简介

- C++函数常用函数

std::remove

- 简介：
  - C++11中的std::remove函数属于标准库的文件操作函数，用于删除指定的文件
- 原型：

```
int remove(const char* filename);
```

- 参数：
  - filename：要删除的文件的路径（C风格的字符串）
- 返回值：
  - 如果文件成功删除，则返回0。
  - 如果删除文件失败，则返回非零值
- 详解：
  - std::remove函数用于删除指定的文件。它将指定的文件路径作为参数，并尝试将文件从文件系统中永久删除。如果文件删除成功，返回值为0，否则返回非零值
  - std::remove函数不会将文件移动到回收站，而是直接从文件系统中删除文件
  - 如果要删除的文件不存在或无法删除，则会返回非零值，可以通过检查返回值来判断删除操作是否成功
- 示例：

```
#include <iostream>
#include <cstdio>

int main() {
    const char* filename = "/path/to/file.txt"; // 指定文件路径

    if (std::remove(filename) != 0) {
        std::cerr << "Failed to delete file: " << filename << std::endl;
    } else {
        std::cout << "File deleted: " << filename << std::endl;
    }

    return 0;
}
```

- 注：
  - 在上述示例中，我们使用std::remove函数来删除指定路径的文件。我们传递要删除的文件路径作为参数，并检查返回值是否为0来确定是否删除成功
  - 请注意，std::remove函数只能删除文件，不能删除目录。如果要删除目录及其内容，您可以使用其他方法，例如递归删除目录中的所有文件，然后再删除空的目录

---

statvfs

- 简介：
  - statvfs() 是一个Linux系统调用函数，用于获取文件系统的相关信息
- 原型：

```
int statvfs(const char *path, struct statvfs *buf);
```

- 参数：
  - statvfs() 函数接受两个参数：path 是文件系统路径名，buf 是指向 struct statvfs 结构的指针，用于存储文件系统信息
- 详解：
  - struct statvfs 结构包含了多个字段，提供了关于文件系统的各种属性。下面是 struct statvfs 的定义：

```
struct statvfs {
    unsigned long f_bsize;    /* 文件系统块大小 */
    unsigned long f_frsize;   /* 文件系统的基本块大小 */
    fsblkcnt_t    f_blocks;   /* 文件系统总块数 */
    fsblkcnt_t    f_bfree;    /* 文件系统空闲块数 */
    fsblkcnt_t    f_bavail;   /* 非超级用户可用块数 */
    fsfilcnt_t    f_files;    /* 文件系统节点数 */
    fsfilcnt_t    f_ffree;    /* 文件系统空闲节点数 */
    fsfilcnt_t    f_favail;   /* 非超级用户可用节点数 */
    unsigned long f_fsid;     /* 文件系统标识符 */
    unsigned long f_flag;     /* 挂载标志 */
    unsigned long f_namemax;  /* 最大文件名长度 */
};
```

- 常用字段解释：
  - f_bsize：文件系统块的大小（字节）
  - f_frsize：文件系统的基本块大小（字节）
  - f_blocks：文件系统的总块数
  - f_bfree：文件系统的空闲块数
  - f_bavail：非超级用户可用的块数

    - f_files：文件系统的节点（文件）数
    - f_ffree：文件系统的空闲节点数
    - f_favail：非超级用户可用的节点数
    - f_fsid：文件系统标识符
    - f_flag：文件系统的挂载标志
    - f_namemax：文件系统支持的最大文件名长度

- 详解：

    - 通过调用 statvfs() 函数并传递文件系统路径名和一个 struct statvfs 结构指针，可以获取与指定文件系统相关的各种信息。如果函数调用成功，返回值为 0，否则返回值为 -1，表示失败。在失败的情况下，可以使用 errno 来获取具体的错误码
    - 使用 statvfs() 函数，您可以获取文件系统的块大小、总空间、可用空间、文件节点数等信息，以便进行磁盘空间管理和监控等操作

sysinfo

- 简介：

    - sysinfo() 函数是一个系统调用，用于获取关于系统整体状态的信息，包括内存使用情况、负载信息、运行时间等。它位于 <sys/sysinfo.h> 头文件中

- 原型：

```
int sysinfo(struct sysinfo *info);
```

- 详解：
    - 该函数接受一个指向 sysinfo 结构的指针作为参数，并将系统信息填充到该结构中。sysinfo 结构定义如下

```
struct sysinfo {
    long uptime;             // 系统运行时间
    unsigned long loads[3];  // 1分钟、5分钟、15分钟的负载平均值
    unsigned long totalram;  // 总内存大小（字节）
    unsigned long freeram;   // 可用内存大小（字节）
    unsigned long sharedram; // 共享内存大小（字节）
    unsigned long bufferram; // 缓冲区大小（字节）
    unsigned long totalswap; // 总交换空间大小（字节）
    unsigned long freeswap;  // 可用交换空间大小（字节）
    unsigned short procs;    // 进程数量
    char _f[22];             // 保留字段，不使用
};
```

- 返回值：

    - sysinfo() 函数将系统的状态信息填充到传入的 sysinfo 结构中，然后返回 0 表示成功，-1 表示失败

- 详解-2：
    - 通过调用 sysinfo() 函数并传入一个 sysinfo 结构的指针，可以获取系统的各种信息，如系统运行时间、负载平均值、内存使用情况等。在获取内存信息时，可以使用 totalram 字段获取总内存大小，freeram 字段获取可用内存大小
- 示例：

```
#include <sys/sysinfo.h>
#include <iostream>

int main() {
    struct sysinfo sys_info;
    if (sysinfo(&sys_info) != -1) {
        unsigned long total_memory = sys_info.totalram;
        unsigned long free_memory = sys_info.freeram;

        std::cout << "Total Memory: " << total_memory << std::endl;
        std::cout << "Free Memory: " << free_memory << std::endl;
    } else {
        std::cerr << "Failed to retrieve system memory information." <<
std::endl;
    }

    return 0;
}
```

- 注：
    - 请注意，sysinfo() 函数返回的内存大小是以字节为单位的整数值。如果需要以更友好的方式显示内存大小（如以MB或GB为单位），可以进行相应的单位转换操作。

prctl

- 简介：
    - prctl() 是一个 Linux 系统调用函数，用于控制进程的各种属性和行为。它可以用于获取和修改进程的运行时属性，包括名称、资源限制、父子关系、信号处理等
- 原型：

```
#include <sys/prctl.h>

int prctl(int option, unsigned long arg2, unsigned long arg3, unsigned long
arg4, unsigned long arg5);
```

- 参数：
    - prctl() 函数的第一个参数 option 用于指定要执行的操作
    - 后面的参数 arg2、arg3、arg4、arg5 则是与特定操作相关的参数。

- 详解：
    - PR_SET_NAME：设置进程名称。
    - PR_GET_NAME：获取进程名称。
    - PR_SET_PDEATHSIG：设置子进程在父进程终止时接收的信号。
    - PR_SET_CHILD_SUBREAPER：设置进程为子进程的子进程收割器。
    - PR_GET_CHILD_SUBREAPER：检查进程是否是子进程的子进程收割器
- 示例：

```
#include <sys/prctl.h>
#include <iostream>
#include <cstring>

int main()
{
    const char* processName = "MyProcess";

    if (prctl(PR_SET_NAME, reinterpret_cast<unsigned long>(processName), 0,
0, 0) != 0)
    {
        std::cerr << "Failed to set process name" << std::endl;
        return 1;
    }

    char buffer[16];
    if (prctl(PR_GET_NAME, reinterpret_cast<unsigned long>(buffer), 0, 0,
0) != 0)
    {
        std::cerr << "Failed to get process name" << std::endl;
        return 1;
    }

    std::cout << "Process name: " << buffer << std::endl;

    return 0;
}
```

- 注：
    - 在上面的示例中，我们首先使用 prctl() 函数将进程的名称设置为 "MyProcess"。然后，我们使用 prctl() 函数的 PR_GET_NAME 参数获取进程的名称，并将其输出到控制台
    - 需要注意的是，prctl() 函数的具体操作和参数在不同的操作系统版本和架构上可能会有所不同。因此，在使用 prctl() 函数时，建议参考相关的文档和系统头文件以了解特定操作和参数的正确使用方式

pthread_setname_np

- 简介：

    - pthread_setname_np() 是一个非标准的 POSIX 函数，用于设置 POSIX 线程的名称。它允许你为线程分配一个描述性的名称，以便在调试和跟踪应用程序时更容易地识别不同的线程

- 原型：

```
#include <pthread.h>

int pthread_setname_np(pthread_t thread, const char *name);
```

- 参数：

    - pthread_setname_np() 函数的第一个参数是要设置名称的线程标识符 thread
    - 第二个参数是要分配给线程的名称字符串 name

- 示例：

```
#include <pthread.h>
#include <iostream>

void* threadFunction(void*)
{
    // 线程执行的任务

    return nullptr;
}

int main()
{
    pthread_t thread;
    const char* threadName = "MyThread";

    // 创建线程
    pthread_create(&thread, nullptr, threadFunction, nullptr);

    // 设置线程名称
    pthread_setname_np(thread, threadName);

    // 主线程执行的任务

    // 等待线程结束
    pthread_join(thread, nullptr);

    return 0;
}
```

- 注：
    - 在上面的示例中，我们首先创建了一个线程 thread，然后使用 pthread_setname_np() 函数将其名称设置为 "MyThread"。

- 需要注意的是，pthread_setname_np() 函数是非标准的，它可能在不同的操作系统和实现中有所差异。在某些平台上，可能需要添加一些特定的宏定义或包含其他头文件来使用该函数。
- 此外，pthread_setname_np() 函数仅用于 POSIX 线程，不能用于其他类型的线程，如 Windows 线程

signal

- 简介：
  - signal() 是一个 C 标准库函数，用于捕捉和处理信号。信号是在操作系统中用于通知进程发生某个事件的机制
- 原型：

```
#include <signal.h>

void (*signal(int signum, void (*handler)(int)))(int);
```

- 参数：
  - signum：一个整数，表示要捕捉的信号的编号。常见的信号包括 SIGINT（中断信号）和 SIGTERM（终止信号），还有其他许多可用的信号。
  - handler：一个函数指针，指向一个处理信号的函数。函数的类型为 void (*)(int)，它接受一个整数作为参数，并返回 void
- 返回值：
  - 返回一个函数指针，指向以前与 signum 相关联的信号处理函数。如果以前没有设置过信号处理函数，则返回 SIG_DFL（默认信号处理）或 SIG_IGN（忽略信号）
- 详解：
  - signal() 函数用于为指定的信号 signum 设置信号处理函数 handler。
  - 要使用 signal() 函数，需要包含头文件 <signal.h>。
  - 通过设置信号处理函数，可以在程序中对特定的信号做出相应的操作，例如捕捉 SIGINT 信号以处理用户中断操作。
  - 当进程接收到指定的信号时，系统将调用与之相关联的信号处理函数来处理该信号。
  - 信号处理函数可以是用户自定义的函数，用于执行特定的操作，或者可以是 SIG_DFL（默认信号处理）或 SIG_IGN（忽略信号）。
- 原型：

```
#include <stdio.h>
#include <signal.h>

// 信号处理函数
void sigintHandler(int signum) {
    printf("Received SIGINT signal. Exiting...\n");
    // 在此处执行清理操作或其他必要的处理
```

```
    // 例如关闭文件、释放资源等

    // 退出程序
    exit(signum);
}

int main() {
    // 设置 SIGINT 信号的处理函数为 sigintHandler
    signal(SIGINT, sigintHandler);

    // 进入主循环
    while (1) {
        // 在此处执行主要的程序逻辑
    }

    return 0;
}
```

- 注：
  - 在此示例中，我们定义了一个名为 sigintHandler() 的信号处理函数，用于处理 SIGINT 信号（通常是通过键盘输入 Ctrl+C 产生的）。在 main() 函数中，我们使用 signal() 函数将 SIGINT 信号与 sigintHandler() 函数关联起来。当程序接收到 SIGINT 信号时，将调用 sigintHandler() 函数来处理该信号，并在处理函数中执行相应的操作。在此示例中，我们简单地打印一条消息并退出程序

---

setpriority

- 简介：
  - setpriority() 是一个 C 标准库函数，用于设置进程的调度优先级。通过调整进程的调度优先级，可以影响操作系统对进程的调度顺序
- 原型：

```
#include <sys/time.h>
#include <sys/resource.h>

int setpriority(int which, id_t who, int prio);
```

- 参数：
  - which：一个整数，指定要设置优先级的对象。可取以下值
    - PRIO_PROCESS：设置进程的优先级。
    - PRIO_PGRP：设置进程组的优先级。
    - PRIO_USER：设置用户的所有进程的优先级
  - who：一个整数，表示要设置优先级的对象的标识符。根据 which 参数的取值，可有不同的解释：
    - 当 which 为 PRIO_PROCESS 时，who 是进程ID（PID）。
    - 当 which 为 PRIO_PGRP 时，who 是进程组ID（PGID）。

        - 当 which 为 PRIO_USER 时，who 是用户ID（UID）
    - prio：一个整数，表示要设置的优先级。优先级的范围通常是 -20 到 19，其中 -20 表示最高优先级，19 表示最低优先级。

- 返回值：
    - 成功时，返回 0。
    - 失败时，返回 -1，并设置 errno 变量表示错误的原因

- 详解：
    - setpriority() 函数用于设置进程的调度优先级。
    - 要使用 setpriority() 函数，需要包含头文件 <sys/time.h> 和 <sys/resource.h>。
    - 进程的调度优先级影响了操作系统对进程的调度顺序。更高的优先级意味着进程更有可能被操作系统选择进行执行。
    - setpriority() 函数需要适当的权限来设置进程的优先级。通常，只有超级用户（root）或具有特权的用户才能更改其他进程的优先级。
    - 可以使用 getpriority() 函数来获取进程的当前优先级。
    - 设置进程的优先级可能会受到操作系统和调度器的限制，具体的行为可能因操作系统的实现而异

- 示例：

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>

int main() {
    // 获取当前进程ID
    pid_t pid = getpid();

    // 设置进程的优先级为最高优先级
    int priority = -20;
    int result = setpriority(PRIO_PROCESS, pid, priority);
    if (result == -1) {
        perror("setpriority");
        return 1;
    }

    printf("Process priority set to %d\n", priority);

    return 0;
}
```

- 注：
    - 在此示例中，我们使用 setpriority() 函数将当前进程的优先级设置为最高优先级（-20）。请注意，设置进程的优先级可能需要适当的权限，并且对于不同的操作系统和调度器，行为可能会有所不同。因此，具体的效果和行为可能因系统而异

std::uniform_int_distribution

`std::uniform_int_distribution`是C++标准库中的一个分布类，用于生成指定范围内的均匀分布的整数随机数。

以下是一些关键点来详细解释`std::uniform_int_distribution`的用法和特性：

1. `std::uniform_int_distribution`的构造函数：

```
explicit uniform_int_distribution(IntType a = 0, IntType b =
std::numeric_limits<IntType>::max());
```

构造一个`std::uniform_int_distribution`对象。参数`a`和`b`用于指定生成随机数的范围，生成的随机数将包含`a`和`b`。

2. `std::uniform_int_distribution`的成员函数`operator()()`：

```
result_type operator()(Engine& eng);
```

通过调用`operator()`函数，可以生成一个随机数。需要提供一个随机数引擎对象`eng`作为参数，该引擎将用于生成随机数。

3. `std::uniform_int_distribution`的成员函数`param()`和`param(const param_type& parm)`：

```
param_type param() const;
void param(const param_type& parm);
```

`param()`函数用于获取当前分布对象的参数，即范围的上下界。`param(const param_type& parm)`函数用于设置分布对象的参数。

4. `std::uniform_int_distribution`对象的用法示例：

```
#include <iostream>
#include <random>

int main() {
    std::random_device rd;
    std::mt19937 rng(rd());

    std::uniform_int_distribution<int> dist(1, 10);

    for (int i = 0; i < 5; ++i) {
        int randomNumber = dist(rng);
        std::cout << "Random number: " << randomNumber << std::endl;
    }
```

```
    return 0;
}
```

在上述示例中，首先创建一个`std::random_device`对象`rd`用于获取随机种子。然后，创建一个`std::mt19937`对象`rng`作为随机数引擎。接下来，创建一个`std::uniform_int_distribution<int>`对象`dist`，表示生成1到10之间（包括1和10）的整数随机数。最后，通过在循环中调用`dist(rng)`生成随机数，并将其打印输出。

`std::uniform_int_distribution`用于生成均匀分布的整数随机数，确保生成的随机数在指定范围内的分布是均匀的。它通常与随机数引擎（如`std::mt19937`）配合使用，可以满足许多随机数生成的需求。

std::mt19937

`std::mt19937`是C++标准库中的一个随机数引擎类，用于生成高质量的伪随机数序列。它基于梅森旋转算法（Mersenne Twister）实现，并且具有良好的随机性和周期性。

以下是一些关键点来详细解释`std::mt19937`的用法和特性：

1. `std::mt19937`的构造函数：

```
explicit mt19937();
explicit mt19937(unsigned int seed);
template <class Sseq> explicit mt19937(Sseq& q);
```

`std::mt19937`提供了几种不同的构造函数。默认构造函数会使用默认的种子值初始化随机数引擎。第二个构造函数接受一个无符号整数作为种子值。第三个构造函数接受一个序列生成器对象作为参数，并使用该序列生成器生成种子序列。

2. `std::mt19937`的成员函数`operator()()`：

```
result_type operator()();
```

通过调用`operator()`函数，可以生成一个随机数。返回的随机数的类型由`result_type`定义。

3. `std::mt19937`的成员函数`seed()`：

```
void seed(unsigned int seed);
```

通过调用`seed()`函数，可以重新设置随机数引擎的种子值。

4. `std::mt19937`的成员函数`discard()`：

```
void discard(unsigned long long z);
```

`discard()`函数用于丢弃指定数量的随机数。这在某些情况下可以用于跳过初始的随机数，以获得更随机的序列。

5. `std::mt19937`对象的用法示例：

```
#include <iostream>
#include <random>

int main() {
    std::mt19937 rng;

    // 设置种子值
    rng.seed(42);

    // 生成随机数
    int randomNumber = rng();

    std::cout << "Random number: " << randomNumber << std::endl;

    return 0;
}
```

在上述示例中，首先创建一个`std::mt19937`对象`rng`。然后，通过调用`rng.seed()`来设置种子值，这里使用了常数值42作为种子。最后，通过调用`rng()`生成一个随机数，并将其打印输出。

`std::mt19937`是一个高质量的随机数引擎，通常比`std::rand()`等传统的随机数生成函数产生更好的随机数。它在C++11标准中被引入，用于替代了旧的随机数引擎。

std::random_device

`std::random_device`是C++标准库中的一个类，用于生成真正的随机数。它是一个非确定性随机数生成器，通常通过操作系统提供的熵源来获取随机数。

`std::random_device`类定义在头文件 `<random>` 中，用于产生高质量的随机数，适用于密码学、模拟和其他需要真正随机性的应用场景。

以下是一些关键点来详细解释`std::random_device`的用法和特性：

1. `std::random_device`的构造函数：

```
explicit random_device(const string& token = implementation-defined);
```

构造一个`std::random_device`对象。可选的`token`参数用于指定特定的随机数生成器实现。

2. `std::random_device`的成员函数`operator()`：

```
result_type operator()();
```

通过调用`operator()`函数，可以生成一个随机数。返回的随机数的类型由`result_type`定义。

3. `std::random_device`的成员函数`entropy()`：

```
result_type entropy() const;
```

返回随机数生成器的熵值，表示生成的随机数的质量。如果熵值为0，表示随机数生成器没有可用的熵源。

4. `std::random_device`对象的用法示例：

```
#include <iostream>
#include <random>

int main() {
    std::random_device rd;

    // 生成一个随机数
    int randomNumber = rd();

    std::cout << "Random number: " << randomNumber << std::endl;
    std::cout << "Entropy: " << rd.entropy() << std::endl;

    return 0;
}
```

在上述示例中，首先创建一个`std::random_device`对象`rd`。然后，通过调用`rd()`来生成一个随机数，并将其打印输出。最后，使用`rd.entropy()`获取熵值，并将其打印输出。

需要注意的是，由于`std::random_device`是非确定性的，它可能会依赖于操作系统的随机数生成器实现。因此，不同的操作系统和编译器可能会有不同的行为。在某些情况下，`std::random_device`可能返回伪随机数，而不是真正的随机数。如果您需要高质量的随机数，请考虑使用其他的随机数生成器，如`std::mt19937`。

std::move

在C++中，`std::move()`是一个函数模板，位于`<utility>`头文件中。它用于将对象的状态从一个对象转移到另一个对象，通常用于实现高效的移动语义。

`std::move()`函数将给定的对象转换为右值引用，使得可以使用移动语义来操作该对象。移动语义是C++11引入的一个重要特性，允许在对象转移所有权时避免不必要的复制操作，提高性能。

以下是`std::move()`函数的一般语法：

```
std::move(argument);
```

其中，argument是要移动的对象。

使用std::move()函数的主要目的是将对象的状态从一个对象转移到另一个对象，而不进行深拷贝操作。这对于移动构造函数和移动赋值运算符非常有用，可以显著提高对象的性能。

以下是std::move()函数的一些常见用法：

1. 移动语义的实现：

```
// 移动构造函数
MyClass(MyClass&& other) : data(std::move(other.data)) {
    // ...
}

// 移动赋值运算符
MyClass& operator=(MyClass&& other) {
    data = std::move(other.data);
    // ...
    return *this;
}
```

在移动构造函数和移动赋值运算符中，通过使用std::move()函数，将数据从一个对象移动到另一个对象，避免了不必要的复制。

2. 传递参数的移动语义：

```
void processData(std::vector<int>&& data) {
    // 对移动后的data进行操作
}

std::vector<int> vec = {1, 2, 3};
processData(std::move(vec));
```

在此示例中，通过使用std::move()函数，将vec对象转换为右值引用，并将其作为参数传递给processData()函数。这样可以避免不必要的拷贝操作。

需要注意的是，std::move()函数只是将对象转换为右值引用，本身并不进行实际的数据移动。实际的移动操作需要由移动构造函数或移动赋值运算符来实现。

使用std::move()函数需要谨慎操作，确保正确处理移动后的对象状态，避免悬空引用和使用已经移动的对象。

std::localtime

- 简介：
    - std::localtime() 是 C++ 标准库中的一个函数，用于将时间值（通常是由 std::time() 函数获得的时间戳）转换为本地时间的结构

- 原型：

```
#include <ctime>

std::tm* std::localtime(const std::time_t* timePtr);
```

- 参数：
  - timePtr：一个指向 std::time_t 类型变量的指针，表示要转换为本地时间的时间值
- 返回值：
  - 如果成功转换为本地时间，则返回一个指向 std::tm 结构的指针，其中包含了年、月、日、时、分、秒等时间信息
  - 如果转换失败或 timePtr 为 nullptr，则返回 nullptr
- 详解：
  - std::tm 结构包含了以下成员：

```
struct tm {
    int tm_sec;   // 秒 (0 ~ 59)
    int tm_min;   // 分 (0 ~ 59)
    int tm_hour;  // 小时 (0 ~ 23)
    int tm_mday;  // 一个月的日期 (1 ~ 31)
    int tm_mon;   // 月份 (0 ~ 11，0 表示 1 月)
    int tm_year;  // 年份 (自 1900 年起)
    int tm_wday;  // 一周的日期 (0 ~ 6，0 表示星期日)
    int tm_yday;  // 一年的日期 (0 ~ 365，0 表示 1 月 1 日)
    int tm_isdst; // 夏令时标识符 (-1 表示不确定，0 表示不使用夏令时，>0 表示
使用夏令时)
};
```

- 示例：

```
#include <iostream>
#include <ctime>

int main() {
    std::time_t currentTime = std::time(nullptr);
    std::tm* localTime = std::localtime(&currentTime);

    if (localTime != nullptr) {
        std::cout << "本地时间：" << localTime->tm_year + 1900 << "-"
                  << localTime->tm_mon + 1 << "-" << localTime->tm_mday
                  << " " << localTime->tm_hour << ":"
                  << localTime->tm_min << ":" << localTime->tm_sec <<
std::endl;
```

```
    } else {
        std::cout << "转换为本地时间失败" << std::endl;
    }

    return 0;
}
```

- 注：
  - 在上述示例中，我们使用 std::time() 函数获取当前的系统时间，并将其存储在 currentTime 变量中。然后，我们使用 std::localtime() 函数将 currentTime 转换为本地时间的 std::tm 结构，并将其存储在 localTime 指针中
  - 我们可以通过访问 std::tm 结构的成员，如 tm_year、tm_mon、tm_mday 等，获取转换后的本地时间的年、月、日、小时、分钟和秒等信息
  - 需要注意的是，std::localtime() 函数返回的指针指向一个静态分配的 std::tm 结构，因此在下一次调用 std::localtime() 之前应尽快使用转换后的时间信息

std::time

- 简介：
  - std::time() 是 C++ 标准库中的一个函数，用于获取当前的系统时间
- 原型：

```
#include <ctime>

std::time_t std::time(std::time_t* timePtr);
```

- 参数：
  - timePtr：一个指向 std::time_t 类型的变量的指针，用于存储获取的系统时间。可以为 nullptr，表示不存储时间值。
- 返回值：
  - 如果成功获取系统时间，则返回一个 std::time_t 类型的值，表示自 1970 年 1 月 1 日以来的秒数（UTC 时间）
  - 如果获取系统时间失败，则返回 -1
- 示例：

```
#include <iostream>
#include <ctime>

int main() {
    std::time_t currentTime = std::time(nullptr);

    if (currentTime != -1) {
```

```
        std::cout << "当前时间：" <<
std::asctime(std::localtime(&currentTime));
    } else {
        std::cout << "获取系统时间失败" << std::endl;
    }

    return 0;
}
```

- 注：
  - 在上述示例中，我们使用 std::time() 函数获取当前的系统时间，并将其存储在 currentTime 变量中。然后，我们使用 std::localtime() 函数将 currentTime 转换为本地时间，并使用 std::asctime() 函数将本地时间转换为字符串格式进行输出
  - 需要注意的是，std::time() 返回的时间是从 1970 年 1 月 1 日以来经过的秒数，也称为 UNIX 时间戳。为了方便使用和显示，可以使用其他时间处理函数将时间戳转换为可读的日期和时间格式
  - 此外，std::time() 函数返回的时间精度可能取决于操作系统和实现。通常情况下，时间精度为秒级，但有些系统可能提供更高的精度

---

sched_yield

- 简介：
  - sched_yield() 函数是一个库函数，它允许一个正在运行的线程主动放弃 CPU 的使用权，从而使得其他可运行的线程有机会运行
- 原型

```
#include <sched.h>

int sched_yield(void);
```

- 参数
  - sched_yield() 函数没有参数，它的返回值为调用是否成功
- 详解：
  - sched_yield() 函数的作用是将当前线程移出调度器的运行队列，让其他具有相同或更高优先级的可运行线程有机会被调度和执行。简而言之，调用 sched_yield() 函数会主动让出 CPU，给其他线程执行的机会
  - sched_yield() 函数是一种协作式的调度机制。在使用 sched_yield() 之前，内核会保证当前线程在运行队列中是处于就绪状态的。调用 sched_yield() 后，内核会选择一个合适的线程来运行，具体的选择是由调度算法和线程的优先级决定的
  - sched_yield() 函数不能保证当前线程立即放弃 CPU 控制权，也不能保证其他线程立即运行。它仅仅是给其他线程运行的机会，具体的调度还是由内核来决定
  - sched_yield() 函数通常在以下情况下使用：

        - 当线程在忙碌运行一段时间后，如果没有更多的工作可以做，可以调用 sched_yield() 来让其他线程有机会运行，避免资源的浪费
        - 在实现自旋锁或自旋等待时，可以使用 sched_yield() 来减小自旋的开销，避免过度消耗 CPU 资源

- 注：

    - 需要注意的是，sched_yield() 函数只对线程调度起到协作的作用，具体的调度策略和行为由内核来决定，并且在不同的操作系统和调度器中可能会有一些差异。因此，在实际使用中，需要谨慎考虑 sched_yield() 函数的使用时机和效果，并进行充分的测试和验证

lseek

- 简介：

    - 文件的定位操作可以使用 lseek() 函数，该函数可以将文件的当前位置指针移动到指定的位置

- 原型：

```
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

- 参数：

    - fd：文件描述符，表示要进行定位操作的文件。
    - offset：偏移量，表示相对于 whence 参数指定位置的偏移量。
    - whence：定位基准位置，可以是以下值之一
        - SEEK_SET：从文件开头开始计算偏移量
        - SEEK_CUR：从当前位置开始计算偏移量
        - SEEK_END：从文件末尾开始计算偏移量

- 返回值：

    - 如果成功，返回新的文件位置指针的偏移量
    - 如果发生错误，返回值为 -1，并设置 errno 变量来指示具体的错误原因。

- 示例：

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main() {
    int fd = open("file.txt", O_RDONLY);
    if (fd == -1) {
        perror("Failed to open file");
        return 1;
    }
```

```
    off_t offset = lseek(fd, 0, SEEK_SET);
    if (offset == -1) {
        perror("Failed to move file pointer to the beginning");
        close(fd);
        return 1;
    }

    // 现在文件位置指针已经在文件开头，可以进行后续操作

    close(fd);
    return 0;
}
```

- 注：
  - 在上述示例中，我们使用 open() 函数打开文件，并使用 lseek() 函数将文件位置指针移动到文件开头。如果 lseek() 返回值为 -1，表示定位失败，我们可以使用 perror() 函数打印错误消息
  - 需要注意的是，lseek() 函数在处理二进制文件或随机访问文件时非常有用。对于顺序读取文件的情况，可以使用更高级的文件读取函数，如 fseek() 和 rewind()

tcflush

- 简介：
  - tcflush()函数用于刷新串口输入输出缓冲区。它是与串口相关的函数，可用于清空或刷新串口的数据缓冲区
- 原型：

```
int tcflush(int fd, int queue_selector);
```

- 参数：
  - fd：文件描述符，表示要刷新的串口
  - queue_selector：刷新队列的选择器，可以是以下值之一
    - TCIFLUSH：刷新输入队列，丢弃输入缓冲区中的数据
    - TCOFLUSH：刷新输出队列，丢弃输出缓冲区中的数据
    - TCIOFLUSH：刷新输入和输出队列，丢弃输入和输出缓冲区中的数据
- 返回值：
  - 如果刷新成功，返回值为0
  - 如果刷新失败，返回值为-1，并设置errno变量以指示错误类型
- 详解：
  - 使用tcflush()函数可以清空串口的输入和输出缓冲区，以便开始或终止数据传输之前进行必要的清理

- 示例：

```
#include <termios.h>
#include <stdio.h>

int main() {
    int serial_fd; // 串口文件描述符

    // 打开串口并设置相关参数...

    // 清空输入缓冲区
    if (tcflush(serial_fd, TCIFLUSH) == -1) {
        perror("Failed to flush input queue");
        return 1;
    }

    // 清空输出缓冲区
    if (tcflush(serial_fd, TCOFLUSH) == -1) {
        perror("Failed to flush output queue");
        return 1;
    }

    printf("Serial port input and output queues flushed\n");

    // 关闭串口...

    return 0;
}
```

- 注：
  - 在上述示例中，我们首先打开串口并设置相关参数（未在示例中展示）。然后，我们使用tcflush()函数两次调用来分别清空输入和输出缓冲区。最后，我们关闭串口（未在示例中展示）
  - 请注意，上述示例仅演示了刷新串口缓冲区的情况。在实际应用中，你可能还需要进行其他串口操作，如读取或写入数据等。同时，还要注意进行错误处理，以检查函数调用是否成功

cfsetospeed

- 简介：
  - cfsetospeed() 函数用于设置串口设备的输出波特率。它是与串口相关的函数，用于配置串口的通信参数
- 原型：

```
int cfsetospeed(struct termios *termios_p, speed_t speed);
```

- 参数：
  - termios_p：指向 termios 结构的指针，该结构包含了串口的配置参数

    - speed：要设置的输出波特率

- 返回值：

    - 如果设置成功，返回值为 0
    - 如果设置失败，返回值为 -1，并设置 errno 变量以指示错误类型

- 详解：

    - 使用 cfsetospeed() 函数时，通常需要先获取当前的串口配置参数，然后修改其中的输出波特率，最后将修改后的参数重新设置给串口

- 示例：

```c
#include <termios.h>
#include <stdio.h>

int main() {
    struct termios term;

    // 获取当前的串口配置参数
    tcgetattr(STDOUT_FILENO, &term);

    // 设置输出波特率为 9600
    if (cfsetospeed(&term, B9600) == -1) {
        perror("Failed to set output baud rate");
        return 1;
    }

    // 将修改后的配置参数重新设置给串口
    if (tcsetattr(STDOUT_FILENO, TCSANOW, &term) == -1) {
        perror("Failed to apply serial port settings");
        return 1;
    }

    printf("Output baud rate set to 9600\n");

    return 0;
}
```

- 注：
    - 在上述示例中，我们首先通过调用 tcgetattr() 函数获取当前的串口配置参数，并将其存储在 term 结构中。然后，我们使用 cfsetospeed() 函数将输出波特率设置为 9600。最后，我们调用 tcsetattr() 函数将修改后的配置参数重新应用到串口
    - 请注意，上述示例仅演示了设置输出波特率的情况。如果你还需要设置输入波特率，可以使用类似的方法调用 cfsetispeed() 函数进行设置。同时，还要注意进行错误处理，以检查函数调用是否成功

cfsetispeed

- 简介：
    - cfsetispeed() 函数用于设置串口设备的输入波特率。它是与串口相关的函数，用于配置串口的通信参数
- 原型：

```
int cfsetispeed(struct termios *termios_p, speed_t speed);
```

- 参数：
    - termios_p：指向 termios 结构的指针，该结构包含了串口的配置参数
    - speed：要设置的输入波特率
- 返回值：
    - 如果设置成功，返回值为 0
    - 如果设置失败，返回值为 -1，并设置 errno 变量以指示错误类型
- 详解：
    - 使用 cfsetispeed() 函数时，通常需要先获取当前的串口配置参数，然后修改其中的输入波特率，最后将修改后的参数重新设置给串口
- 示例：

```
#include <termios.h>
#include <stdio.h>

int main() {
    struct termios term;

    // 获取当前的串口配置参数
    tcgetattr(STDIN_FILENO, &term);

    // 设置输入波特率为 9600
    if (cfsetispeed(&term, B9600) == -1) {
        perror("Failed to set input baud rate");
        return 1;
    }

    // 将修改后的配置参数重新设置给串口
    if (tcsetattr(STDIN_FILENO, TCSANOW, &term) == -1) {
        perror("Failed to apply serial port settings");
        return 1;
    }

    printf("Input baud rate set to 9600\n");

    return 0;
}
```

- 注：
  - 在上述示例中，我们首先通过调用 tcgetattr() 函数获取当前的串口配置参数，并将其存储在 term 结构中。然后，我们使用 cfsetispeed() 函数将输入波特率设置为 9600。最后，我们调用 tcsetattr() 函数将修改后的配置参数重新应用到串口
  - 请注意，上述示例仅演示了设置输入波特率的情况。如果你还需要设置输出波特率，可以使用类似的方法调用 cfsetospeed() 函数进行设置。同时，还要注意进行错误处理，以检查函数调用是否成功

tcgetattr

- 简介：
  - tcgetattr() 是一个 C 函数，用于获取终端设备的属性。它用于读取终端设备的当前设置，并将这些设置存储在一个 termios 结构体中
- 原型：

```
int tcgetattr(int fd, struct termios *termios_p);
```

- 参数：
  - fd：要获取属性的终端设备文件描述符。
  - termios_p：指向 termios 结构体的指针，用于存储终端设备的属性
- 返回值：
  - tcgetattr() 函数会返回一个整数值，表示函数执行的结果。如果函数执行成功，则返回 0；否则，返回 -1，并设置相应的错误码
- 详解
  - 使用 tcgetattr() 函数可以获取终端设备的属性，如输入输出模式、字符大小、超时设置等。可以通过修改 termios 结构体的成员来更改终端设备的属性，然后使用 tcsetattr() 函数将修改后的属性应用到终端设备上
- 示例：

```
#include <stdio.h>
#include <termios.h>
#include <unistd.h>

int main() {
    int fd = STDIN_FILENO; // 标准输入设备文件描述符

    struct termios term;
    if (tcgetattr(fd, &term) == -1) {
        perror("tcgetattr");
```

```
        return 1;
    }

    // 打印终端设备的属性
    printf("Input flags: %x\n", term.c_iflag);
    printf("Output flags: %x\n", term.c_oflag);
    printf("Control flags: %x\n", term.c_cflag);
    printf("Local flags: %x\n", term.c_lflag);

    return 0;
}
```

- 注：
  - 这个示例程序获取标准输入设备的属性，并打印各个属性的值。

ioctl

- 简介：
  - ioctl() 是一个系统调用函数，用于控制设备的操作，它在不同的设备和操作系统上有不同的功能和用法。在 C 语言中，可以使用 ioctl() 函数来发送命令给设备驱动程序，并执行特定的操作
- 原型：

```
#include <sys/ioctl.h>

int ioctl(int fd, unsigned long request, ...);
```

- 参数：
  - fd：文件描述符，用于指定要进行控制的设备
  - request：控制命令，是一个无符号长整型数值，用于指定具体的操作
  - ...：可选参数，根据不同的控制命令可能需要提供额外的参数
- 返回值：
  - ioctl() 函数的返回值通常为 0 表示操作成功，-1 表示操作失败，并通过 errno 变量来获取具体的错误信息
- 详解：
  - 要使用 ioctl() 函数，需要事先了解设备的控制命令和参数的定义。不同的设备和操作系统可能有不同的命令和参数定义，因此需要参考相关的文档或设备驱动程序的头文件来获取正确的命令和参数定义
- 示例

```
#include <stdio.h>
#include <fcntl.h>
```

```c
#include <termios.h>
#include <unistd.h>
#include <sys/ioctl.h>

int main() {
    int fd = open("/dev/ttyS0", O_RDWR);
    if (fd == -1) {
        perror("Failed to open serial port");
        return 1;
    }

    struct termios options;
    if (tcgetattr(fd, &options) == -1) {
        perror("Failed to get serial port options");
        close(fd);
        return 1;
    }

    cfsetispeed(&options, B9600);
    cfsetospeed(&options, B9600);

    if (tcsetattr(fd, TCSANOW, &options) == -1) {
        perror("Failed to set serial port options");
        close(fd);
        return 1;
    }

    close(fd);
    return 0;
}
```

- 注：
  - 在上述示例中，首先使用 open() 函数打开串口设备文件 /dev/ttyS0，然后使用 tcgetattr() 函数获取当前的串口设置
  - 接下来使用 cfsetispeed() 和 cfsetospeed() 函数将输入和输出波特率设置为 9600。这里使用了波特率常量 B9600，具体的常量定义可以在 <termios.h> 头文件中找到
  - 最后使用 tcsetattr() 函数将修改后的串口设置应用到设备上，并关闭文件描述符
  - 这只是一个简单的示例，实际的 ioctl() 使用可能涉及更复杂的控制命令和参数，具体的用法和命令定义需要参考相关文档或设备驱动程序的说明

ftruncate

- 简介：
  - ftruncate() 函数是 C/C++ 中的一个文件操作函数，用于调整文件的大小
- 原型：

```c
#include <unistd.h>
```

```
int ftruncate(int fd, off_t length);
```

- 参数：
  - fd：文件描述符，指定要操作的文件。
  - length：要设置的新文件大小
- 详解：
  - 函数功能：
    - 如果文件当前的大小大于指定的 length，则文件将被截断到指定的大小。
    - 如果文件当前的大小小于指定的 length，则文件将被扩展到指定的大小，并使用空字节填充扩展的部分。
    - 如果文件当前的大小等于指定的 length，则文件大小保持不变
  - 返回值
    - 如果调用成功，返回值为 0。
    - 如果调用失败，返回值为 -1，并设置相应的错误码，可以通过 errno 全局变量获取具体的错误信息。
  - 使用 ftruncate() 函数需要先打开文件，获取文件描述符 fd，然后调用该函数即可。请确保在使用该函数时具有足够的权限来修改文件大小
- 示例

```
#include <fcntl.h>
#include <unistd.h>
#include <iostream>

int main() {
    const char* filename = "example.txt";
    int fd = open(filename, O_RDWR | O_CREAT, 0644);
    if (fd == -1) {
        std::cerr << "Failed to open file: " << filename << std::endl;
        return 1;
    }

    off_t new_size = 1024;  // New file size in bytes

    if (ftruncate(fd, new_size) == -1) {
        std::cerr << "Failed to truncate file" << std::endl;
        close(fd);
        return 1;
    }

    close(fd);

    std::cout << "File size adjusted successfully" << std::endl;

    return 0;
}
```

- 注：
  - 上述代码打开名为 "example.txt" 的文件，将其大小调整为 1024 字节。如果调用成功，将输出 "File size adjusted successfully"。否则，将输出相应的错误信息
  - 请注意，使用 ftruncate() 函数时应格外谨慎，确保操作的文件和大小设置是正确的，以免意外丢失数据

shm_unlink

- 简介：
  - shm_unlink() 函数用于解除与共享内存对象的链接，并将其标记为待删除状态。具体来说，shm_unlink() 函数的作用是删除共享内存对象的名字，但并不立即释放共享内存资源，只有当所有打开该共享内存对象的文件描述符都关闭时，才会释放共享内存资源
- 原型

```
int shm_unlink(const char* name);
```

- 参数：
  - 参数 name 是共享内存对象的名字，通过该名字可以标识和访问共享内存对象。
- 详解：
  - 调用 shm_unlink() 函数后，如果成功解除链接并标记为删除，函数返回 0；如果发生错误，返回 -1，并设置对应的错误码
- 示例：

```
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>

int main() {
    const char* shm_name = "/my_shared_memory";

    // 打开共享内存对象
    int shm_fd = shm_open(shm_name, O_RDWR, 0666);
    if (shm_fd == -1) {
        // 错误处理
        return 1;
    }

    // 解除链接并标记为删除
    int result = shm_unlink(shm_name);
    if (result == -1) {
        // 错误处理
        return 1;
    }
```

```
    // 关闭共享内存对象
    close(shm_fd);

    return 0;
}
```

- 注：
  - 在上述示例中，我们通过 shm_open() 打开了一个共享内存对象，然后调用 shm_unlink() 解除与该对象的链接并标记为删除。最后，通过 close() 关闭了共享内存对象的文件描述符
  - 需要注意的是，只有当所有打开共享内存对象的进程都解除链接后，该共享内存对象才会被真正删除。因此，在使用 shm_unlink() 函数时需要确保所有进程都已解除链接，以避免资源泄漏

shm_open

- 简介：
  - 在 C++ 中，shm_open() 函数用于创建或打开共享内存对象。它是 POSIX 共享内存机制的一部分，用于在不同进程之间共享内存数据。
- 原型：

```
int shm_open(const char* name, int oflag, mode_t mode);
```

- 参数：
  - name：共享内存对象的名称，以字符串形式表示。在创建共享内存对象时，可以给它一个唯一的名称，以便其他进程可以使用相同的名称来访问该对象
  - oflag：打开标志，用于指定打开共享内存的方式。常用的选项包括 O_CREAT（如果共享内存对象不存在，则创建它）和 O_RDWR（可读写方式打开共享内存）等
  - mode：权限模式，用于设置共享内存对象的权限。只有在创建新的共享内存对象时才会使用该参数，它类似于 open() 函数中的文件权限参数
- 详解：
  - shm_open() 函数返回一个整数值，代表共享内存对象的文件描述符。如果打开或创建共享内存对象成功，则返回非负整数值；否则返回 -1，并设置相应的错误码。
- 示例：

```
#include <iostream>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>

int main() {
    // 共享内存对象的名称
    const char* name = "/my_shared_memory";
```

```
    // 创建或打开共享内存对象
    int fd = shm_open(name, O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
    if (fd == -1) {
        std::cout << "Failed to create/open shared memory" << std::endl;
        return 1;
    }

    // 关闭共享内存对象的文件描述符
    close(fd);

    std::cout << "Shared memory created/opened successfully" << std::endl;

    return 0;
}
```

- 注：
  - 在上述示例中，我们使用 shm_open() 函数创建或打开了一个名为 "/my_shared_memory" 的共享内存对象。通过指定 O_CREAT 和 O_RDWR 标志，我们创建了一个新的共享内存对象，并以可读写方式打开它。然后，我们关闭了共享内存对象的文件描述符，并输出一条成功的消息
  - 请注意，shm_open() 函数只是打开或创建共享内存对象，它并没有定义共享内存的大小或分配实际的内存空间。要分配内存空间并映射到共享内存对象，可以使用 mmap() 函数。同时，创建或打开共享内存对象后，应该适当处理错误情况并进行错误检查，以确保操作的成功
  - 在编译链接时，需要包含头文件`<sys/mman.h>`和链接库`librt.so`

shmat

- 简介：
  - shmat()函数用于将共享内存段连接到当前进程的地址空间。
- 原型：

```
#include <sys/types.h>
#include <sys/shm.h>

void *shmat(int shmid, const void *shmaddr, int shmflg);
```

- 参数：
  - shmid：共享内存标识符，即共享内存段的标识符。
  - shmaddr：指定共享内存段连接到当前进程地址空间的地址。通常设置为NULL，让系统自动选择一个可用的地址
  - shmflg：标志位，用于指定共享内存段的连接方式和权限。可以使用SHM_RDONLY表示只读方式连接共享内存段，也可以使用0表示可读写方式连接
- 详解：

    - shmat()函数的返回值是一个void*指针，指向共享内存段在当前进程地址空间的起始地址。连接成功时，返回指向共享内存段的指针；连接失败时，返回-1。

- 示例：

```
#include <stdio.h>
#include <sys/shm.h>

int main() {
    int shmid = 1234; // 共享内存标识符

    // 连接共享内存段
    void *shm_ptr = shmat(shmid, NULL, 0);
    if (shm_ptr == (void*)-1) {
        perror("shmat");
        return 1;
    }

    printf("Shared memory segment attached at address: %p\n", shm_ptr);

    // 在连接的共享内存段中读写数据
    int *data = (int*)shm_ptr;
    *data = 42;
    printf("Data in shared memory: %d\n", *data);

    // 断开共享内存段的连接
    int result = shmdt(shm_ptr);
    if (result == -1) {
        perror("shmdt");
        return 1;
    }

    printf("Shared memory segment detached\n");

    return 0;
}
```

- 注：
    - 在这个示例中，我们首先使用shmat()函数将共享内存段连接到当前进程的地址空间。通过将shmaddr参数设置为NULL，让系统自动选择可用的地址进行连接。如果连接成功，返回的shm_ptr指向共享内存段在当前进程地址空间的起始地址
    - 接下来，我们可以在连接的共享内存段中进行读写操作。在示例中，我们将共享内存段当作一个整数变量来使用，将值42写入共享内存段，并从共享内存段中读取该值
    - 最后，我们使用shmdt()函数断开共享内存段的连接，将共享内存段从当前进程的地址空间中分离。注意，断开连接并不会删除共享内存段，只是将共享内存段从当前进程中移除。如果其他进程仍然连接到该共享内存段，它们仍然可以访问和修改共享内存数据
    - 需要注意的是，使用shmat()和shmdt()函数时需要确保当前用户对共享内存段具有足够的权限。另外，要小心处理共享内存的并发访问，使用同步机制来避免竞争条件和数据不一致的问题

shmctl

- 简介：
    - shmctl()函数用于控制和操作共享内存段的属性。
- 原型：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- 参数：
    - shmid：共享内存标识符，即共享内存段的标识符。
    - cmd：控制命令，用于指定对共享内存段的操作，可以是下列值之一：
        - IPC_STAT：获取共享内存段的状态信息，将共享内存段的信息存储在buf指向的shmid_ds结构体中。
        - IPC_SET：设置共享内存段的状态信息，将buf指向的shmid_ds结构体中的信息应用于共享内存段。
        - IPC_RMID：删除共享内存段，释放共享内存资源。
    - buf：用于存储共享内存段状态信息的结构体指针。
- 详解：
    - 使用IPC_STAT命令可以获取共享内存段的状态信息，包括共享内存段的大小、创建者的用户ID、最后一次操作的时间等。这些信息被存储在shmid_ds结构体中，通过传递一个指向该结构体的指针作为buf参数来获取这些信息
    - 使用IPC_SET命令可以修改共享内存段的属性，比如更改共享内存段的权限、所有者等。需要将一个填充了新属性值的shmid_ds结构体指针传递给buf参数
    - 使用IPC_RMID命令可以删除共享内存段，释放相关的资源。注意，删除共享内存段并不会立即清除共享内存中的数据，只是使得该共享内存段不再可用。实际的共享内存段会在所有连接到它的进程都断开连接后被系统清理
- 示例

```
#include <stdio.h>
#include <sys/shm.h>

int main() {
    int shmid = 1234; // 共享内存标识符

    struct shmid_ds shm_info;

    // 获取共享内存段的状态信息
    int result = shmctl(shmid, IPC_STAT, &shm_info);
    if (result == -1) {
        perror("shmctl");
        return 1;
```

```
    }

    printf("Size of shared memory segment: %lu\n", shm_info.shm_segsz);
    printf("Owner's user ID: %d\n", shm_info.shm_perm.uid);
    printf("Last attach time: %ld\n", shm_info.shm_atime);

    // 修改共享内存段的权限
    shm_info.shm_perm.mode = 0666; // 设置权限为rw-rw-rw-
    result = shmctl(shmid, IPC_SET, &shm_info);
    if (result == -1) {
        perror("shmctl");
        return 1;
    }

    printf("Shared memory segment permissions updated successfully\n");

    return 0;
}
```

- 注：
  - 在这个示例中，我们首先使用shmctl()函数和IPC_STAT命令获取共享内存段的状态信息，并将信息存储在shmid_ds结构体（shm_info）中。然后，我们打印出共享内存段的大小、所有者的用户ID以及最后一次附加到共享内存段的时间
  - 接下来，我们修改共享内存段的权限，将权限设置为0666，即读写权限对所有用户都可用。我们将新的权限信息存储在shm_info结构体中，并使用shmctl()函数和IPC_SET命令应用这些新的属性
  - 请注意，使用shmctl()函数进行修改操作时，需要确保当前用户对共享内存段具有足够的权限。另外，要小心使用IPC_RMID命令删除共享内存段，因为这将导致所有连接到该共享内存段的进程失去对它的访问权限

shmdt

- 简介：
  - shmdt()函数用于将共享内存段从当前进程的地址空间中分离，即将共享内存段与进程的连接断开
- 原型：

```
#include <sys/types.h>
#include <sys/shm.h>

int shmdt(const void *shmaddr);
```

- 参数
  - shmaddr：共享内存段的起始地址，即附加到进程地址空间的地址。
- 详解：

    - shmdt()函数将共享内存段从当前进程的地址空间中分离，并释放与共享内存段相关的资源。一旦共享内存段被分离，进程将无法再访问共享内存段中的数据。
    - 使用shmdt()函数时需要注意以下几点：
        - shmaddr参数应该是之前使用shmat()函数将共享内存段附加到进程地址空间时返回的地址
        - 成功执行shmdt()函数后，共享内存段与进程的连接被断开，但共享内存段本身并没有被销毁。其他进程仍然可以访问该共享内存段。
        - 如果多个进程共享同一个共享内存段，在一个进程中调用shmdt()函数分离共享内存段后，其他进程仍然可以继续访问该共享内存段。
        - 分离共享内存段后，可以使用shmctl()函数删除共享内存段。

- 示例：

```c
#include <stdio.h>
#include <sys/shm.h>

int main() {
    int shmid = 1234; // 共享内存标识符

    void *shmaddr = shmat(shmid, NULL, 0);
    if (shmaddr == (void *)-1) {
        perror("shmat");
        return 1;
    }

    printf("Shared memory segment attached at address: %p\n", shmaddr);

    int result = shmdt(shmaddr);
    if (result == -1) {
        perror("shmdt");
        return 1;
    }

    printf("Shared memory segment detached successfully\n");

    return 0;
}
```

- 注：
    - 在这个示例中，我们首先使用shmat()函数将共享内存段附加到进程地址空间，并获取共享内存段的起始地址（shmaddr）。然后，我们使用shmdt()函数分离共享内存段，断开进程与共享内存段的连接。如果分离共享内存段时出现错误，我们使用perror()函数打印相应的错误消息
    - 请注意，分离共享内存段后，进程将无法再访问共享内存段中的数据

shmget

- 简介：
    - shmget()函数是一个系统调用，用于创建或打开一个共享内存段。
- 原型：

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);
```

- 参数
    - key：共享内存段的键值，通常由ftok()函数生成。
    - size：共享内存段的大小，以字节为单位
    - shmflg：共享内存段的访问权限和标志，可以是一个或多个标志的位或运算结果，如IPC_CREAT、IPC_EXCL等。
- 详解
    - shmget()函数根据给定的键值和大小创建或打开一个共享内存段，并返回一个与共享内存段相关的标识符（shmid）。这个标识符可以用于后续的共享内存操作，如附加、分离、读写等。
    - 使用shmget()函数时需要注意以下几点：
        - key参数应该是一个与其他进程共享的键值，通常由ftok()函数生成。不同的key值将对应不同的共享内存段。
        - size参数指定了共享内存段的大小，即申请的内存空间大小。
        - shmflg参数用于设置共享内存段的权限和标志。常见的标志有IPC_CREAT表示创建新的共享内存段，IPC_EXCL表示只在共享内存段不存在时创建，0666表示权限设置为可读可写。
        - 如果成功创建或打开共享内存段，shmget()函数返回一个非负整数的共享内存标识符（shmid），用于后续的共享内存操作。
        - 如果共享内存段已经存在且没有指定IPC_EXCL标志，则返回已存在的共享内存标识符
- 示例

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main() {
    key_t key = ftok("/path/to/file", 1);
    if (key == -1) {
        perror("ftok");
        return 1;
    }

    int size = 1024; // 共享内存段的大小，以字节为单位

    int shmid = shmget(key, size, IPC_CREAT | 0666);
    if (shmid == -1) {
        perror("shmget");
        return 1;
    }

    printf("Shared memory segment created with shmid: %d\n", shmid);
```

```
    return 0;
}
```

- 注：
  - 这个示例中，我们首先使用ftok()函数生成一个键值，然后使用shmget()函数创建一个共享内存段，并获取与该共享内存段相关的标识符（shmid）。如果创建共享内存段时出现错误，我们使用perror()函数打印相应的错误消息。
  - 请注意，创建共享内存段后，还需要使用其他函数（如shmat()）将共享内存段附加到进程的地址空间中，才能进行读写操作。此处仅演示了创建共享内存段的基本过程

ftok

- 简介：
  - `ftok()`函数是C语言中的一个函数，用于生成一个唯一的键值（key）来创建一个共享内存、消息队列或信号量等系统对象。
- 原型：

```
key_t ftok(const char *pathname, int proj_id);
```

- 参数
  - pathname：一个存在的文件的路径名，用于生成键值。
  - proj_id：项目标识符，通常为一个非负整数，用于区分不同的键值。
- 详解
  - ftok()函数根据pathname和proj_id生成一个唯一的键值，并返回一个key_t类型的值作为结果。这个键值可以用于创建和访问共享内存、消息队列或信号量等系统对象
  - 使用ftok()函数时需要注意以下几点：
    - pathname参数必须是一个有效的文件路径，且文件必须存在。如果文件不存在或路径无效，ftok()函数将返回一个错误的键值。
    - proj_id参数通常是一个非负整数，用于区分不同的键值。不同的proj_id值将生成不同的键值。
- 示例

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>

int main() {
    key_t key = ftok("/path/to/file", 1);

    if (key == -1) {
        perror("ftok");
        return 1;
```

```
    }

    printf("Generated key: %d\n", key);

    return 0;
}
```

- 注：
  - 请注意，生成的键值应该在进程之间共享，以便其他进程可以使用相同的键值访问同一个系统对象。通常，可以将生成的键值传递给共享内存、消息队列或信号量等相关函数来创建或访问这些对象
  - 相同的文件，相同的标识符，生成相同的键值

newlocale

- `newlocale`是一个C语言函数，用于创建一个新的区域设置对象。区域设置对象是一个数据结构，包含了与特定地区相关的本地化信息，如日期格式、货币符号、语言偏好等。
- `newlocale`函数的原型如下：

```
#include <locale.h>

locale_t newlocale(int category_mask, const char *locale, locale_t base);
```

- 参数解释如下：
  - `category_mask`：一个位掩码，用于指定要设置的区域设置的不同方面。可以使用`LC_ALL`表示所有方面，或使用以下值之一：`LC_COLLATE`、`LC_CTYPE`、`LC_MONETARY`、`LC_NUMERIC`、`LC_TIME`。
  - `locale`：一个字符串，表示要创建的区域设置的名称。可以是特定地区的名称（例如："en_US"表示美国英语），也可以是一些特殊值（例如："C"表示默认的"POSIX"区域设置）。
  - `base`：一个基本区域设置对象，用作新区域设置的基础。可以将其设置为`NULL`，表示没有基础区域设置。
- 函数返回一个`locale_t`类型的值，表示新创建的区域设置对象。如果创建失败，函数返回`NULL`。
- 使用`newlocale`函数可以创建新的区域设置对象，以便在程序中进行本地化设置，从而使程序能够适应不同的地区和语言环境。创建后的区域设置对象可以传递给其他一些与本地化相关的函数，如`strftime`、`strcoll`等，以便根据特定区域的规则进行相应的操作。

memcpy

- 简介：
  - C 库函数 void *memcpy(void *str1, const void *str2, size_t n) 从存储区 str2 复制 n 个字节到存储区 str1。

- 声明
  - `void *memcpy(void *str1, const void *str2, size_t n);`
- 参数：
  - str1 -- 指向用于存储复制内容的目标数组，类型强制转换为 void* 指针。
  - str2 -- 指向要复制的数据源，类型强制转换为 void* 指针。
  - n -- 要被复制的字节数。
- 返回值：
  - 该函数返回一个指向目标存储区 str1 的指针。

getopt

- 简介：getopt() 方法是用来分析命令行参数的，该方法由 Unix 标准库提供，包含在 `<unistd.h>` 头文件中。
- 定义：

```
int getopt(int argc, char * const argv[], const char *optstring);

extern char *optarg;
extern int optind, opterr, optopt;
```

- getopt参数：
  - argc：通常由 main 函数直接传入，表示参数的数量
  - argv：通常也由 main 函数直接传入，表示参数的字符串变量数组
  - optstring：一个包含正确的参数选项字符串，用于参数的解析。例如 “abc:”，其中 -a，-b 就表示两个普通选项，-c 表示一个必须有参数的选项，因为它后面有一个冒号
    - 可选参数：一个冒号表示选项后必须有参数，没有参数就会报错。如果有两个冒号的话，那么这个参数就是可选参数了，即可有可没有。
    - 注意这里 可选参数 选项 -c 后面跟参数的时候，一定不能有空格。
    - 但是如果是 必选参数，即选项后面只有一个冒号，则是有没有空格都可以。
- 外部变量说明
  - optarg：如果某个选项有参数，这包含当前选项的参数字符串
  - optind：argv 的当前索引值
  - opterr：正常运行状态下为 0。非零时表示存在无效选项或者缺少选项参数，并输出其错误信息
  - optopt：当发现无效选项字符时，即 getopt() 方法返回 ? 字符，optopt 中包含的就是发现的无效选项字符
- 输入字符串转 int
  - 由于 optarg 都是字符串类型的，所以当我们想要整型的输入参数时，会经常用到 atio() 这个方法，这里也简单介绍一下。

- atoi (表示 ascii to integer) 是把字符串转换成整型数的一个函数，包含在 `<stdlib.h>` 头文件中，使用方式如下：
  - `int num = atoi(optarg);`

getopt_long

- 简介：getopt_long支持处理长短选项的命令行解析，函数在`<getopt.h>`头文件中
- 定义：`int getopt_long(int argc, char *const *argv, const char *shortopts, const option *longopts, int *longind);`
- 参数
  - argc、argv 和 main 函数的两个参数一致；
  - optstring：短选项字符串；
    - 形式如 a🅱:cd，分别表示程序支持的命令行短选项有 -a、-b、-c、-d，冒号含义如下：
      - 只有一个字符，不带冒号-----只表示选项，如: -c
      - 一个字符，后面接一个冒号-----表示选项后面带一个参数，如: -a 100
      - 一个字符，后面接两个冒号-----表示选项后面带一个可选参数，即参数可有可无，如果带参数，则选项与参数之间不能有空格，如: -b200
  - longopts：表示长选项结构体：

```
  struct option
  {
    const char *name;
    /* has_arg can't be an enum because some compilers complain
about
       type mismatches in all the code that assumes it is an int.
*/
    int has_arg;
    int *flag;
    int val;
  };

  static struct option long_options[] = {
      {"help", no_argument,       0,   'h'},
      {"cfg",  required_argument, 0,   'f'},
      {"http-proxy", required_argument, &lopt, 1 },
      {"http-user", required_argument, &lopt, 2 },
      {"http-passwd", required_argument, &lopt, 3 },
      {"http-proxy-user", required_argument, &lopt, 4 },
      {"http-proxy-passwd", required_argument, &lopt, 5 },
      {"http-auth-scheme", required_argument, &lopt, 6 },
      {nullptr, 0, nullptr, 0}
  };
```

    - name：表示选项的名称，如 help、cfg 等；
    - has_arg：表示选项后面是否携带参数，该参数有三个不同值，如：
      - no_argument(0): 参数后面不跟参数值，如：–help；

        - required_argument(1): 参数输入格式为：–参数 值 or –参数=值，
          如：./dist_measurment_node --cfg ./cal_result；
        - optional_argument(2): 参数输入格式只能为：–参数=值。
      - flag：该参数有两种，空或者非空：
        - 如果参数为空(NULL)：当选中某个长选项的时候，getopt_long 将返回 val 值，如：./可执行程序 --help，getopt_long 的返回值为 h。
        - 如果参数不为空：当选中某个长选项的时候，getopt_long 将返回 0，并且将 flag 指针指向 val 值，如：./可执行程序 --http-proxy=127.0.0.1:80，getopt_long 的返回值为 0，并且 lopt 值为 1。
      - val：表示指定函数找到该选项时的返回值，或者当 flag 非空时指定 flag 指向的数据的值。
      - longindex：longindex 非空，它指向的变量将记录当前找到参数符合 longopts 里的第几个元素的描述，即 longopts 的下标值。

- 全局变量

  - optarg：会被系统自动赋为当前选项的参数；
  - optind：表示下一个将被处理到的参数在 argv 中的下标值；
  - opterr：如果 opterr = 0，在getopt、getopt_long、getopt_long_only 遇到错误将不会输出错误信息到标准输出流。opterr 在非 0 时，向屏幕输出错误。
  - optopt：会被自动赋值当前未标识的选项；

- 返回值：

  - 如果短选项找到，那么将返回短选项对应的字符。
  - 如果长选项找到，如果flag为NULL，返回val。如果flag不为空，返回0
  - 如果遇到一个选项没有在短字符、长字符里面，或者在长字符里面存在二义性的，返回“？”
  - 如果解析完所有字符没有找到（一般是输入命令参数格式错误，eg： 连斜杠都没有加的选项），返回“-1”
  - 如果选项需要参数，忘了添加参数。返回值取决于optstring，如果其第一个字符是“：”，则返回“：”，否则返回“？”

- 注意：

  - longopts的最后一个元素必须是全0填充，否则会报段错误
  - 短选项中每个选项都是唯一的。而长选项如果简写，也需要保持唯一性。

perror()

- 功能：把一个描述性错误消息输出到标准错误 stderr
- 原型：`void perror(const char *str);`
- 参数：
  - `str` -- 这是 C 字符串，包含了一个自定义消息，将显示在原本的错误消息之前。

daemon()

- 功能：将程序置于后台，并与控制终端分离
- 原型：`int daemon(int nochdir, int noclose);`
- 参数：
  - `nochdir` -- 如果为0，执行`chdir ("/")`

    - `noclose` -- 如果为0，将`stdin, stdout, stderr`重定向到`/dev/null`

strchr()

- 功能：在参数 `str` 所指向的字符串中搜索最后一次出现字符 `c`（一个无符号字符）的位置
- 原型：`char* strrchr(const char *str, int c);`
- 参数：
    - `str` -- C字符串
    - `c` -- 要搜索的字符。以 `int` 形式传递，但是最终会转换回 `char` 形式
- 返回值：
    - 成功 -- 返回`str`中最后一次出现字符`c`的位置
    - 失败 -- 没有找到该值，返回一个空指针
- 需求：
    - 头文件：`#include <string.h>`

pipe()

- 功能：创建单项通信通道(管道)
- 原型：`int pipe(int *__pipedes);`
    - 如果成功，两个文件描述符存储在`pipedes`中
    - `pipedes[1]`上写入的字节可以从`pipedes[0]`中读取
- 参数：
    - `__pipedes` -- 数组
- 返回值：
    - 成功 -- 0
    - 失败 -- -1
- 需求：
    - 头文件：`#include <unistd.h>`
- 注意：
    - 管道没有外部或永久的名字，因此程序只能通过它的两个描述符来访问它
    - 因此，管道只能由创建它的进程或在`fork`时继承了它的描述符的后代进程使用。

mkfifo()

- 功能：创建新的FIFO特殊文件
- 原型：`int mkfifo(const char *path, mode_t mode);`
- 参数：
    - `path` -- 创建的文件路径
    - `mode` -- 文件权限
- 返回值：
    - 成功 -- 0
    - 失败 -- -1，并设置errno

fork()

- 功能：克隆调用进程，创建一个精确的副本。
- 原型：`pid_t fork();`
- 参数：无

- 返回值：
  - 成功 -- 新进程返回 0，或者新进程对旧进程的进程ID
  - 失败 -- -1

getpid()

- 简介：
  - getpid() 是一个 C 标准库函数，用于获取当前进程的进程ID（Process ID）
- 原型：

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);
```

- 返回值：
  - 返回一个整数值，表示当前进程的进程ID
- 详解：
  - getpid() 函数用于获取当前进程的进程ID，进程ID是一个唯一标识符，用于在系统中唯一地标识一个进程。
  - 该函数属于 POSIX 标准，并可在大多数操作系统上使用，如 Linux、Unix 和 macOS。
  - 要使用 getpid() 函数，需要包含头文件 <sys/types.h> 和 <unistd.h>。
  - 通过调用 getpid() 函数，可以在程序中获取当前进程的进程ID，并在需要时进行处理或记录
  - 需要注意的是，进程ID是由操作系统分配和管理的，它在进程的生命周期中保持不变。每个进程都有一个唯一的进程ID，可以用于识别和跟踪进程。在多进程环境中，使用 getpid() 函数可以帮助确定不同进程之间的执行路径和行为
- 示例：

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
    pid_t pid = getpid();
    printf("Process ID: %d\n", pid);
    return 0;
}
```

getppid()

- 功能：返回父进程ID
- 原型：`pid_t getppid(void);`

- 参数：无
- 返回值：
  - 成功 -- 父进程ID
  - 失败 --
- 需求：
  - 头文件：`#include <unistd.h>`
- 注意：
  - `pid_t`是用来表示进程ID的一个无符号整数类型
  - `getppid`函数不能返回错误

geteuid()

- 功能：返回用户ID
- 原型：`uid_t geteuid(void);`
- 参数：无
- 返回值：
  - 成功 -- 用户ID
  - 失败 --
- 需求：
  - 头文件：`#include <unistd.h>`
- 注意：
  - `uid_t`是代表用户ID的整数类型
  - `geteuid()`不返回错误

getegid()

- 功能：返回组ID
- 原型：`gid_t getegid(void);`
- 参数：无
- 返回值：
  - 成功 -- 组ID
  - 失败 --
- 需求：
  - 头文件：`#include <unistd.h>`
- 注意：
  - `gid_t`是代表组ID的整数类型
  - `getegid()`不返回错误

fork()

- 功能：创建新的进程
- 原型：`pid_t fork(void);`
- 参数：无
- 返回值：
  - 成功 -- 向子进程返回0，并将子进程ID返回父进程
  - 失败 -- -1，并设置errno，没有创建子进程
- 需求：
  - 头文件：`#include <unistd.h>`

- 注意：
  - fork函数的返回值是允许父进程和子进程区别自己并执行不同代码的关键特征

execl()

- 功能：使用参数`arg`以及之后的参数 执行`path`
- 原型：`int execl(const char* path, const char* arg,...);`
- 参数：
  - `path` -- 要执行的文件
  - `arg` -- 执行的参数
- 返回值：
  - 成功 --
  - 失败 --

pthread_sigmask()

- 功能：修改，控制调用线程的信号掩码
- 原型：`int pthread_sigmask(int how, const sigset_t *newmask, sigset_t *oldmask);`
- 参数：
  - `how` --
    - `SIG_BLOCK` ：结果集是当前集合参数集的并集，
    - `SIG_UNBLOCK`：结果集是当前集合参数集的差集，
    - `SIG_SETMASK`：结果集是由参数集指向的集
  - `newmask`
  - `oldmask`
- 返回值：
  - 成功 -- 0
  - 失败 -- -1，并设置errno
- 需求：
  - 头文件：`#include <signal.h>`

pthread_create()

- 功能：创建一个新的线程，并让它可执行
- 原型：`int pthread_create(pthread_t *newthread, const pthread_attr_t attr, void *(*start_routine)(void *), void *arg);`
- 参数：
  - `newthread` -- 存储新线程的句柄，指向线程标识符指针
  - `attr` -- 设置线程属性
  - `start_routine` -- 线程运行函数的起始地址
  - `arg` -- 运行函数的参数
- 返回值：
  - 线程创建成功 -- 0
  - 线程创建失败 -- 返回出错编号，并且*thread中的内容是未定义的
- 需求：
  - 头文件：`#include <pthread>`
- 注意：

    - 返回成功时，由`newthread`指向的内存单元被设置为新创建线程的线程ID
    - attr参数用于指定各种不同的线程属性
    - 新创建的线程从`start_routine`函数的地址开始运行，该函数只有一个万能指针参数`arg`
    - 如果需要向`start_routine`函数传递的参数不止一个，那么需要把这些参数放到一个结构中，然后把这个结构的地址作为arg的参数传入
    - linux下用C语言开发多线程程序，Linux系统下的多线程遵循POSIX线程接口，称为pthread

pthread_join()

- 功能：子线程合入主线程，主线程阻塞等待子线程结束，然后回收子线程资源。
- 原型：`int pthread_join(pthread_t thread, void **retval);`
- 参数：
    - `thread` -- 线程标识符，即线程ID，标识唯一的线程
    - `retval` -- 用户定义的指针，用来存储被等待线程的返回值
- 返回值：
    - 成功 -- 0
    - 失败 -- 其他值
- 需求：
    - 头文件：`#include <pthread>`
- 注意：
    - pthread_join()函数，以阻塞的方式等待thread指定的线程结束。
    - 当函数返回时，被等待线程的资源被收回。
    - 如果线程已经结束，那么该函数会立即返回。并且thread指定的线程必须是joinable的。

pthread_detach()

- 功能：指示线程`th`永远不会与`PTHREAD_JOIN`连接。因此，`th`的资源将在它终止时立即被释放，而不是等待另一个线程对它执行`PTHREAD_JOIN`。
    - 主线程与子线程分离，子线程结束后，资源自动回收。
- 原型：`int pthread_detach(pthread_t th);`
- 参数：
    - `th` -- 线程标识符，指向需要自动释放的线程
- 返回值：
    - 成功 -- 0
    - 失败 -- 其他情况
- 需求：
    - 头文件：`#include <pthread>`
- 注意：

pthread 两种状态

- `pthread`有两种状态`joinable`状态和`unjoinable`状态
- 如果线程是`joinable`状态，当线程函数自己返回退出时或`pthread_exit`时都不会释放线程所占用堆栈和线程描述符（总计8K多）
- 只有当你调用了`pthread_join`之后这些资源才会被释放
- 若是`unjoinable`状态的线程，这些资源在线程函数退出时或`pthread_exit`时自动会被释放

- `unjoinable`属性可以在`pthread_create`时指定，或在线程创建后在线程中`pthread_detach`自己
- `pthread_detach(pthread_self());`
    - 将状态改为`unjoinable`状态，确保资源的释放。或者将线程置为 `joinable`,然后适时调用 `pthread_join`.
    - 简单的说就是在线程函数头加上 `pthread_detach(pthread_self())`的话，线程状态改变，**在函数尾部直接 `pthread_exit` 线程就会自动退出**。省去了给线程擦屁股的麻烦

_exit()

- 功能：使用`__status`的`low-order 8 bits`终止程序执行
- 原型：`void _exit(int __status);`
- 参数：
    - `__status` --
- 返回值：

waitpid()

- 功能：等待进程为`pid`的子进程死亡，
- 原型：`pid_t waitpid(pid_t pid, int* stat_loc, int options);`
- 参数：
    - `pid` -- 等待的进程,指向返回状态所在单元的指针和一个用来指定可选项的标志符
        - pid有四种情况：
            - pid < -1 等待组ID等于pid绝对值的任意子进程
            - pid == -1 等待任意子进程
            - pid == 0 等待组ID等于调用进程组ID的任意子进程
            - pid > 0 等待进程ID与pid相等的子进程
    - `stat_loc` -- 指向终止进程的终止状态，如果不关心终止状态可指定为空指针
    - `options` -- 控制waitpid的操作
        - WCONTINUED
        - WUNTRACED
        - WNOHANG waitpid不阻塞
- 返回值：
    - 成功 -- 返回pid，并将子进程的状态存储到`stat_loc`中
    - 失败 -- -1

mkstemp()

- 功能：根据`__tmplate`生成唯一的临时文件名
- 原型：`int mkstemp(char *__template);`
- 参数：
    - `__tmplate` -- 需要操作的文件
- 返回值：
    - 成功 -- 返回在文件上打开的用于读取和写入的文件描述符
    - 失败 -- -1 （如果它不能创建一个唯一的文件名）

openlog()

- 功能：此函数用来打开一个到系统日志记录程序的连接，打开之后就可以用`syslog`或`vsyslog`函数向系统日志里添加信息了。而`closelog`函数就是用来关闭此连接的
- 原型：`void openlog(const char *ident, int option, int facility);`
- 参数：
  - `ident` -- 一个标记，`ident`所表示的字符串将固定地加在每行日志的前面以标识这个日志，通常就写成当前程序的名称以作标记
  - `option` -- 参数option是下列值取与运算的结果：
    - `LOG_CONS`，`LOG_NDELAY`， `LOG_NOWAIT`， `LOG_ODELAY`， `LOG_PERROR`，`LOG_PID`
    - 各值意义请参考`man openlog`手册
  - `facility` -- 指明记录日志的程序的类型
- 返回值：空

setlogmask()

- 功能：设置日志掩码级别
- 原型：`int setlogmask(int mask);`
- 参数：
  - `mask` --
- 返回值：
  - 成功 --
  - 失败 --

C 标准库 `<stdarg.h>`

- `stdarg.h` 头文件定义了一个变量类型 `va_list` 和三个宏，这三个宏可用于在参数个数未知（即参数个数可变）时获取函数中的参数。
- 可变参数的函数通在参数列表的末尾是使用省略号`(,...)`定义的

库变量 -- va_list

- 功能：这是一个适用于 `va_start()`、`va_arg()` 和 `va_end()` 这三个宏存储信息的类型
- 声明：`typedef _G_va_list va_list;`

va_start()

- 功能：
  - C 库宏 `void va_start(va_list ap, last_arg)` 初始化 `ap` 变量，它与 `va_arg` 和 `va_end` 宏是一起使用的
  - `last_arg` 是最后一个传递给函数的已知的固定参数，即**省略号之前的参数**
  - 这个宏必须在使用 `va_arg` 和 `va_end` 之前被调用
- 原型：`void va_start(va_list ap, last_arg);`
- 参数：
  - `ap` -- 这是一个 `va_list` 类型的对象，它用来存储通过 `va_arg` 获取额外参数时所必需的信息
  - `last_arg` -- 最后一个传递给函数的已知的固定参数
- 返回值：空

va_arg()

- 功能：

    - C 库宏 `type va_arg(va_list ap, type)` 检索函数参数列表中类型为 `type` 的下一个参数。
    - 它无法判断检索到的参数是否是传给函数的最后一个参数
- 原型：`type va_arg(va_list ap, type);`
- 参数：
    - `ap` -- 这是一个 `va_list` 类型的对象，存储了有关额外参数和检索状态的信息。该对象应在第一次调用 `va_arg` 之前通过调用 `va_start` 进行初始化
    - `type` -- 这是一个类型名称。该类型名称是作为扩展自该宏的表达式的类型来使用的。
- 返回值：
    - 该宏返回下一个额外的参数，是一个类型为 `type` 的表达式

va_end()

- 功能：
    - C 库宏 `void va_end(va_list ap)` 允许使用了 `va_start` 宏的带有可变参数的函数返回。
    - 如果在从函数返回之前没有调用 `va_end`，则结果为未定义
- 原型：`void va_end(va_list ap);`
- 参数：
    - `ap` -- 这是之前由同一函数中的 `va_start` 初始化的 `va_list` 对象
- 返回值：空

vsnprintf()

- 功能：将可变参数列表的格式化数据写入到大小为`n`的缓冲区
- 原型：`int vsnprintf(char *s, size_t n, const char *format, va_list arg);`
- 参数：
    - `s` -- 指向存储C字符串的缓冲区指针
    - `n` -- 缓冲区最大可用的字节数
    - `format` -- 包含格式化字符串的C字符串，和printf()相同
    - `arg`-- 识别使用`va_start`初始化的可变参数列表的值
- 返回值：
    - 成功 -- 被写入的字符数`n`
    - 失败 -- -1

sort()

- 功能：对在范围`[first, last]`内的元素进行排序，不能保证保留同等元素的顺序
- 原型：
    - `template <class RandomIt>, class <Compare>`
    - `void sort(RandomIt first, RandomIt last, Compare comp);`
- 参数：
    - `first` -- 排序元素范围的开始
    - `last` -- 排序元素范围的结束
    - `comp` -- 比较函数对象（即满足比较要求的对象）
- 返回值：空
- 需求：
    - 头文件：`#include <algorithm>`

sqrt()

- 功能：返回x的平方根
- 原型：`double sqrt(double x);`
- 参数：
  - `x` -- 需要处理的浮点数
- 返回值：
  - 成功 -- x的平方根
- 需求：
  - 头文件：`#include <math.h>`

opendir()

- 功能：打开一个目录文件
- 原型：`DIR *opendir(const char *dirname);`
- 参数：
  - `dirname` -- 目录
- 返回值：
  - 成功 -- 返回一个指向目录对象的指针
  - 失败 -- 返回一个空指针，并设置errno
- 需求：
  - 头文件：`#include <dirent.h>`
- 注意：
  - 定义在dirent.h中的DIR类型，表示的是一个**目录流**(directory stream)
  - 目录流是一个特定目录中所有目录项组成的一个有序序列。目录流中的条目不一定是按文件名的字母顺序排列的

readdir()

- 功能：读取一个目录文件中的数据
- 原型：`struct dirent *readdir(DIR *dirp);`
- 参数：
  - `dirp` -- 目录对象的指针
- 返回值：
  - 成功 -- 返回一个指向`struct dirent`结构的指针，结构中包含了与下一个目录项有关的信息
  - 失败 -- 返回一个NULL指针，并设置errno
    - `readdir()`的实现必须检测的错误只有一种，就是要返回的结构中的值无法正确表达，其错误码是`EOVERFLOW`
    - `readdir()`函数也返回NULL来指示目录的末尾，但在这种情况下它并不改变errno
- 需求：
  - 头文件：`#include <dirent.h>`
- 注意：
  - `readdir`在每次调用之后都将流转移到下一个位置上去

stat()

- 功能：通过名字来访问文件
- 原型：`int stat(const char *restrict path, struct stat *restrict buf);`
- 参数：
  - `path` -- 指定了需要返回状态的文件或符号链接的名字

    - `buf` -- 指向一个用户提供的缓冲区，这些函数都将信息存储在这个缓冲区中
- 返回值：
    - 成功 -- 0
    - 失败 -- -1，并设置errno

link()

- 功能：为`path1`指定的已存在文件创建一个新的目录项，这个文件位于`path2`指定的目录中
- 原型：`int link(const char *path1, const char *path2)`
- 参数：
    - `path1` -- 已存在文件
    - `path2` -- 指定的目录
- 返回值：
    - 成功 -- 0
    - 失败 -- -1
- 需求：
    - 头文件：`#include <unistd.h>`

unlink()

- 功能：删除了path指定的目录项
- 原型：`int unlink(const char *path);`
- 参数：
    - `path` -- 需要删除的目录项
- 返回值：
    - 成功 -- 0
    - 失败 -- -1
- 需求：
    - 头文件：`#include <unistd.h>`

symlink()

- 功能：创建一个符号链接
- 原型：`int symlink(const char *path1, const char *path2);`
- 参数：
    - `path1` -- 包含了将成为链接的内容的字符串
    - `path2` -- 链接的路径名
- 返回值：
    - 成功 -- 0
    - 失败 -- -1
- 需求：
    - 头文件：`#include <unistd.h>`
- 注意：
    - 换句话来说，path2就是新创建的链接，而新链接指向path1

sigset_t 信号集

- 信号集被定义为一种数据类型：

    - `typedef struct {unsigned long sig[_NSIG_WORDS];} sigset_t;`
- 信号集用来描述信号的集合，每个信号占用一位（64位）。
- Linux所支持的所有信号可以全部或部分的出现在信号集中，主要与信号阻塞相关函数配合使用

信号集操作定义的相关函数：

- `int sigemptyset(sigset_t *set);` 初始化由`set`指定的信号集，信号集里面的所有信号被清空，相当于`64`为置`0`；
- `int sigfillset(sigset_t *set);` 调用该函数后，`set`指向的信号集中将包含`linux`支持的`64`种信号，相当于`64`为都置`1`；
- `int sigaddset(sigset_t *set, int signum);`在`set`指向的信号集中加入`signum`信号，相当于将给定信号所对应的位置`1`；
- `int sigdelset(sigset_t *set, int signum);`在`set`指向的信号集中删除`signum`信号，相当于将给定信号所对应的位置`0`；
- `int sigismember(const sigset_t *set, int signum);`判定信号`signum`是否在`set`指向的信号集中，相当于检查给定信号所对应的位是`0`还是`1`。

信号处理

- 信号是内核提供的向用户态进程发送信息的机制,
- 常见的有使用`SIGUSR1`唤醒用户进程执行子程序或发生段错误时使用`SIGSEGV`保存用户错误现场.

sigwait()

- 功能：从`set`中选择任何挂起的信号或等待任何一个信号到达
- 原型：`int sigwait(const sigset_t *set, int sig);`
- 参数：
    - `set` -- 信号集
    - `sig` --
- 返回值：
    - 成功 --
    - 失败 --
- 需求：
    - 头文件：`#include <signal.h>`
- 注意：
    - sigwait是同步的等待信号的到来，而不是像进程中那样是异步的等待信号的到来
    - sigwait函数使用一个信号集作为他的参数，并且在集合中的任一个信号发生时返回该信号值，解除阻塞，然后可以针对该信号进行一些相应的处理。