

Sigaction函数使用方法

1. sigaction函数怎么使用.....	2
2. 细节那么多我记不住.....	5
3. `SA_RESTART`不懂.....	6
4. 这么多使用细节怎么记得住，而且常用的功能也就那几个，其实来signal的功能就简单明确需要处理的信号和相应的操作，搞那么多起什么用？还有为什么for(int i = 0; i < 64; i++)signal(i, SIG_IGN)是危险的？.....	8
5. 问题是先全部忽略然后再把需要的信号回复或者注册函数覆盖忽略.....	10



1. sigaction函数怎么使用

非常好，这个问题问得很关键。**sigaction** 是 Linux/Unix 系统中**最强大也最安全**的信号处理接口之一，用来取代旧的 **signal()** 函数。下面我会用**分层讲解 + 示例代码**的方式帮你彻底搞懂它的使用。

一、作用概述

sigaction() 用于：

- 设置信号处理函数（比如收到 **SIGINT** 时执行自定义函数）
- 查询原有处理动作
- 控制信号的行为（如是否自动重启被中断的系统调用）

相比 **signal()**，它更加可控、安全且可移植。

二、函数原型

```
c
#include <signal.h>

int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

参数说明：

参数	含义
signum	要处理的信号编号（如 SIGINT 、 SIGTERM 、 SIGCHLD ）
act	指向新信号处理方式的结构体（ NULL 表示不改变）
oldact	指向保存旧信号处理方式的结构体（可为 NULL ）

返回值：

- 成功返回 0
- 失败返回 -1，并设置 **errno**

三、关键结构体：struct sigaction

```
c
struct sigaction {
    void (*sa_handler)(int); // 传统信号处理函数
    void (*sa_sigaction)(int, siginfo_t *, void *); // 带信息的信号处理函数（更强大）
    sigset_t sa_mask; // 处理该信号时要屏蔽的信号集
```

```
};    int    sa_flags;    // 标志位控制行为
```

常用标志位（sa_flags）：

标志	含义
SA_SIGINFO	使用 sa_sigaction 而不是 sa_handler
SA_RESTART	自动重启被信号中断的系统调用
SA_NODEFER	处理信号时不自动屏蔽当前信号
SA_RESETHAND	信号处理后恢复为默认动作

四、最常见用法示例

示例 1：简单捕捉 SIGINT（Ctrl+C）

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

void handler(int signum) {
    printf("捕捉到信号 %d!\n", signum);
}

int main() {
    struct sigaction act;
    act.sa_handler = handler;
    sigemptyset(&act.sa_mask); // 清空信号屏蔽集
    act.sa_flags = 0;          // 默认为行为

    sigaction(SIGINT, &act, NULL);

    while (1) {
        printf("运行中...\n");
        sleep(1);
    }
}
```

运行后按下 **Ctrl+C**：

```
运行中...
运行中...
捕捉到信号 2！
运行中...
```

示例 2：使用 SA_SIGINFO 获取更详细信息

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

```
#include <unistd.h>

void handler(int signo, siginfo_t *info, void *context) {
    printf("收到信号 %d 来自进程 %d\n", signo, info->si_pid);
}

int main() {
    struct sigaction act;
    act.sa_sigaction = handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = SA_SIGINFO; // 使用 sa_sigaction 而不是 sa_handler

    sigaction(SIGUSR1, &act, NULL);

    printf("进程PID=%d, 等待信号...\n", getpid());
    while (1) pause(); // 等待信号
}
```

然后用另一个终端发送信号：

```
bash

kill -USR1 <PID>
```

输出：

```
收到信号 10 来自进程 12345
```

示例 3：屏蔽某些信号

```
c

sigemptyset(&act.sa_mask);
sigaddset(&act.sa_mask, SIGQUIT); // 在处理SIGINT时屏蔽SIGQUIT
```

这意味着在执行 **SIGINT** 的处理函数期间，如果收到了 **SIGQUIT**，会被暂时挂起，直到 **handler** 执行完毕才恢复处理。

🔧 五、实战建议

需求	建议写法
想安全处理信号（比如 SIGCHLD）	一定用 sigaction 而不是 signal
需要知道信号来源进程	设置 SA_SIGINFO
系统调用不能被中断	设置 SA_RESTART
调试或编写多进程代码	使用 sigaction 结合 sigprocmask 管理信号

🕒 六、总结

特性	signal()	sigaction()
可靠性	低	高
是否可移植	较差	POSIX 标准
能否获取详细信息	否	是（通过 SA_SIGINFO ）
控制选项	少	多

要不要我帮你写一个**进程通信（父子进程 + sigaction 处理 SIGCHLD）**的实战例子？这个能让你在实战中体会 **sigaction** 的意义，比如防止子进程僵尸化。



2. 细节那么多我记不住

这很正常。**sigaction** 这种系统调用细节确实太多，第一次看容易懵。你不用一次性全记住，我们分成 **三个阶段** 来学，只记最关键的部分就行。👉

👤 第一步：只记最常用的三行

先会用是第一步，其他的以后自然会懂。

```
c
struct sigaction act;
act.sa_handler = handler;           // 设置信号处理函数
sigemptyset(&act.sa_mask);         // 清空屏蔽集
act.sa_flags = 0;                  // 默认标志

sigaction(SIGINT, &act, NULL);      // 注册信号
```

理解要点：

- **SIGINT** 就是 Ctrl+C。
- 这几行表示：当收到 Ctrl+C，不退出程序，而是执行 **handler()**。

例子：

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

void handler(int sig) {
    printf("收到信号：%d\n", sig);
}

int main() {
    struct sigaction act;
    act.sa_handler = handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
```

```
sigaction(SIGINT, &act, NULL);

while (1) {
    printf("运行中...\n");
    sleep(1);
}
```

运行后 Ctrl+C：

```
运行中...
运行中...
收到信号：2
```

✅ 你现在已经能用 `sigaction` 替代 `signal()` 了。

第二步：稍微加一点点“配置”

当你看到别人的代码里写 `SA_SIGINFO` 或 `SA_RESTART`，只记住下面两个常见用法就行：

标志	含义	是否必须
<code>SA_SIGINFO</code>	想拿到信号来源信息（比如发送者PID）	可选
<code>SA_RESTART</code>	让系统调用被信号中断后自动重启	可选

第三步（等你熟了再说）

像 `sa_mask`、`siginfo_t`、`SA_NODEFER` 这些是进阶内容，你只要知道它们“能控制信号行为”，暂时不用背。

给你一句口诀记忆：

“三行搞定信号：`sa_handler` 设函数，`sigemptyset` 清屏蔽，`flags` 设零。”

我可以帮你写一个**更实战的小例子**：比如父进程用 `sigaction` 捕捉子进程退出（`SIGCHLD`），防止僵尸进程。要我给你写一个这个简化版练习吗？它会让你立刻理解 `sigaction` 真正的价值。



```
3. ↳
`SA_RESTART`
不懂
```

非常好 👍，你抓到重点了。你说你“不懂”，我猜你是指 `SA_RESTART` 这一行。这确实是信号机制里最难理解的一点之一。

我给你用最简单的方式讲清楚它👉

🧩 一句话总结

`SA_RESTART` 的意思是：

当系统调用（比如 `read()`、`write()`、`sleep()`）被信号中断时，自动重新执行，而不是直接返回错误。

🧠 背景（为什么会有这个）

Linux 里很多系统调用（比如 `read`、`wait`、`nanosleep`）是阻塞的，也就是说，它们会等待事件发生。但是如果在等待的过程中来了一个信号，就会被“打断”。

⚠️ 不加 `SA_RESTART` 时会发生什么

比如你写了：

```
c

sleep(5); // 睡5秒
```

假设你在第2秒时按了 `Ctrl+C`（触发 `SIGINT`），`sleep()` 被信号打断，它就会提前返回，并且返回值是还剩几秒：

```
c

int left = sleep(5);
printf("还剩 %d 秒\n", left);
```

输出：

```
还剩 3 秒
```

也就是说：信号中断了系统调用。

如果是 `read()`、`accept()` 之类的系统调用被打断，它会返回 `-1`，并设置 `errno = EINTR`。你得自己判断然后重启调用，非常麻烦。

✅ 加上 `SA_RESTART` 会怎样

你加上这句：

```
c

act.sa_flags = SA_RESTART;
```

意思是：

“如果系统调用被信号打断，就别返回错误，自动再执行一次。”

于是刚才的例子中，`sleep(5)` 就不会被中断，
信号处理完后，它会继续睡完剩下的时间。

举个更实用的例子：服务器 I/O

在网络服务器里常见写法：

```
c

act.sa_flags = SA_RESTART;
sigaction(SIGCHLD, &act, NULL);
```

如果没有 `SA_RESTART`，当子进程退出时触发 `SIGCHLD`，
`accept()` 会被打断返回错误。
如果你没检查 `errno == EINTR`，服务器就会异常退出。
加了 `SA_RESTART` 后，`accept()` 自动恢复继续等新连接，
程序更稳定。

□ 用最简单的比喻记住

“`SA_RESTART` 就像闹钟响了，我醒一下看看时间，
然后又继续睡，不会完全起床。”

要不要我写一个演示 `SA_RESTART` 效果的最短可运行例子？
你能亲眼看到加与不加的区别，一下就记住了。



4. 这么多使用细节怎么记得住，而且常用的功能也就那几个，其实来signal的功能就简单明确需要处理的信号和相应的操作，搞那么多起什么用？还有为什么for(int i = 0; i < 64; i++)signal(i, SIG_IGN)是危险的？

你这个问题问得很有思考性，咱们一条一条拆开说。

一、为什么要有 sigaction

你说得对，`signal()` 简单直接：
「信号编号 + 处理函数」，搞定。

那为什么还要 `sigaction()` 呢？

原因其实就两个字：**可靠**。

旧的 `signal()` 存在几个历史遗留问题：

1. **行为不一致**：不同系统上处理逻辑不同。
2. **容易丢信号**：信号可能在处理时被覆盖或忽略。
3. **不能设置信息屏蔽**：比如处理 `SIGINT` 时暂时不想被别的信号打断。
4. **无法精细控制**：拿不到信号来源进程、无法自动重启系统调用等。

而 `sigaction()` 就是为了解决这些问题：

- 它能设置屏蔽集 (`sa_mask`) ；
- 能控制行为 (`sa_flags`) ；
- 能拿到详细信息 (`sa_sigaction + SA_SIGINFO`) ；
- 最关键：**不会被信号打断出错** (`SA_RESTART`) 。

简单说：`sigaction()` 是“工业级”版本的 `signal()`。

🔧 二、记不住细节怎么办？

你根本不用记完，只要记口诀：

“三行搞定信号：`sa_handler` 设函数，`sigemptyset` 清屏蔽，`flags=0`。”

也就是：

```
c
struct sigaction act;
act.sa_handler = handler;
sigemptyset(&act.sa_mask);
act.sa_flags = 0;
sigaction(SIGINT, &act, NULL);
```

这已经是 90% 的实战场景。

等你写服务器、进程通信那类项目时，再慢慢加上 `SA_RESTART` 或 `SA_SIGINFO`。

💣 三、为什么 `for (int i = 0; i < 64; i++) signal(i, SIG_IGN)` 危险？

这句其实很多人第一次见都觉得聪明，
意思是“一次性忽略所有信号”。

但它非常危险，原因如下：

原因	说明
🚫 会屏蔽关键系统信号	比如 SIGKILL 、 SIGSTOP 等根本不能被捕捉或忽略，强行设置会出错。
💀 会导致无法终止进程	你忽略了 SIGTERM 、 SIGINT ，那外部 kill 、Ctrl+C 都杀不掉你。
🔄 会破坏系统机制	某些信号（如 SIGCHLD ）系统需要用来通知子进程状态，你忽略后子进程会变成僵尸。
🔪 不同系统信号数量不一样	64 只是 Linux 常见的最大值，别的系统可能更多或更少，存在越界风险。

所以，这种“一刀切忽略所有信号”的写法相当于：

“告诉内核别再和我通信”，
程序从此变成一座孤岛，甚至关不掉。

✅ 正确做法

你要忽略信号，可以有选择地：

```
c
signal(SIGPIPE, SIG_IGN); // 网络程序常用，防止写断开连接崩溃
signal(SIGCHLD, SIG_IGN); // 不关心子进程退出
```

或更安全一点：

```
c
struct sigaction act;
act.sa_handler = SIG_IGN;
sigemptyset(&act.sa_mask);
act.sa_flags = 0;
sigaction(SIGPIPE, &act, NULL);
```

🎯 结论总结

问题	简答
为啥 sigaction 这么复杂	因为它是 signal 的安全、可控升级版
必须记住全部细节吗	不用，只要会那“三行模板”
为什么 for signal(i, SIG_IGN) 危险	因为会屏蔽系统关键信号、导致僵尸进程、无法退出

要不要我帮你写一个“危险示例 vs 正确写法”的对比小程序？

你能亲眼看到忽略所有信号会怎样死机似的跑不掉。



5. 问题是先全部忽略然后再把需要的信号回复或者注册函数覆盖忽略

直说了：“先全部忽略，再把需要的信号恢复/覆盖”是个坏主意。确实能“跑起来”，但会埋下坑，理由很硬：

为什么这个套路危险

1. 竞态窗口 (race)

你先把一堆信号设成 `SIG_IGN`，在“恢复/注册”之前这段时间到来的信号会被直接忽略、**永远丢失**。比如本来你想处理 `SIGTERM` 做清理，结果它恰好在恢复之前来了——抱歉，错过了。

2. 有的信号根本不能忽略

`SIGKILL`、`SIGSTOP` **不允许**忽略/捕获；你的循环会报错，程序以为都处理了，实际没处理干净。

另外一批“致命/同步”类信号（如 `SIGSEGV/ILL/FPE/BUS`）**忽略毫无意义**，很多系统会直接拒绝或效果不确定；即便设置成功，也只是在更糟的状态下硬撑。

3. 行为副作用

随便忽略 `SIGCHLD` 会导致子进程自动回收（等价于 `SA_NOCLDWAIT`），改变进程模型；忽略 `SIGPIPE` 常见但要**明确知道**为什么（写断开套接字时不崩溃，改由 `write` 返回 `EPIPE`）。

4. 可移植性和维护性差

你用 `for (int i = 0; i < 64; ++i)` 这种魔法数字，一换平台/内核版本就不准；即使用 `NSIG` 也不代表“全部都该动”。

正确、安全的模式（推荐）

目标是：**安装时不丢信号、安装后按需处理**。做法不是“忽略”，而是“**安装期间先阻塞，再精确注册，最后解除阻塞**”。

步骤

1. 用 `sigfillset` 准备一个“全阻塞”的信号集。
2. `sigprocmask`（单线程）或 `pthread_sigmask`（多线程）把信号暂时**阻塞**起来（不是忽略）。
3. 按需对**特定信号**调用 `sigaction` 注册处理函数或设为 `SIG_IGN`。
4. 恢复原来的屏蔽集，解除阻塞。

迷你模板（单线程示例）

```
c
#include <signal.h>
#include <string.h>
#include <unistd.h>
#include <stdio.h>

static void on_term(int sig) {
    write(STDOUT_FILENO, "got SIGTERM\n", 12);
}

int main() {
    sigset_t all, old;
    sigfillset(&all);
    // 1) 安装期间阻塞所有信号（避免竞态丢信号）
    sigprocmask(SIG_SETMASK, &all, &old);

    // 2) 精确注册你关心的信号
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = on_term;
    sigemptyset(&sa.sa_mask);
```

```
sa.sa_flags = SA_RESTART; // 常用：避免系统调用被中断
sigaction(SIGTERM, &sa, NULL);

// 可选：明确忽略你“理解其后果”的少数信号
struct sigaction ign = {0};
ign.sa_handler = SIG_IGN;
sigaction(SIGPIPE, &ign, NULL); // 典型：网络程序

// 3) 安装完成，恢复原屏蔽（解除阻塞）
sigprocmask(SIG_SETMASK, &old, NULL);

// 4) 正常工作
for (;;) pause();
}
```

多线程额外注意

- 信号是**进程范围**的，但屏蔽是**线程级**的；在多线程程序里，请用 `pthread_sigmask` 在**主线程**统一屏蔽，然后用专门的“信号处理线程”做 `sigwaitinfo/sigtimedwait` 或安装处理器，避免信号随机落到任意线程。

什么时候该“忽略”

- **SIGPIPE**：大多数网络服务都会忽略，然后用返回值/`errno=EPIPE` 处理写失败。
- **SIGCHLD**：**明确**不需要 `wait()` 子进程时（例如你只 `fork` 启动外部程序且不关心它），可以设 `SIG_IGN` 或在 `sigaction` 里加 `SA_NOCLDWAIT`。但在你写多进程服务器时，通常**不要**这么做，而是正确 `waitpid` 回收。

总结一句硬话

“先全忽略再恢复”= 偷懒方案，存在竞态、丢信号、不可移植和隐性副作用。

正确姿势是：安装期间阻塞、精确注册、再解除阻塞。

需要忽略的，只忽略**少数你完全理解后果**的信号（如 **SIGPIPE**），其余用 `sigaction` 明确处理。

如果你愿意，我给你把上面的模板改成**父子进程 + 正确处理 **SIGCHLD****的版本，再配一个“错误写法对比”小样，本地跑一下你就能直观看到差异。