型加上实现定义的文件类型就构成了*其他*文件类型。

20.3 路径操作

有很多处理文件系统的操作。这些操作的关键是一个类型 `std::filesystem::path`，它表示一个可能存在也可能不存在的文件的绝对或相对的路径。

你可以创建路径、检查路径、修改路径、比较路径。因为这些操作一般都不会访问实际的文件系统（例如不会检查文件是否存在，也不会解析符号链接），所以它们的开销很小。因此，它们通常被定义为成员函数（如果这些操作既不是构造函数也不是运算符的话）。

20.3.1 创建路径

表创建路径列出了创建新的路径对象的方法。

调用	效果
`path{charseq}`	用一个字符序列初始化路径
`path{beg, end}`	用一个范围初始化路径
`u8path{u8string}`	用一个 UTF-8 字符串初始化路径
`current_path()`	返回当前工作目录的路径
`temp_directory_path()`	返回临时文件的路径

Table 20.3: 创建路径

第一个构造函数以字符序列为参数，这里的字符序列代表一系列有效的方式：

- 一个 `string`
- 一个 `string_view`
- 一个以空字符结尾的字符数组
- 一个以空字符结尾的字符输入迭代器（指针）

注意 `current_path()` 和 `temp_directory_path()` 都是开销较大的操作，因为它们依赖于系统调用。如果给 `current_path()` 传递一个参数，它也可以用来 修改当前工作目录。

通过 `u8path()` 你可以使用 UTF-8 字符串创建可移植的路径。例如：

```
// 将路径p初始化为"Köln"（Cologne的德语名）
std::filesystem::path p{std::filesystem::u8path(u8"K\u00F6ln")};
```

[1] Windows 管道的行为有些不同，也不被识别为 `fifo`。

```
...

// 用UTF-8字符串创建目录
std::string utf8String = readUTF8String(...);
create_directory(std::filesystem::u8path(utf8String));
```

20.3.2 检查路径

表检查路径列出了检查路径p时可以调用的函数。注意这些操作都不会访问底层的操作系统，因此都是path类的成员函数。

调用	效果
p.empty()	返回路径是否为空
p.is_absolute()	返回路径是否是绝对的
p.is_relative()	返回路径是否是相对的
p.has_filename()	返回路径是否既不是目录也不是根名称
p.has_stem()	和has_filename()一样
p.has_extension()	返回路径是否有扩展名
p.has_root_name()	返回路径是否包含根名称
p.has_root_directory()	返回路径是否包含根目录
p.has_root_path()	返回路径是否包含根名称或根目录
p.has_parent_path()	返回路径是否包含父路径
p.has_relative_path()	返回路径是否不止包含根元素
p.filename()	返回文件名（或者空路径）
p.stem()	返回没有扩展名的文件名（或者空路径）
p.extension()	返回扩展名（或者空路径）
p.root_name()	返回根名称（或者空路径）
p.root_directory()	返回根目录（或者空路径）
p.root_path()	返回根元素（或者空路径）
p.parent_path()	返回父路径（或者空路径）
p.relative_path()	返回不带根元素的路径（或者空路径）
p.begin()	返回路径元素的起点
p.end()	返回路径元素的终点

Table 20.4: 检查路径

每一个路径要么是绝对的要么是相对的。如果没有根目录那么路径就是相对的（相对路径也可能包含根名称；例如，C:hello.txt就是Windows下的一个相对路径）。

has_...()函数等价于检查相应的没有has_前缀的函数返回值是否为空路径。

注意下面几点：

- 如果路径含有根元素或者目录分隔符那么就包含父路径。如果路径只由根元素组成（也就是说相对路径为空），`parent_path()` 的返回值就是整个路径。也就是说，路径`"/"` 的父路径还是`"/"`。只有纯文件名的路径例如 `"hello.txt"` 的父路径是空。
- 如果一个路径包含文件名那么一定包含 stem（文件名中不带扩展名的部分）。[2]
- 空路径是相对路径（除了 `is_empty()` 和 `is_relative()` 之外的操作都返回 `false` 或者空路径）。

这些操作的结果可能会依赖于操作系统。例如，路径 `C:/hello.txt`

- 在 Unix 系统上
  - 是相对路径
  - 没有根元素（既没有根名称也没有根目录），因为 `C:` 只是一个文件名
  - 有父路径 `C:`
  - 有相对路径 `C:/hello.txt`
- 在 Windows 系统上
  - 是绝对的
  - 有根名称 `C:` 和根目录 `/`
  - 沒有父路径
  - 有相对路径 `hello.txt`

迭代路径

你可以迭代一个路径，这将会返回路径的所有元素：根名称（如果有的话）、根目录（如果有的话）、所有的文件名。如果路径以目录分隔符结尾，最后的元素将是空文件名。[3]

路径迭代器是双向迭代器，所以你可以递减它。迭代器的值的类型是 `path`。然而，两个在同一个路径上路径上迭代的迭代器可能不指向同一个 `path` 对象，即使它们迭代到了相同的路径元素。

例如，考虑：

```
void printPath(const std::filesystem::path& p)
{
    std::cout << "path elements of \"" << p.string() << "\":\n";
    for (std::filesystem::path elem : p) {
        std::cout << "  \"" << elem.string() << '"';
    }
    std::cout << '\n';
}
```

和如下代码效果相同：

```
void printPath(const std::filesystem::path& p)
{
    std::cout << "path elements of \"" << p.string() << "\":\n";
    for (auto pos = p.begin(); pos != p.end(); ++pos) {
        std::filesystem::path elem = *pos;
```

[2] 从 C++17 才开始这样，因为以前文件名可以只包含扩展名。

[3] 在 C++17 之前，文件系统库使用 . 来表示结尾的目录分隔符。更改这个行为是为了区分以路径分隔符结尾的路径和以路径分隔符后还有一个点结尾的路径。

```
            std::cout << "  \"" << elem.string() << '"';
        }
        std::cout << '\n';
    }
```

如果像下面这样调用这个函数：

```
printPath("../sub/file.txt");
printPath("/usr/tmp/test/dir/");
printPath("C:\\usr\\tmp\\test\\dir\\");
```

在 POSIX 兼容系统上的输出将会是：

```
path elements of "../sub/file.txt":
  ".."  "sub"  "file.txt"
path elements of "/usr/tmp/test/dir/":
  "/"  "usr"  "tmp"  "test"  "dir"  ""
path elements of "C:\\usr\\tmp\\test\\dir\\":
  "C:\\usr\\tmp\\test\\dir\\"
```

注意最后一个路径只是一个文件名，因为在 POSIX 兼容系统上 `C:` 不是有效的根名称，反斜杠也不是有效的目录分隔符。

在 Windows 上的输出将是：

```
path elements of "../sub/file.txt":
  ".."  "sub"  "file.txt"
path elements of "/usr/tmp/test/dir/":
  "/"  "usr"  "tmp"  "test"  "dir"  ""
path elements of "C:\usr\tmp\test\dir\":
  "C:"  "\"  "usr"  "tmp"  "test"  "dir"  ""
```

为了检查路径 `p` 是否以目录分隔符结尾，你可以这么写：

```
if (!p.empty() && (--p.end())->empty()) {
    std::cout << p << " has a trailing separator\n";
}
```

20.3.3 路径 I/O 和转换

表路径 I/O 和转换列出了路径的读写操作和转换操作。这些函数也不会访问实际的文件系统。如果你必须要处理符号链接，你可能要使用 依赖文件系统的路径转换。

`lexically_...()` 函数会返回一个新的路径，而其他的转换函数将返回相应的字符串类型。所有这些函数都不会修改调用者的路径。

例如，下面的代码：

```
std::filesystem::path p{"/dir/./sub//sub1/../sub2"};
std::cout <<  "path:                " << p << '\n';
std::cout <<  "string():            " << p.string() << '\n';
std::wcout << "wstring():           " << p.wstring() << '\n';
std::cout <<  "lexically_normal(): " << p.lexically_normal() << '\n';
```

调用	效果
`strm << p`	用双引号括起来输出路径
`strm >> p`	读取用双引号括起来的路径
`p.string()`	以 `std::string` 返回路径
`p.wstring()`	以 `std::wstring` 返回路径
`p.u8string()`	以类型为 `std::u8string` 的 UTF-8 字符串返回路径
`p.u16string()`	以类型为 `std::u16string` 的 UTF-16 字符串返回路径
`p.u32string()`	以类型为 `std::u32string` 的 UTF-32 字符串返回路径
`p.string<...>()`	以 `std::basic_string` 返回路径
`p.lexically_normal()`	返回正规化的路径
`p.lexically_relative(p2)`	返回从 `p2` 到 `p` 的相对路径（如果没有则返回空路径）
`p.lexically_proximate(p2)`	返回从 `p2` 到 `p` 的路径（如果没有则返回 `p`）

Table 20.5: 路径 I/O 和转换

前三行的输出是相同的：

```
path                "/dir/./sub//sub1/../sub2"
string():           /dir/./sub//sub1/../sub2
wstring():          /dir/./sub//sub1/../sub2
```

但最后一行的输出就依赖于目录分隔符了。在 POSIX 兼容系统上输出是：

```
lexically_normal(): "/dir/sub/sub2"
```

而在 Windows 上输出是：

```
lexically_normal(): "\\dir\\sub\\sub2"
```

路径 I/O

首先，注意 I/O 运算符以双引号括起来的字符串方式读写路径。你可以把它们转换为字符串来避免双引号：

```
std::filesystem::path file{"test.txt"};
std::cout << file << '\n';          // 输出："test.txt"
std::cout << file.string() << '\n'; // 输出：test.txt
```

在 Windows 上，情况可能会更糟糕。下面的代码：

```
std::filesystem::path tmp{"C:\\Windows\\Temp"};
std::cout << tmp << '\n';
std::cout << tmp.string() << '\n';
std::cout << '"' << tmp.string() << "\"\n";
```

将会有如下输出：

```
"C:\\Windows\\Temp"
C:\Windows\Temp
"C:\Windows\Temp"
```

注意读取路径时既支持带双引号的字符串也支持不带双引号的字符串。因此，所有的输出形式都能使用输入运算符再读取回来：

```
std::filesystem::path tmp;
std::cin >> tmp;    // 读取有双引号和无双引号的路径
```

正规化

当你处理可移植代码时正规化可能会导致更多令人惊奇的结果。例如：

```
std::filesystem::path p2{"//host\\dir/sub\\/./\\"};
// 译者注：此处原文是
// std::filesystem::path p2{"//dir\\subdir/subsubdir\\/./\\"};
// 应是作者笔误

std::cout << "p2:                 " << p2 << '\n';
std::cout << "lexically_normal(): " << p2.lexically_normal() << '\n';
```

在 Windows 系统上可能会有如下输出：

```
p2:                 "//host\\dir/sub\\/./\\"
lexically_normal(): "\\\\host\\dir\\sub\\"
```

然而，在 POSIX 兼容系统上，输出将是：

```
p2:                 "//host\\dir/sub\\/./\\"
lexically_normal(): "/host\\dir/sub\\/\\"
```

原因是对于 POSIX 兼容系统来说反斜杠既不是路径分隔符也不是有效的根名称，这意味着我们得到了一个有三个文件名的绝对路径，三个文件名分别是`host\dir`、`sub\`、`\`。在 POSIX 兼容系统上，没有办法把反斜杠作为目录分隔符处理（`generic_string()`和`make_preferred()`也没有用）。因此，对于可移植的代码，当处理路径时你应该总是使用通用路径格式。

但是，当迭代当前路径时使用`lexically_normal()`移除开头的点是个好方法。

相对路径

`lexically_relative()`和`lexically_proximate()`都可以被用来计算两个路径间的相对路径。不同之处在于如果没有相对路径时的行为，只有当一个是相对路径一个是绝对路径或者两个路径的根名称不同时才会发生这种情况。这种情况下：

- 对于`p.lexically_relative(p2)`，如果没有从`p2`到`p`的相对路径，会返回空路径。
- 对于`p.lexically_proximate(p2)`，如果没有从`p2`到`p`的相对路径，会返回`p`。

因为这两个操作都是词法操作，所以不会考虑实际的文件系统（可能会有符号链接）和`current_path()`。如果两个路径相同，相对路径将是`"."`。例如：

```
fs::path{"/a/d"}.lexically_relative("/a/b/c");      // "../../d"
fs::path{"/a/b/c"}.lexically_relative("/a/d");      // "../b/c"
fs::path{"/a/b"}.lexically_relative("/a/b");        // "."
fs::path{"/a/b"}.lexically_relative("/a/b/");       // "."
```

```
fs::path{"/a/b"}.lexically_relative("/a/b\\");      // "."
fs::path{"/a/b"}.lexically_relative("/a/d/../c");   // "../b"
fs::path{"a/d/../b"}.lexically_relative("a/c");     // "../d/../b"
fs::path{"a//d/..//b"}.lexically_relative("a/c");   // "../d/../b"
```

在 Windows 平台上，将会有：

```
fs::path{"C:/a/b"}.lexically_relative("c:/c/d");    // ""
fs::path{"C:/a/b"}.lexically_relative("D:/c/d");    // ""
fs::path{"C:/a/b"}.lexically_proximate("D:/c/d");   // "C:/a/b"
```

转换为字符串

通过 `u8string()` 你可以将路径用作 UTF-8 字符串，这是当前存储数据的通用格式。例如：

```
// 把路径存储为UTF-8字符串：
std::vector<std::string> utf8paths; // 自从C++20起将改为std::u8string
for (const auto& entry : fs::directory_iterator(p)) {
    utf8paths.push_back(entry.path().u8string());
}
```

注意自从 C++20 起 `u8string()` 的返回值可能会从 `std::string` 改为 `std::u8string()`（新的 UTF-8 字符串类型和存储 UTF-8 字符的 `char8_t` 类型的提案在 https://wg21.link/p0482）。[4]

成员模板 `string<>()` 可以用来转换成特殊的字符串类型，例如一个大小写无关的字符串类型：

```
struct ignoreCaseTraits : public std::char_traits<char> {
    // 大小写不敏感的比较两个字符：
    static bool eq(const char& c1, const char& c2) {
        return std::toupper(c1) == std::toupper(c2);
    }
    static bool lt(const char& c1, const char& c2) {
        return std::toupper(c1) < std::toupper(c2);
    }
    // 比较s1和s2的至多前n个字符：
    static int compare(const char* s1, const char* s2, std::size_t n);
    // 搜索s中的字符c：
    static const char* find(const char* s, std::size_t n, const char& c);
};

// 定义一个这种类型的字符串：
using icstring = std::basic_string<char, ignoreCaseTraits>;

std::filesystem::path p{"/dir\\subdir/subsubdir\\/./\\"};
icstring s2 = p.string<char, ignoreCaseTraits>();
```

注意你不应该使用函数 `c_str()`，因为它会转换为本地字符串格式，例如可能会使用 `wchar_t`，因此你需要使用 `std::wcout` 代替 `std::cout` 来输出到输出流。

[4] 感谢 Tom Honermann 指出这一点，并做出这个改进（C++ 开始提供真正的 UTF-8 支持是非常重要的）。

20.3.4 本地和通用格式的转换

表本地和通用格式的转换列出了在通用路径格式 和实际平台特定实现的格式之间转换的方法。

调用	效果
`p.generic_string()`	返回 `std::string` 类型的通用路径
`p.generic_wstring()`	返回 `std::wstring` 类型的通用路径
`p.generic_u8string()`	返回 `std::u8string` 类型的通用路径
`p.generic_u16string()`	返回 `std::u16string` 类型的通用路径
`p.generic_u32string()`	返回 `std::u32string` 类型的通用路径
`p.generic_string<...>()`	返回 `std::basic_string<...>()` 类型的通用路径
`p.native()`	返回 `path::string_type` 类型的本地路径格式
到本地路径的转换	到本地字符类型的隐式转换
`p.c_str()`	返回本地字符串格式的字符序列形式的路径
`p.make_preferred()`	把 `p` 中的目录分隔符替换为本地格式的分隔符并返回修改后的 `p`

Table 20.6: 本地和通用格式的转换

这些函数在 POSIX 兼容系统上没有效果，因为这些系统的本地格式和通用格式没有区别。在其他平台上调用这些函数可能会有效果：

- `generic...()` 函数返回转换为通用格式之后的相应类型的字符串。
- `native()` 返回用本地字符串编码的路径，其类型为 `std::filesystem::path::string_type`。这个类型在 Windows 下是 `std::wstring`，这意味着你需要使用 `std::wcout` 代替 `std::cout` 来输出。新的重载允许我们向文件流传递本地字符串。
- `c_str()` 以空字符结尾的字符序列形式返回结果。注意使用这个函数是不可移植的，因为使用 `std::cout` 打印字符序列在 Windows 上的输出不正确。你应该使用 `std::wcout`。
- `make_preferred()` 会使用本地的目录分隔符替换除了根名称之外的所有的目录分隔符。注意这是唯一一个会修改调用者的函数。因此，严格来讲，这个函数应该属于下一节的修改路径的函数，但是因为它是处理本地格式的转换，所以也在这里列出。

例如，在 Windows 上，下列代码：

```
std::filesystem::path p{"/dir\\subdir/subsubdir\\/./\\"};
std::cout <<  "p:                 " << p << '\n';
std::cout <<  "string():          " << p.string() << '\n';
std::wcout << "wstring():         " << p.wstring() << '\n';
std::cout <<  "lexically_normal(): " << p.lexically_normal() << '\n';
std::cout <<  "generic_string():   " << p.generic_string() << '\n';
std::wcout << "generic_wstring():  " << p.generic_wstring() << '\n';
// 因为这是在Windows下，相应的本地字符串类型是wstring：
std::wcout << "native():          " << p.native() << '\n'; // Windows!
std::wcout << "c_str():           " << p.c_str() << '\n';
std::cout <<  "make_preferred():   " << p.make_preferred() << '\n';
std::cout <<  "p:                 " << p << '\n';
```

将会有如下输出：

```
p:                  "/dir\\subdir/subsubdir\\/./\\"
string():           /dir\subdir/subsubdir\/./\
wstring():          /dir\subdir/subsubdir\/./\
lexically_normal(): "\\dir\\subdir\\subsubdir\\"
generic_string():   /dir/subdir/subsubdir//.//
generic_wstring():  /dir/subdir/subsubdir//.//
native():           /dir\subdir/subsubdir\/./\
c_str():            /dir\subdir/subsubdir\/./\
make_preferred():   "\\dir\\subdir\\subsubdir\\\\.\\\\"
p:                  "\\dir\\subdir\\subsubdir\\\\.\\\\"
```

再次注意：

- 本地字符串格式是不可移植的。在 Windows 上是 `wstring`，而在 POSIX 兼容系统上是 `string`，这意味着你要使用 `cout` 而不是 `wcout` 来打印 `native()` 的结果。
- 只有 `make_preferred()` 会修改调用者。其他的调用都会保持 `p` 不变。

20.3.5 修改路径

表修改路径列出了可以直接修改路径的操作。

注意 `+=` 和 `concat()` 简单的把字符添加到路径后，`/`、`/=`、`append()` 则是在路径后用目录分隔符添加一个子路径：

```
std::filesystem::path p{"myfile"};
p += ".git";        // p:myfile.git
p /= ".git";        // p:myfile.git/.git
p.concat("1");      // p:myfile.git/.git1
p.append("1");      // P:myfile.git/.git1/1
std::cout << p << '\n';
std::cout << p / p << '\n';
```

在 POSIX 兼容系统上输出将是：

```
"myfile.git/.git1/1"
"myfile.git/.git1/1/myfile.git/.git1/1"
```

在 Windows 系统上输出将是：

```
"myfile.git\\.git1\\1"
"myfile.git\\.git1\\1\\myfile.git\\.git1\\1"
```

注意如果添加一个绝对路径子路径意味着替换原本的路径。例如，如下操作之后：

```
namespace fs = std::filesystem;
auto p1 = fs::path("/usr") / "tmp";     // 路径是/usr/tmp或者/usr\tmp
auto p2 = fs::path("/usr/") / "tmp";    // 路径是/usr/tmp
auto p3 = fs::path("/usr") / "/tmp";    // 路径是/tmp
auto p4 = fs::path("/usr/") / "/tmp";   // 路径是/tmp
```

我们有了四个指向两个不同文件的路径：

调用	效果
`p = p2`	赋予一个新路径
`p = sv`	赋予一个字符串（视图）作为新路径
`p.assign(p2)`	赋予一个新路径
`p.assign(sv)`	赋予一个字符串（视图）作为新路径
`p.assign(beg, end)`	赋予从 beg 到 end 的元素组成的路径
`p1 / p2`	返回把 p2 作为子路径附加到 p1 之后的结果
`p /= sub`	把 sub 作为子路径附加到路径 p 之后
`p.append(sub)`	把 sub 作为子路径附加到路径 p 之后
`p.append(beg, end)`	把从 beg 到 end 之间的元素作为子路径附加到路径 p 之后
`p += str`	把 str 里的字符添加到路径 p 之后
`p.concat(str)`	把 str 里的字符添加到路径 p 之后
`p.concat(beg, end)`	把从 beg 到 end 之间的元素附加到路径 p 之后
`p.remove_filename()`	移除路径末尾的文件名
`p.replace_filename(repl)`	替换末尾的文件名（如果有的话）
`p.replace_extension()`	移除末尾的文件的扩展名
`p.replace_extension(repl)`	替换末尾的文件的扩展名（如果有的话）
`p.clear()`	清空路径
`p.swap(p2)`	交换两个路径
`swap(p1, p2)`	交换两个路径
`p.make_preferred()`	把 p 中的目录分隔符替换为本地格式的分隔符并返回修改后的 p

Table 20.7: 修改路径

- p1 和 p2 相等，都指向文件 /usr/tmp（注意在 Windows 上它们相等，但 p1 将是 /usr\tmp。
- p3 和 p4 相等，指向文件 /tmp，因为附加的子路径是绝对路径。

对于根元素，是否赋予新的根元素的结果将不同。例如，在 Windows 上，结果将是：

```
auto p1 = fs::path("usr") / "C:/tmp";   // 路径是C:/tmp
auto p2 = fs::path("usr") / "C:";       // 路径是C:
auto p3 = fs::path("C:") / "":          // 路径是C:
auto p4 = fs::path("C:usr") / "/tmp";   // 路径是C:/tmp
auto p5 = fs::path("C:usr") / "C:tmp";  // 路径是C:usr\tmp
auto p6 = fs::path("C:usr") / "c:tmp";  // 路径是c:tmp
auto p7 = fs::path("C:usr") / "D:tmp";  // 路径是D:tmp
```

20.3.6 比较路径

20.3.7 其他路径操作

20.4 文件系统操作

20.4.1 文件属性

检查文件是否存在

其他文件属性

处理最后修改的时间

20.4.2 文件状态

20.4.3 权限

20.4.4 修改文件系统

创建和删除文件

修改已存在的文件

20.4.5 符号链接和依赖文件系统的路径转换

20.4.6 其他文件系统操作

20.5 迭代目录

传入的参数 `dir` 可以是一个 `path` 也可以是其他可以隐式转换为路径的类型（特指各种形式的字符串）。

目录迭代器表示一个范围

目录迭代器选项

20.5.1 目录项

目录项缓存

20.6 后记

Part IV

已有标准库的拓展和修改

这一部分介绍 C++17 对已有标准库组件的拓展和修改。

Part V

专家的工具

这一部分介绍了普通应用程序员通常不需要知道的新的语言特性和库。它主要包括了为编写基础库和语言特性的程序员准备的用来解决特殊问题语言特性（例如修改了堆内存的管理方式）。

Part VI

一些通用的提示

这一部分介绍了一些有关C++17的通用的提示，例如对C语言和废弃特性的兼容性更改。