libuv 入门

发布 ***1.0.0***

作者：**Nikhil Marathe**　　翻译：**mingshun**

2013 年 09 月 10 日

Contents

CHAPTER 1

简介

本书是一个教程小集锦，内容围绕 libuv 高性能事件 I/O 库的使用， libuv 在 Windows 和 Unix 下提供了一致的 API。

这意味着涵盖了 libuv 的主要方面，但不是一个讨论所有函数及数据结构的完整参考。 libuv 的官方文档 则直接包含在 libuv 的头文件中。

本书尚在写作中，部分章节可能不完整，但我希望你会喜欢它，尽管它还在不断地完善中。

1.1 适用本书的读者

如果你正在阅读本书，你可能是：

1. 欲构建诸如守护进程或网络服务及客户端等底层程序的系统程序员。你会发现事件循环的动作方式很适合你的应用程序并打算使用 libuv。
2. 欲将已用 C 或 C++ 编写的平台 API 以异步 API 封装起来并暴露给 JavaScript 的 node.js 模块编写人员。你只会在 node.js 上下文中使用到 libuv。你还将需要另外一些本书没有涉及的与 v8/node.js 相关的内容。

本书假设你已经对 C 程序语言非常熟悉。

1.2 背景

node.js 项目始于 2009 年，是一个从浏览器上分离出来的 JavaScript 环境。通过使用 Google 的 V8 和 Marc Lehmann 的 libev ， node.js 将 I/O 模型与一种非常适合事件编程风格的程序语言结合起来，这和浏览器的结合方式一样。随着 node.js 的逐渐流行，让其支持 Windows 变得非常重要，可是 libev 只能在 Unix 上使用。在 Windows 下，类似于 kqueue 或 (e)poll 的内核事件通知机制为 IOCP。libuv 是 libev 或 IOCP 在不同平台上的抽象，提供给用户一套基于 libev 的 API。在 node-v0.9.0 的 libuv 中， libev 已被移除了 。

其后，libuv 逐渐成熟为高品质的系统程序库。除了 node.js，其用户还包含 Mozilla 的 Rust 程序语言及 各种语言的绑定。

第一个独立发布的 libuv 版本是 0.10.3。

1.3 示例代码

本书中的所有示例代码都包含在本书 GitHub 项目上的 source 部分中。 Clone/下载 本书并在 `code/` 目录下执行 `make` 以编译所有示例代码。本书和示例代码都是基于 libuv v0.11.1 的。 `libuv/` 目录中包含了一个 libuv 的版本。在编译示例代码的时候，libuv 代码会自动被编译。

CHAPTER 2

libuv 基础

libuv 严格使用 **异步** 、 **事件驱动** 的编程风格。其核心工作是提供事件循环及基于 I/O 或其他活动事件的回调机制。libuv 库包含了诸如计时器、非阻塞网络支持、异步文件系统访问、子进程等核心工具。

2.1 事件循环

在事件驱动程序设计中，应用程序关注特定事件并在它们发生时作出响应。libuv 负责从操作系统那里收集事件或监视其他资源的事件，而用户可以注册在某个事件发生时要调用的回调函数。事件循环通常会 不间断 运行。用伪代码可以表示成:

```
while 还有待处理的事件:
    e = 获取下一个事件
    if 有与 e 关联的回调函数:
        调用此回调函数
```

一些事件的例子:

- 文件写入就绪
- socket 有数据可读
- 计时器超时

事件循环被 `uv_run()` 封装起来 – 使用 libuv 的程序里最后都要调用的函数。

系统程序的普遍工作是处理输入和输出，而非复杂数据运算。使用常见的输入/输出函数（ `read` ，`fprintf` 等）的问题在于它们是 **阻塞的** 。向磁盘写入数据或从网络读取数据实际所花费的时间与处理的速度并不成正比。函数在任务没有完成之前都不会返回，所以你的程序在此期间什么都没做。对于需要高性能的程序而言，这成了主要的障碍，因为其他活动及 I/O 操作一直在等待着当前操作的完成。

一种常规的解决方案是使用线程。每个阻塞 I/O 操作在单独的线程（或线程池）中启动。当线程调用阻塞函数后，处理器可以调度运行另一个此刻需要使用 CPU 资源的线程。

libuv 的实现用了另一种方式 – **异步，非阻塞** 。大多数操作系统都提供了事件通知子系统。例如，套接字上常规的 `read` 调用会一直阻塞到发送方确实发送了一些数据回来为止。相反，应用程序可以要求操作系统来监视套接字，并将事件通知添加到队列中。应用程序可以在方便的时候（也许是进行一些需要大量处理器资源的数据处理后）去查看事件，并读取数据。因为应用程序在某刻/某处关注了感兴趣的事件，然后在其他时刻/其他地方才使用数据，所以是 **异步** 的。因为应用程序可以自由地处理其他任务，所以是 **非阻塞**

的。这非常符合 libuv 的事件循环处理方式，因为操作系统事件可被看作是不同的 libuv 事件。非阻塞保证了可以继续尽可能快地处理其他即将到来的事件 [1] 。

注解： 我们不关心 I/O 在后台是如何运作的，然而针对计算机硬件的工作方式 – 以线程作为处理器的基本执行单元，libuv 和系统通常会启动后台/工作线程和/或使用轮询来实现以非阻塞的方式来执行任务。

---

libuv 的核心开发者之一 Bert Belder 提供了一段解释 libuv 的架构及其内部工作原理的小视频。如果你之前没有接触过 libuv 或 libev ，这个视频可以为你作一个快速、实用的讲解。

2.2 Hello World

基于上面的基础，让我们来编写第一个 libuv 程序。它什么都没做，只是启动了一个事件循环就立即退出了。

helloworld/main.c

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

int main() {
    uv_loop_t *loop = uv_loop_new();

    printf("Now quitting.\n");
    uv_run(loop, UV_RUN_DEFAULT);

    return 0;
}
```

此程序在运行后会立刻退出，因为没有要处理的事件。你必须使用各种 API 函数告知 libuv 要监视的事件。

2.2.1 内置事件循环

内置事件循环由 libuv 提供，可以通过 `uv_default_loop()` 函数来访问。如果你只想要一个事件循环，就应该使用这个事件循环。

注解： node.js 使用内置事件循环作为其主要的事件循环。如果你是编写 node.js 模块，应该要清楚这点。

2.3 监视器

监视器是 libuv 用户用于监视特定事件的工具。监视器是以 `uv_TYPE_t` 命名的抽象结构体，这个类型表明了监视器的用途。libuv 支持的监视器完整列表如下：

[1] 当然，这取决于硬件的承载能力。

libuv watchers

```
typedef struct uv_loop_s uv_loop_t;
typedef struct uv_err_s uv_err_t;
typedef struct uv_handle_s uv_handle_t;
typedef struct uv_stream_s uv_stream_t;
typedef struct uv_tcp_s uv_tcp_t;
typedef struct uv_udp_s uv_udp_t;
typedef struct uv_pipe_s uv_pipe_t;
typedef struct uv_tty_s uv_tty_t;
typedef struct uv_poll_s uv_poll_t;
typedef struct uv_timer_s uv_timer_t;
typedef struct uv_prepare_s uv_prepare_t;
typedef struct uv_check_s uv_check_t;
typedef struct uv_idle_s uv_idle_t;
typedef struct uv_async_s uv_async_t;
typedef struct uv_process_s uv_process_t;
typedef struct uv_fs_event_s uv_fs_event_t;
typedef struct uv_fs_poll_s uv_fs_poll_t;
typedef struct uv_signal_s uv_signal_t;
```

注解: 所有监视器结构体都是 `uv_handle_t` 的子类型，并且在 libuv 和本书中所说的 **事件处理器** 通常就是指它们。

监视器是通过调用:

```
uv_TYPE_init(uv_TYPE_t*)
```

函数来创建。

注解: 有些监视器初始化函数要用事件循环作为第一个参数。

让监视器监听事件则调用:

```
uv_TYPE_start(uv_TYPE_t*, callback)
```

而停止监听则调用:

```
uv_TYPE_stop(uv_TYPE_t*)
```

回调函数是当监视器感兴趣的事件发生时，由 libuv 调用的函数。应用程序指定的逻辑一般会在回调函数中的实现。例如，IO 监视器的回调函数会接收从文件读取过来的数据，定时器回调函数会在超时的时候被触发等等。

2.3.1 空闲的监视器

这是一个使用监视器的例子。空闲监视器的回调函数会不停地被调用。在 实用工具 会讨论其更深层次的语义，但在这里我们先可以忽略它们。让我们用一个空闲的监视器来了解一下监视器的生命周期，并看看由于监视器的出现， *uv_run()*‘ 现在会是如何阻塞的。空闲监视器在计完数后停止，且因为没有活动的事件监视器， `uv_run()` 退出了。

idle-basic/main.c

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

int64_t counter = 0;

void wait_for_a_while(uv_idle_t* handle, int status) {
    counter++;

    if (counter >= 10e6)
        uv_idle_stop(handle);
}

int main() {
    uv_idle_t idler;

    uv_idle_init(uv_default_loop(), &idler);
    uv_idle_start(&idler, wait_for_a_while);

    printf("Idling...\n");
    uv_run(uv_default_loop(), UV_RUN_DEFAULT);

    return 0;
}
```

void *data 模式

说明没必要在栈上创建结构体类型

CHAPTER 3

文件系统

文件系统的简单读/写操作可使用 `uv_fs_*` 函数和 `uv_fs_t` 结构体来完成。

注解： libuv 的文件系统操作与 套接字操作 不同。套接字操作使用操作系统提供的非阻塞操作。文件系统在内部使用阻塞函数，但会在线程池中调用这些函数并在应用程序需要与之交互时通知已被注册到事件循环的监视器。

所有的文件系统操作函数都有形式 - 同步 和 异步 。

如果不指定回调函数， 同步 函数会自动被调用（并 **阻塞**）。函数的返回值相当于 Unix 返回值（通常是成功返回 0，出错返回 -1）。

在回调函数传入后， 异步 函数会被调用，并返回 0。

3.1 读/写文件

文件描述符使用下列函数获取

```
int uv_fs_open(uv_loop_t* loop, uv_fs_t* req, const char* path, int flags, int mode, uv_fs_cb cb)
```

`flags` 和 `mode` 为标准的 Unix 标志 。libuv 会负责将它们转换成相应的 Windows 标志。

文件描述符使用下列函数关闭

```
int uv_fs_close(uv_loop_t* loop, uv_fs_t* req, uv_file file, uv_fs_cb cb)
```

文件系统操作的回调函数有下列的签名：

```
void callback(uv_fs_t* req);
```

让我们看一个 `cat` 命令的简单实现。我们从给已打开的文件注册回调函数开始：

uvcat/main.c - 打开文件

```
1 void on_open(uv_fs_t *req) {
2     if (req->result != -1) {
3         uv_fs_read(uv_default_loop(), &read_req, req->result,
```

```
                    buffer, sizeof(buffer), -1, on_read);
    }
    else {
        fprintf(stderr, "error opening file: %d\n", req->errorno);
    }
    uv_fs_req_cleanup(req);
}
```

uv_fs_open 接收的回调函数的参数 uv_fs_t 中的 “result‘ 字段为文件描述符。若成功打开文件我们就开始读取它。

警告: 必须调用 uv_fs_req_cleanup() 函数来释放 libuv 内部分配的内存。

uvcat/main.c - 读取回调函数

```
void on_read(uv_fs_t *req) {
    uv_fs_req_cleanup(req);
    if (req->result < 0) {
        fprintf(stderr, "Read error: %s\n", uv_strerror(uv_last_error(uv_default_loop())));
    }
    else if (req->result == 0) {
        uv_fs_t close_req;
        // synchronous
        uv_fs_close(uv_default_loop(), &close_req, open_req.result, NULL);
    }
    else {
        uv_fs_write(uv_default_loop(), &write_req, 1, buffer, req->result, -1, on_write);
    }
}
```

在调用读取操作时，你应该传入一个 已初始化 的缓冲区，在触发读取回调函数前，数据将被缓存在此缓冲区内。

对于 EOF 的情况，读取回调函数的 result 字段为 0；读取出错时，其值为 -1；读取成功时，其值为已读取的字节数。

这里你会看到一个编写异步程序的共同模式。uv_fs_close() 的调用是同步执行的。 通常，一次性的或作为启动或关闭部分被分离出来的任务会被同步地执行，因为我们关心的是在程序执行主要任务和处理多个 *I/O* 源时的快速 *I/O* 。对于单个任务，其性能的差异是微不足道的，且编写出来的代码可能会更简单。

我们可以把这种模式概括为，原始系统调用的真正返回值被存放在 uv_fs_t.result 之中。

文件系统的写入同样是简单地使用 uv_fs_write()。 你的回调函数将会在写入完成后被调用 。在我们的例子中，回调函数只是简单地驱动下一轮读取操作。所以，回调函数让读写操作处在一个连锁的处理过程之内。

uvcat/main.c - 写入回调函数

```
void on_write(uv_fs_t *req) {
    uv_fs_req_cleanup(req);
    if (req->result < 0) {
        fprintf(stderr, "Write error: %s\n", uv_strerror(uv_last_error(uv_default_loop())));
    }
    else {
        uv_fs_read(uv_default_loop(), &read_req, open_req.result, buffer, sizeof(buffer), -1, on_read
```

```
8       }
9   }
```

注解: 保存在 errno 中的错误信息可以通过 `uv_fs_t.errorno` 访问，但它已被转换为标准的 `UV_*` 错误代码。现在还没有办法可以直接从 `errorno` 字段获取错误消息的字符串。

警告: 由于文件系统和磁盘驱动器的运行方式是以性能优先来配置的，写入'成功'的数据可能还未被提交到磁盘。关于更强的写入保证，请见 `uv_fs_fsync` 。

我们在 `main()` 函数中让多米诺骨牌开始翻倒吧:

uvcat/main.c

```
1   int main(int argc, char **argv) {
2       uv_fs_open(uv_default_loop(), &open_req, argv[1], O_RDONLY, 0, on_open);
3       uv_run(uv_default_loop(), UV_RUN_DEFAULT);
4       return 0;
5   }
```

3.2 文件系统操作

所有标准的文件系统操作，像 `unlink` 、 `rmdir` 、 `stat` 都支持异步调用方式，并具有直观的参数顺序。它们都遵循和读/写/打开调用相同的模式，调用结果返回到 `uv_fs_t.result` 字段中。完整的函数列表如下:

文件操作

```
UV_EXTERN int uv_fs_close(uv_loop_t* loop, uv_fs_t* req, uv_file file,
    uv_fs_cb cb);

UV_EXTERN int uv_fs_open(uv_loop_t* loop, uv_fs_t* req, const char* path,
    int flags, int mode, uv_fs_cb cb);

UV_EXTERN int uv_fs_read(uv_loop_t* loop, uv_fs_t* req, uv_file file,
    void* buf, size_t length, int64_t offset, uv_fs_cb cb);

UV_EXTERN int uv_fs_unlink(uv_loop_t* loop, uv_fs_t* req, const char* path,
    uv_fs_cb cb);

UV_EXTERN int uv_fs_write(uv_loop_t* loop, uv_fs_t* req, uv_file file,
    void* buf, size_t length, int64_t offset, uv_fs_cb cb);

UV_EXTERN int uv_fs_mkdir(uv_loop_t* loop, uv_fs_t* req, const char* path,
    int mode, uv_fs_cb cb);

UV_EXTERN int uv_fs_rmdir(uv_loop_t* loop, uv_fs_t* req, const char* path,
    uv_fs_cb cb);

UV_EXTERN int uv_fs_readdir(uv_loop_t* loop, uv_fs_t* req,
    const char* path, int flags, uv_fs_cb cb);
```

```
UV_EXTERN int uv_fs_stat(uv_loop_t* loop, uv_fs_t* req, const char* path,
    uv_fs_cb cb);

UV_EXTERN int uv_fs_fstat(uv_loop_t* loop, uv_fs_t* req, uv_file file,
    uv_fs_cb cb);

UV_EXTERN int uv_fs_rename(uv_loop_t* loop, uv_fs_t* req, const char* path,
    const char* new_path, uv_fs_cb cb);

UV_EXTERN int uv_fs_fsync(uv_loop_t* loop, uv_fs_t* req, uv_file file,
    uv_fs_cb cb);

UV_EXTERN int uv_fs_fdatasync(uv_loop_t* loop, uv_fs_t* req, uv_file file,
    uv_fs_cb cb);

UV_EXTERN int uv_fs_ftruncate(uv_loop_t* loop, uv_fs_t* req, uv_file file,
    int64_t offset, uv_fs_cb cb);

UV_EXTERN int uv_fs_sendfile(uv_loop_t* loop, uv_fs_t* req, uv_file out_fd,
    uv_file in_fd, int64_t in_offset, size_t length, uv_fs_cb cb);

UV_EXTERN int uv_fs_chmod(uv_loop_t* loop, uv_fs_t* req, const char* path,
    int mode, uv_fs_cb cb);

UV_EXTERN int uv_fs_utime(uv_loop_t* loop, uv_fs_t* req, const char* path,
    double atime, double mtime, uv_fs_cb cb);

UV_EXTERN int uv_fs_futime(uv_loop_t* loop, uv_fs_t* req, uv_file file,
    double atime, double mtime, uv_fs_cb cb);

UV_EXTERN int uv_fs_lstat(uv_loop_t* loop, uv_fs_t* req, const char* path,
    uv_fs_cb cb);

UV_EXTERN int uv_fs_link(uv_loop_t* loop, uv_fs_t* req, const char* path,
    const char* new_path, uv_fs_cb cb);
```

回调函数应该使用 `uv_fs_req_cleanup()` 释放 `uv_fs_t` 类型的参数。

3.3 缓冲区与流

libuv 中基本的 I/O 工具是流（`uv_stream_t`）。TCP 套接字，UDP 套接字和文件 I/O 及 IPC 管道全都被视作流的子类。

流的初始化使用各子类自定义的函数，然后使用这些函数操作它们

```
int uv_read_start(uv_stream_t*, uv_alloc_cb alloc_cb, uv_read_cb read_cb);
int uv_read_stop(uv_stream_t*);
int uv_write(uv_write_t* req, uv_stream_t* handle,
             uv_buf_t bufs[], int bufcnt, uv_write_cb cb);
```

基于流的函数使用起来比文件系统的更简单，且一旦调用了 `uv_read_start()`，libuv 会不断地从流中读取数据，直到调用了 `uv_read_stop()` 为止。

离散的数据单元为缓冲区 – `uv_buf_t`。它仅仅是指向一系列字节的指针（`uv_buf_t.base`）和长度值（`uv_buf_t.len`）。`uv_buf_t` 是轻量级的，且以值的方式传递。真正需要管理的是字节数据，它们必须由应用程序自行分配和释放。

为了展示流的使用方法，我们会要使用 uv_pipe_t。这样就可以在本地文件上使用流了[1]。这就是一个用 libuv 实现的 tee 命令了。所有操作都是异步地执行着，可见事件 I/O 的强大。两个写入操作不会相互阻塞，但我们要小心处理缓冲区数据的复制，确保在缓冲区数据被写入后才释放缓冲区。

该程序要这样执行:

```
./uvtee <output_file>
```

我们从需要处理的文件上打开管道开始。libuv 的文件管道默认是双向的。

uvtee/main.c - 在管道上读取数据

```
int main(int argc, char **argv) {
    loop = uv_default_loop();

    uv_pipe_init(loop, &stdin_pipe, 0);
    uv_pipe_open(&stdin_pipe, 0);

    uv_pipe_init(loop, &stdout_pipe, 0);
    uv_pipe_open(&stdout_pipe, 1);

    uv_fs_t file_req;
    int fd = uv_fs_open(loop, &file_req, argv[1], O_CREAT | O_RDWR, 0644, NULL);
    uv_pipe_init(loop, &file_pipe, 0);
    uv_pipe_open(&file_pipe, fd);

    uv_read_start((uv_stream_t*)&stdin_pipe, alloc_buffer, read_stdin);

    uv_run(loop, UV_RUN_DEFAULT);
    return 0;
}
```

对于使用命名管道的 IPC，uv_pipe_init() 的第三个参数 应该设置成 1。这在 进程 中会提到。调用 uv_pipe_open() 将文件描述符与文件关联起来。

我们开始监视 stdin。回调函数 alloc_buffer 在需要用新的缓冲区存放传入数据时被调用。调用 read_stdin 时会传入这些缓冲区。

uvtee/main.c - 读取缓冲区

```
uv_buf_t alloc_buffer(uv_handle_t *handle, size_t suggested_size) {
    return uv_buf_init((char*) malloc(suggested_size), suggested_size);
}

void read_stdin(uv_stream_t *stream, ssize_t nread, uv_buf_t buf) {
    if (nread == -1) {
        if (uv_last_error(loop).code == UV_EOF) {
            uv_close((uv_handle_t*)&stdin_pipe, NULL);
            uv_close((uv_handle_t*)&stdout_pipe, NULL);
            uv_close((uv_handle_t*)&file_pipe, NULL);
        }
    }
    else {
        if (nread > 0) {
            write_data((uv_stream_t*)&stdout_pipe, nread, buf, on_stdout_write);
```

[1] 见 管道

```
16              write_data((uv_stream_t*)&file_pipe, nread, buf, on_file_write);
17          }
18      }
19      if (buf.base)
20          free(buf.base);
21  }
```

标准的 malloc 在这里已经足够了，但你可以使用任何内存分配策略。例如，node.js 使用自身的 slab 内存分配器，该分配器把缓冲区与 V8 对象关联在一起。

出错时，读取回调函数的 nread 参数为 -1。这个错误可能会是 EOF，在这种情况下我们关闭所有的流，使用的是通用关闭函数 uv_close()，这个关闭函数负责处理基于其内部类型的句柄。反之，nread 为非负数，我们可以尝试向输出流写入这个数目的字节数据。最后别忘了缓冲区的分配和释放是由应用程序负责的，所以我们释放了数据。

uvtee/main.c - 向管道写入

```
1   typedef struct {
2       uv_write_t req;
3       uv_buf_t buf;
4   } write_req_t;
5
6   void free_write_req(uv_write_t *req) {
7       write_req_t *wr = (write_req_t*) req;
8       free(wr->buf.base);
9       free(wr);
10  }
11
12  void on_stdout_write(uv_write_t *req, int status) {
13      free_write_req(req);
14  }
15
16  void on_file_write(uv_write_t *req, int status) {
17      free_write_req(req);
18  }
19
20  void write_data(uv_stream_t *dest, size_t size, uv_buf_t buf, uv_write_cb callback) {
21      write_req_t *req = (write_req_t*) malloc(sizeof(write_req_t));
22      req->buf = uv_buf_init((char*) malloc(size), size);
23      memcpy(req->buf.base, buf.base, size);
24      uv_write((uv_write_t*) req, (uv_stream_t*)dest, &req->buf, 1, callback);
25  }
```

write_data() 将读取到的缓冲数据复制一份。重复一遍，不要把这个缓冲区带入写入完成所触发的回调函数中。为了避免这个问题，我们将写入请求与 write_req_t 中的缓冲区封装起来，再在回调函数中展开。

> **警告：** 如果你的程序打算用于其他程序，它可能会有意无意地向管道写入。这就使得它会受到 触发 SIGPIPE 信号而导致程序中止 的影响。最好在你的应用程序初始化过程中加入:
>
> ```
> signal(SIGPIPE, SIG_IGN)
> ```

3.4 文件变化事件

所有现代的操作系统都提供用于监视单个文件或目录并在文件被修改时发出通知的 API。libuv 封装了常用的文件变化通知库 [2]。这是 libuv 中几个比较不一致的部分之一。在不同的平台上，文件变化通知系统本身的差别就非常大，所以很难做到每样东西在各个平台上都能正常使用。为了演示，我会创建一个简易工具，它在被监视的文件发生变化时运行指定的命令：

```
./onchange <命令> <文件1> [文件2] ...
```

文件变化通知从调用 `uv_fs_event_init()` 开始：

onchange/main.c - 配置

```
    while (argc-- > 2) {
        fprintf(stderr, "Adding watch on %s\n", argv[argc]);
        uv_fs_event_init(loop, (uv_fs_event_t*) malloc(sizeof(uv_fs_event_t)), argv[argc], run_commar
    }
```

第三个参数是在实际要监视的文件或目录。最后一个参数 `flags` 可以是：

```
UV_FS_EVENT_WATCH_ENTRY = 1,
 UV_FS_EVENT_STAT = 2,
 UV_FS_EVENT_RECURSIVE = 3
```

`UV_FS_EVENT_WATCH_ENTRY` 和 `UV_FS_EVENT_STAT` 还没有任何功能。在支持的平台上，`UV_FS_EVENT_RECURSIVE` 会对子目录也进行监视。

其回调函数会接收以下参数：

1. `uv_fs_event_t *handle` - 监视器。监视器的 `filename` 字段为已被设置了监视的文件。
2. `const char *filename` - 如果被监视的是目录，则这是其中发生变化的文件。只在 Linux 和 Windows 下为非 `null`。其他平台可能为 `null`。
3. `int flags` - `UV_RENAME` 或 `UV_CHANGE` 之一。
4. `int status` - 目前为 0。

在这个例子中，我们只是简单地输出参数的内容及用 `system()` 来运行指定的命令。

onchange/main.c - 文件变化通知回调函数

```
void run_command(uv_fs_event_t *handle, const char *filename, int events, int status) {
    fprintf(stderr, "Change detected in %s: ", handle->filename);
    if (events == UV_RENAME)
        fprintf(stderr, "renamed");
    if (events == UV_CHANGE)
        fprintf(stderr, "changed");

    fprintf(stderr, " %s\n", filename ? filename : "");
    system(command);
}
```

[2] Linux 为 inotify，Darwin 为 FSEvents，BSD 系列为 kqueue，Windows 为 ReadDirectoryChangesW，Solaris 为 event ports，不支持 Cygwin

CHAPTER 4

连接网络

用 libuv 连接网络与直接使用 BSD 套接字接口没有太大区别，有些东西用起来更简单，都是非阻塞的，但概念都一样。此外，libuv 提供用于抽象那些烦人的，重复的且底层的任务的工具函数，比如使用 BSD 套接字结构体建立套接字，DNS查找，及调整各个套接字参数。

`uv_tcp_t` 和 `uv_udp_t` 结构体用于网络 I/O。

4.1 TCP

TCP 是面向连接的流协议，因此它是基于 libuv 流基础架构的。

4.1.1 服务器

服务器套接字的调用过程是：

1. `uv_tcp_init` TCP 监视器。
2. `uv_tcp_bind` 这个监视器。
3. 在监视器上调用 `uv_listen` 注册回调函数，该回调函数会在客户端发起新连接时被调用。
4. 使用 `uv_accept` 接受连接。
5. 使用 流操作函数 与客户端进行通信。

这是一个简单的回应服务器

tcp-echo-server/main.c - 监听套接字

```
int main() {
    loop = uv_default_loop();

    uv_tcp_t server;
    uv_tcp_init(loop, &server);

    struct sockaddr_in bind_addr = uv_ip4_addr("0.0.0.0", 7000);
    uv_tcp_bind(&server, bind_addr);
```

```
    int r = uv_listen((uv_stream_t*) &server, 128, on_new_connection);
    if (r) {
        fprintf(stderr, "Listen error %s\n", uv_err_name(uv_last_error(loop)));
        return 1;
    }
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

你可以看到 uv_ip4_addr 工具函数，这个函数将人类可读的 IP 地和端口转换成 BSD 套接字 API 所需的 sockaddr_in 结构体。反之使用 uv_ip4_name。

注解: 与 ip4 的函数类似，还有 uv_ip6_*，免得不够清楚。

大多数建立连接的函数都是正常的函数，因为它们都是计算密集型的。uv_listen 又使我们回到 libuv 的回调编程风格。第二个参数是缓冲队列 – 连接队列的最大长度。

当客户端发起连接时，回调函数要为客户端套捆字创建监视器，并用 uv_accept 与之关联起来。在这种情况下，我们也对从流中读取数据感兴趣。

tcp-echo-server/main.c - 接受客户端连接

```
void on_new_connection(uv_stream_t *server, int status) {
    if (status == -1) {
        // error!
        return;
    }

    uv_tcp_t *client = (uv_tcp_t*) malloc(sizeof(uv_tcp_t));
    uv_tcp_init(loop, client);
    if (uv_accept(server, (uv_stream_t*) client) == 0) {
        uv_read_start((uv_stream_t*) client, alloc_buffer, echo_read);
    }
    else {
        uv_close((uv_handle_t*) client, NULL);
    }
```

余下的函数使用方法与流的例子类似，可以在示例代码中找到。记得在套接字不再需要的时候调用 uv_close。如果不想接受连接，你甚至可以在 uv_listen 的回调函数中调用 uv_close。

4.1.2 客户端

在服务器调用 bind/listen/accept 的地方，客户端要做的只是调用 uv_tcp_connect。同样是 uv_connect_cb 风格的回调函数 uv_listen，被换成了 uv_tcp_connect。试试:

```
uv_tcp_t socket;
uv_tcp_init(loop, &socket);

uv_connect_t connect;

struct sockaddr_in dest = uv_ip4_addr("127.0.0.1", 80);

uv_tcp_connect(&connect, &socket, dest, on_connect);
```

on_connect 会在连接建立后被调用。

4.2 UDP

用户数据报协议 提供无连接，不可靠的网络连接服务。因而 libuv 并不提供流，而是通过 *uv_udp_t* （用于接收）和 *uv_udp_send_t* （用于发送）及相关的函数提供非阻塞的 UDP 支持。也就是说，实际用于读/写的 API 与正常的流读取非常相似。看看如何使用 UDP，这个示例展示了从 DHCP 服务器获取 IP 地址的第一步 – DHCP 查找。

注解: 要在 **root** 权限下运行 *udp-dhcp*，因为它会使用小于 1024 的端口号。

---

udp-dhcp/main.c - 建立并发送 UDP 数据包

```
uv_loop_t *loop;
uv_udp_t send_socket;
uv_udp_t recv_socket;

int main() {
    loop = uv_default_loop();

    uv_udp_init(loop, &recv_socket);
    struct sockaddr_in recv_addr = uv_ip4_addr("0.0.0.0", 68);
    uv_udp_bind(&recv_socket, recv_addr, 0);
    uv_udp_recv_start(&recv_socket, alloc_buffer, on_read);

    uv_udp_init(loop, &send_socket);
    uv_udp_bind(&send_socket, uv_ip4_addr("0.0.0.0", 0), 0);
    uv_udp_set_broadcast(&send_socket, 1);

    uv_udp_send_t send_req;
    uv_buf_t discover_msg = make_discover_msg(&send_req);

    struct sockaddr_in send_addr = uv_ip4_addr("255.255.255.255", 67);
    uv_udp_send(&send_req, &send_socket, &discover_msg, 1, send_addr, on_send);

    return uv_run(loop, UV_RUN_DEFAULT);
}
```

注解: IP 地址 `0.0.0.0` 用于绑定所有网络接口。 IP 地址 `255.255.255.255` 是广播地址，意味着数据包会被发送到子网上的所有网络接口。端口 `0` 表示 OS 随机注册端口。

首先，我们建立接收数据的套接字，绑定所有网络接口及 68 端口（DHCP 客户端）并启动在其上启动读取监视器。然后，我们建立一个类型的发送数据套接字，并使用 `uv_udp_send` 向 67 端口（DHCP 服务器）发送 广播消息。

必需 设置广播标志，否则你会收到 `EACCES` 错误 [1]。发送的确切消息内容与本书无关，如果你感兴趣可以研究一下源代码。和之前一样，如果出现错误，读和写的回调函数会收到状态码 -1。

因为 UDP 套接字不连接到特定的终端，读取数据的回调函数接收了一个额外的关于数据包发送方的参数。`flags` 参数可能为 `UV_UDP_PARTIAL`，如果你的内存分配器提供的缓冲区大小不足以保存接收到的数据的话。 这种情况下，*OS* 会删除存放不下的数据 （那就是你最后得到的 UDP！）。

[1] http://beej.us/guide/bgnet/output/html/multipage/advanced.html#broadcast

udp-dhcp/main.c - 读取数据包

```
void on_read(uv_udp_t *req, ssize_t nread, uv_buf_t buf, struct sockaddr *addr, unsigned flags) {
    if (nread == -1) {
        fprintf(stderr, "Read error %s\n", uv_err_name(uv_last_error(loop)));
        uv_close((uv_handle_t*) req, NULL);
        free(buf.base);
        return;
    }

    char sender[17] = { 0 };
    uv_ip4_name((struct sockaddr_in*) addr, sender, 16);
    fprintf(stderr, "Recv from %s\n", sender);

    // ... DHCP specific code

    free(buf.base);
    uv_udp_recv_stop(req);
}
```

4.2.1 UDP 选项

存活时间

被发送到套接字上数据包的 TTL （译者注：存活时间）可以使用 `uv_udp_set_ttl` 来修改。

仅 IPv6 协议栈

IPv6 套接字可同时用于 IPv4 和 IPv6 的通信。如果你想限制只使用 IPv6 的套接字，给 `uv_udp_bind6` 传入 `UV_UDP_IPV6ONLY` 标志 [2]。

多路广播

套接字订阅/取消订阅多路广播组，可以使用：

```
UV_EXTERN int uv_udp_set_membership(uv_udp_t* handle,
    const char* multicast_addr, const char* interface_addr,
    uv_membership membership);
```

`membership` 为 `UV_JOIN_GROUP` 或 `UV_LEAVE_GROUP`。

默认启用多路广播的本地回路 [3]，使用 `uv_udp_set_multicast_loop` 关闭。

多路广播数据包的包存活时间可以使用 `uv_udp_set_multicast_ttl` 修改主。

4.3 查询 DNS

libuv 提供异步 DNS 解析。对此，它提供了自己的 `getaddrinfo` 替代函数 [4]。在回调函数中，你可以利用已经获取得的地址来执行正常的套接字操作。让我们连接到 Freenode 来看看 DNS 解析的示例吧。

[2] 在 Windows 平台上只有 Windows Vista 及之后的版本才支持。
[3] http://www.tldp.org/HOWTO/Multicast-HOWTO-6.html#ss6.1
[4] libuv 在其线程池中使用系统的 `getaddrinfo`。 libuv v0.8.0 及早期版本还支持 c-ares 作为替代方案，但在 v0.9.0 里已被移除。

dns/main.c

```
int main() {
    loop = uv_default_loop();

    struct addrinfo hints;
    hints.ai_family = PF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;
    hints.ai_flags = 0;

    uv_getaddrinfo_t resolver;
    fprintf(stderr, "irc.freenode.net is... ");
    int r = uv_getaddrinfo(loop, &resolver, on_resolved, "irc.freenode.net", "6667", &hints);

    if (r) {
        fprintf(stderr, "getaddrinfo call error %s\n", uv_err_name(uv_last_error(loop)));
        return 1;
    }
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

如果 `uv_getaddrinfo` 返回非零值，就表明出错了，你的回调函数也不会被调用。在 `uv_getaddrinfo` 返回后，可以立刻释放所有参数使用的内存空间。*hostname*，*servname* 和 *hints* 结构体在 the getaddrinfo man page 文档中有详细说明。

在解析回调函数中，你可以从链表 `struct addrinfo(s)` 中获取 IP。这个示例还使用了 `uv_tcp_connect`。`uv_freeaddrinfo` 必需在回调函数中调用。

dns/main.c

```
void on_resolved(uv_getaddrinfo_t *resolver, int status, struct addrinfo *res) {
    if (status == -1) {
        fprintf(stderr, "getaddrinfo callback error %s\n", uv_err_name(uv_last_error(loop)));
        return;
    }

    char addr[17] = {'\0'};
    uv_ip4_name((struct sockaddr_in*) res->ai_addr, addr, 16);
    fprintf(stderr, "%s\n", addr);

    uv_connect_t *connect_req = (uv_connect_t*) malloc(sizeof(uv_connect_t));
    uv_tcp_t *socket = (uv_tcp_t*) malloc(sizeof(uv_tcp_t));
    uv_tcp_init(loop, socket);

    connect_req->data = (void*) socket;
    uv_tcp_connect(connect_req, socket, *(struct sockaddr_in*) res->ai_addr, on_connect);

    uv_freeaddrinfo(res);
}
```

4.4 网络适配器

操作系统网络接口的信息可以通过 libuv 使用 uv_interface_addresses 来获取。这个简单的程序只是输出全部网络接口的，这样你就可以得知有哪些可用的字段。这对于你在启动允许绑定 IP 地址的服务的时候非常有用。

interfaces/main.c

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

int main() {
    char buf[512];
    uv_interface_address_t *info;
    int count, i;

    uv_interface_addresses(&info, &count);
    i = count;

    printf("Number of interfaces: %d\n", count);
    while (i--) {
        uv_interface_address_t interface = info[i];

        printf("Name: %s\n", interface.name);
        printf("Internal? %s\n", interface.is_internal ? "Yes" : "No");

        if (interface.address.address4.sin_family == AF_INET) {
            uv_ip4_name(&interface.address.address4, buf, sizeof(buf));
            printf("IPv4 address: %s\n", buf);
        }
        else if (interface.address.address4.sin_family == AF_INET6) {
            uv_ip6_name(&interface.address.address6, buf, sizeof(buf));
            printf("IPv6 address: %s\n", buf);
        }

        printf("\n");
    }

    uv_free_interface_addresses(info, count);
    return 0;
}
```

对于回路网络接口，is_internal 的值为真。注意，如果物理网络接口有多个 IPv4/IPv6 地址，其名称会多次出现，而每个地址只会出现一次。

CHAPTER 5

线程

等一下？为什么是线程？事件循环不应该是 云端编程 的 **方法** 吗？哦不。线程依然是处理器工作的媒介，有时候线程还很有用，尽管你可能要很吃力地阅读同步原语。

线程在内部被用于伪装所有系统调用的异步特性。libuv 也是使用线程，让你的程序能够异步地执行那些实际上是阻塞的任务，它是通过创建线程并在任务完成时收集其结果来实现的。

目前，有两个主要的线程库：Windows 线程实现和 pthreads。libuv 的线程 API 模拟的是 pthread 的 API，语义也有很多相似的地方。

libuv 线程工具中值得注意的方面是它是包含在 libuv 中的一个部分。鉴于其他特性非常紧密地依赖于事件循环和回调原理，线程则是完全不可知的，它们在需要的时候阻塞，直接通过返回值发出错误信号，正如 第一个示例 所示的那样，甚至还不需要一个正在运行的事件循环。

libuv 线程 API 也是非常有限的，因为其线程的语义和语法在所有平台上都不相同，表现为不同程度的完整性。

本章作了以下的假设： **只有一个事件循环运行在单个线程中（主线程）** 。没有其他线程与事件循环交互（除了使用 `uv_async_send`）。`multiple` 会提到在不同线程运行事件循环及对它们进行管理。

5.1 核心线程操作

这里没有展示很多东西，你只是使用 `uv_thread_create()` 来启动一个线程，再使用 `uv_thread_join()` 来等待它结束。

thread-create/main.c

```
int main() {
    int tracklen = 10;
    uv_thread_t hare_id;
    uv_thread_t tortoise_id;
    uv_thread_create(&hare_id, hare, &tracklen);
    uv_thread_create(&tortoise_id, tortoise, &tracklen);

    uv_thread_join(&hare_id);
    uv_thread_join(&tortoise_id);
    return 0;
}
```

小技巧: uv_thread_t 只是 Unix 上 pthread_t 的别名，但这是实现细节，避免依赖它总是对的。

第二个参数是要作为线程入口点的函数，最后一个参数是可以用于给线程传入自定义参数的 void * 型参数。hare 函数会在一个独立的线程中运行，被操作系统抢先高调度:

thread-create/main.c

```
1  void hare(void *arg) {
2      int tracklen = *((int *) arg);
3      while (tracklen) {
4          tracklen--;
5          sleep(1);
6          fprintf(stderr, "Hare ran another step\n");
7      }
8      fprintf(stderr, "Hare done running!\n");
9  }
```

不同于 pthread_join() 允许目标线程利用第二个参数回传一个值给调用线程，uv_thread_join() 没有。要传值就使用 线程间通信。

5.2 同步原语

这部分是特意写简单带过。本书不是关于线程的，所以我只在这里罗列在 libuv API 中比较不同的内容。其他内容请见 pthreads 的 man pages。

5.2.1 互斥锁

互斥锁函数是 **直接** 对应到 pthread 中相应的函数的。

libuv 互斥锁函数

```
UV_EXTERN int uv_mutex_init(uv_mutex_t* handle);
UV_EXTERN void uv_mutex_destroy(uv_mutex_t* handle);
UV_EXTERN void uv_mutex_lock(uv_mutex_t* handle);
UV_EXTERN int uv_mutex_trylock(uv_mutex_t* handle);
UV_EXTERN void uv_mutex_unlock(uv_mutex_t* handle);
```

在调用成功时，uv_mutex_init() 和 uv_mutex_trylock() 会返回 0，否则返回 -1 而不是错误代码。

如果编译 *libuv* 时打开了调试开关，uv_mutex_destroy()、uv_mutex_lock() 和 uv_mutex_unlock() 会在出错时 abort()。同样地，如果错误原因不是 EAGAIN，那 uv_mutex_trylock() 会异常退出。

某些平台支持递归互斥锁，但你不应该依赖它们。如是一个锁定了互斥锁的线程再次试图锁定它，BSD 互斥锁的实现会报错。例如，像下面的结构那样:

```
uv_mutex_lock(a_mutex);
uv_thread_create(thread_id, entry, (void *)a_mutex);
uv_mutex_lock(a_mutex);
// 更多代码
```

可能要保持等待，直到另一个线程初始化某些东西然后解除锁定 a_mutex 为止，但如果在调试模式下会导致你的程序挂掉或在第二次调用 uv_mutex_lock() 时返回错误。

注解： Linux 支持带属性的递归互斥锁，但 libuv 没有暴露这个 API。

5.2.2 读写锁

读写锁是更细粒度的访问机制。两个读取者可以同时访问共享内存。读取者持有的写入者可能不需要进行锁定。写入者持有的读取者或写入者可能不需要锁。读写锁在数据库中频繁使用。这里是一个玩具级的示例。

locks/main.c - 简单的读写锁

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

uv_barrier_t blocker;
uv_rwlock_t numlock;
int shared_num;

void reader(void *n)
{
    int num = *(int *)n;
    int i;
    for (i = 0; i < 20; i++) {
        uv_rwlock_rdlock(&numlock);
        printf("Reader %d: acquired lock\n", num);
        printf("Reader %d: shared num = %d\n", num, shared_num);
        uv_rwlock_rdunlock(&numlock);
        printf("Reader %d: released lock\n", num);
    }
    uv_barrier_wait(&blocker);
}

void writer(void *n)
{
    int num = *(int *)n;
    int i;
    for (i = 0; i < 20; i++) {
        uv_rwlock_wrlock(&numlock);
        printf("Writer %d: acquired lock\n", num);
        shared_num++;
        printf("Writer %d: incremented shared num = %d\n", num, shared_num);
        uv_rwlock_wrunlock(&numlock);
        printf("Writer %d: released lock\n", num);
    }
    uv_barrier_wait(&blocker);
}

int main()
{
    uv_barrier_init(&blocker, 4);

    shared_num = 0;
```

```
42      uv_rwlock_init(&numlock);
43
44      uv_thread_t threads[3];
45
46      int thread_nums[] = {1, 2, 1};
47      uv_thread_create(&threads[0], reader, &thread_nums[0]);
48      uv_thread_create(&threads[1], reader, &thread_nums[1]);
49
50      uv_thread_create(&threads[2], writer, &thread_nums[2]);
51
52      uv_barrier_wait(&blocker);
53      uv_barrier_destroy(&blocker);
54
55      uv_rwlock_destroy(&numlock);
56      return 0;
57  }
```

运行并观察读取者有时会重叠。对于多个写入者的情况，调度器通常会给它们更高的优先级，所以如果你增加两个写入者，你会看到两个写入者在所有读取者再次获得执行机会前抢先完成任务。

5.2.3 其他同步原语

libuv 还支持 信号量 、 条件变量 和 屏障 ，它们的 API 与 pthread 中对应的 API 非常类似。

对于条件变量的情况，ibuv 也有等待超时，以平台特定的方式提供 [1]。

此外，libuv 提供了便利的 `uv_once()` 函数。多个线程可以尝试传入一个看守者和一个函数指针来调用 `uv_once()`， **只有第一个才能获得运行机会，而这个函数会且只会被调用一次**:

```
/* 初始化看守者 */
static uv_once_t once_only = UV_ONCE_INIT;

int i = 0;

void increment() {
    i++;
}

void thread1() {
    /* ... 工作代码 */
    uv_once(once_only, increment);
}

void thread2() {
    /* ... 工作代码 */
    uv_once(once_only, increment);
}

int main() {
    /* ... 创建线程 */
}
```

所有线程结束后，`i == 1`。

[1] https://github.com/joyent/libuv/blob/master/include/uv.h#L1853

5.3 libuv 工作队列

`uv_queue_work()` 是允许应用程序在独立线程中执行任务便利函数，还拥有当任务完成时被触发的回调函数。表面上看是很简单的函数，而 `uv_queue_work()` 最吸收人的地方在于它潜在地允许任何第三方库与事件循环范式一起使用。在使用事件循环时，必需确保当执行 *I/O* 或消耗大量 *CPU* 的任务时，没有函数在事件循环线程中周期性阻塞，因为这意味着拖慢循环且不能以全速来处理事件。

不过，许多现存代码都是在多线程中使用的阻塞函数（例如在伪装下执行 I/O 的例程），如果你追求响应能力（经典的'一个线程对应一个客户端'的服务器模型），且想让它们与事件循环库一起工作的话，通常会涉及到让你自己的系统在单独的线程是执行任务。对此，libuv 只提供方便的抽象接口。

这个简单的示例受到 node.js is cancer 的启发而编写的。我们会计算斐波纳契数列，在计算过程中会作短暂休眠，但在单独的线程中执行，这样阻塞和和占用 CPU 的任务就阻碍不了事件循环执行其他任务了。

queue-work/main.c - 懒惰的斐波纳契数列

```
void fib(uv_work_t *req) {
    int n = *(int *) req->data;
    if (random() % 2)
        sleep(1);
    else
        sleep(3);
    long fib = fib_(n);
    fprintf(stderr, "%dth fibonacci is %lu\n", n, fib);
}

void after_fib(uv_work_t *req, int status) {
    fprintf(stderr, "Done calculating %dth fibonacci\n", *(int *) req->data);
}
```

实际的任务函数很简单，没有任何迹象显示它会运行在单独的线程中。`uv_work_t` 结构体就是线索。你可以通过它的 `void* data` 字体段传入任意数据并使用它与线程通信。但如果你在两个线程都运行的时候不断修改变量的值，你要确保你使用了正确的锁。

The trigger is `uv_queue_work`:

queue-work/main.c

```
int main() {
    loop = uv_default_loop();

    int data[FIB_UNTIL];
    uv_work_t req[FIB_UNTIL];
    int i;
    for (i = 0; i < FIB_UNTIL; i++) {
        data[i] = i;
        req[i].data = (void *) &data[i];
        uv_queue_work(loop, &req[i], fib, after_fib);
    }

    return uv_run(loop, UV_RUN_DEFAULT);
}
```

线程函数会在单独的线程中启动，传入 `uv_work_t` 结构体，一旦函数返回，*after* 函数就会被调用，并再次传同一个结构体。

为阻塞库编写封装器时，一个共同的 模式 是使用“接力棒”变量来交换数据。

从 libuv *0.9.4* 版开始，新增了 `uv_cancel()` 函数。它允许你在 libuv 工作队列中取消任务。只有 还没有启动的 任务才可以被取消。如果任务 已经开始运行，或执行完毕，调用 `uv_cancel()` **将失效** 。

> **警告:** `uv_cancel()` 只在 Unix 上可用！

如果用户请求中止，`uv_cancel()` 对清理待运行任务非常有用。例如，音乐播放器可能会将多个需要扫描音频文件的目录放进队列里。如果用户终止程序，它应该立刻退出并不再等待所有待处理请求的运行。

让我们修改一下斐波纳契数列的示例来展示一下 `uv_cancel()`。我们先建立一个终止信息的处理器。

queue-cancel/main.c

```
int main() {
    loop = uv_default_loop();

    int data[FIB_UNTIL];
    int i;
    for (i = 0; i < FIB_UNTIL; i++) {
        data[i] = i;
        fib_reqs[i].data = (void *) &data[i];
        uv_queue_work(loop, &fib_reqs[i], fib, after_fib);
    }

    uv_signal_t sig;
    uv_signal_init(loop, &sig);
    uv_signal_start(&sig, signal_handler, SIGINT);

    return uv_run(loop, UV_RUN_DEFAULT);
}
```

当用户通过 `Ctrl+C` 触发信息，我们就给所有工作者发送 `uv_cancel()`。对那些已经运行或终止的工作者，`uv_cancel()` 会返回 `-1`。

queue-cancel/main.c

```
void signal_handler(uv_signal_t *req, int signum)
{
    printf("Signal received!\n");
    int i;
    for (i = 0; i < FIB_UNTIL; i++) {
        uv_cancel((uv_req_t*) &fib_reqs[i]);
    }
    uv_signal_stop(req);
}
```

对于已经成功取消的任务，*after* 函数会被调用，`status` 设置为 `-1`，而循环错误代码设置为 `UV_ECANCELED`。

queue-cancel/main.c

```
void after_fib(uv_work_t *req, int status) {
    if (status == -1 && uv_last_error(loop).code == UV_ECANCELED)
        fprintf(stderr, "Calculation of %d cancelled.\n", *(int *) req->data);
}
```

uv_cancel() 可与 uv_fs_t 及 uv_getaddrinfo_t 请求一起使用。对于文件系统中的函数，uv_fs_t.errorno 会被设置为 UV_ECANCELED。

小技巧： 一个设计良好的程序应该提供终止已经运行的持续时间长的工作者的方法。这样的工作者可以不断地检查一个只由主进程设置为终止信号的变量。

5.4 线程间通信

有时你确实要 在 多个线程运行 时，在它们之间互相发送消息。例如你可能会在单独的线程中运行一些持续时间长的任务（可能使用 uv_queue_work），但想给主线程报告其进度。这是一个简单的示例，让下载管理器给用户报告正在运行的下载任务的状态。

progress/main.c

```
uv_loop_t *loop;
uv_async_t async;

int main() {
    loop = uv_default_loop();

    uv_work_t req;
    int size = 10240;
    req.data = (void*) &size;

    uv_async_init(loop, &async, print_progress);
    uv_queue_work(loop, &req, fake_download, after);

    return uv_run(loop, UV_RUN_DEFAULT);
}
```

异步线程通信 在事件循环中 进行，所以尽管任何线程都可以充当消息的发送者，但只有 libuv 事件循环的线程才可以充当接收者（更精确地说事件循环才是接收者）。每当接收到消息时，libuv 都会通过异步监视器来调用回调函数（print_progress）。

警告： 充分地了解消息的发送是 异步的，这一点是非常重要的。回调函数可能会在另一个线程调用了 uv_async_send 之后被立即调用，或者在不久之后被调用。libuv 还可能将多个调用合并到 uv_async_send 中，然后只调用一次你的回调函数。libuv 唯一能担保的是 – 在调用了 uv_async_send 之后，回调函数 至少 会被调用 一次 。如果你没有待调用的 uv_async_send，则回调函数不会被调用。如果你调用了二次或以上，且 libuv 还没有机会执行回调函数的话，对于多次调用 uv_async_send 的情况，libuv 可能 只 调用你的回调函数 一次。你的回调函数决不会在在一个事件中被调用两次。

progress/main.c

```
void fake_download(uv_work_t *req) {
    int size = *((int*) req->data);
    int downloaded = 0;
    double percentage;
    while (downloaded < size) {
        percentage = downloaded*100.0/size;
        async.data = (void*) &percentage;
        uv_async_send(&async);

        sleep(1);
        downloaded += (200+random())%1000; // can only download max 1000bytes/sec,
                                           // but at least a 200;
    }
}
```

在下载函数中，我们修改了进度指示器，并用 uv_async_send 将要发送的消息放进队列里。记住：uv_async_send 也是非阻塞的，并且会立即返回。

progress/main.c

```
void print_progress(uv_async_t *handle, int status /*UNUSED*/) {
    double percentage = *((double*) handle->data);
    fprintf(stderr, "Downloaded %.2f%%\n", percentage);
}
```

回调函数是标准的 libuv 模式，从监视器中提取数据。

最后，务必记住清理监视器。

progress/main.c

```
void after(uv_work_t *req, int status) {
    fprintf(stderr, "Download complete\n");
    uv_close((uv_handle_t*) &async, NULL);
}
```

在这个展示了滥用 data 字段的例子之后，bnoordhuis 指出，data 字段的使用不是线程安全的，且 uv_async_send() 实际上只是用于唤醒事件循环。请使用互斥锁或读写锁来确保访问操作是以正确的顺序来执行的。

> **警告：** 互斥锁和读写 **不能** 在信号处理器中正确工作，却可以在 uv_async_send 中正常工作。

一个需要使用 uv_async_send 的例子是在与那些需要线程亲缘性来实现其功能的库进行互操作的时候。例如在 node.js 中，v8 引擎实例、上下文环境及其对象都绑定到了建立 v8 实例的线程上。在另一个线程中与 v8 数据结构交互会导致不确定的结果。现在，请考虑一些与第三方库绑定的 node.js 模块。它可能会像这样:

1. 在 node 中，第三方库随着 JavaScript 回调函数的调用以获取更多信息而被建立:

```
var lib = require('lib');
lib.on_progress(function() {
    console.log("Progress");
});
```

```
lib.do();

// 做其他事件
```

2. lib.do 应该是非阻塞的，但第三方库是阻塞的，所以使用 uv_queue_work 进行绑定。
3. 实际的工作会在想要调用进度回调函数的单独线程中完成，但不能直接在 v8 中调用 JavaScript 进行交互。所以它使用 uv_async_send。
4. 在主事件循环线程，也就是 v8 线程中调用的异步回调函数，于是与 v8 交互以调用 JavaScript 回调函数。

CHAPTER 6

进程

libuv 提供相当多的子进程管理功能，以对平台差异进行抽象及允许使用流和命名管道与子进程通信。

Unix 上的一个常见做法是每个进程只做一件事，并做好这事件。在这种情况下，进程常常使用多个子进程来执行任务（与在 shell 中使用管道相似）。对比多线程和共享内存的模型，用消息进行通信的多进程模型可能会更简单些。

基于事件的程序有一个共同的局限，就是不能利用现代计算机的多核优势。在多线程程序中，内核可以执行调度并将不同的线程分配到不同的核心，从而提高性能。可是，事件循环只有一个线程。而一个替代的办法是启动多进程，每个进程运行一个事件循环，并将各个进程分配到单独的 CPU 核心上。

6.1 建立子进程

最简单的情况是当你仅仅想启动一个进程并在其退出时得知其状态。使用 `uv_spawn` 来实现。

spawn/main.c

```
uv_loop_t *loop;
uv_process_t child_req;
uv_process_options_t options;

int main() {
    loop = uv_default_loop();

    char* args[3];
    args[0] = "mkdir";
    args[1] = "test-dir";
    args[2] = NULL;

    options.exit_cb = on_exit;
    options.file = "mkdir";
    options.args = args;

    if (uv_spawn(loop, &child_req, options)) {
        fprintf(stderr, "%s\n", uv_strerror(uv_last_error(loop)));
        return 1;
    }
```

```
21
22      return uv_run(loop, UV_RUN_DEFAULT);
23  }
```

注解: 因为 options 是全局变量，所以其中的元素会隐式被初始化为 0。如果你将 options 改为局部变量，记住将所有没有使用到的字段都初始化为 0:

```
uv_process_options_t options = {0};
```

---

uv_process_t 结构体只作为监视器，所有选项是通过 uv_process_options_t 设置。如果仅仅要启动进程，你只需要设置 file 和 args 字段。file 是要运行的程序。因为 uv_spawn 在内部使用了 execvp，所以不需要提供完整的路径。最后和每个底层约定一样， **参数数组要比参数的数目大一，最后一个元素为 NULL**。

在调用了 uv_spawn 之后，“uv_process_t.pid“将包含子进程的 PID。

退出回调函数被调用时会传入 退出状态 及导致程序退出的 信号 类型。

spawn/main.c

```
1  void on_exit(uv_process_t *req, int exit_status, int term_signal) {
2      fprintf(stderr, "Process exited with status %d, signal %d\n", exit_status, term_signal);
3      uv_close((uv_handle_t*) req, NULL);
4  }
```

在进程退出后， **需要** 关闭进程监视器。

6.2 修改进程参数

在子进程启动之前，你可以通过使用 uv_process_options_t 中的字段来控制运行环境。

6.2.1 修改运行目录

设置 uv_process_options_t.cwd 为相应的目录。

6.2.2 设置环境变量

uv_process_options_t.env 为以 null 结束的字符串数据，每个形如 VAR=VALUE 的元素用于设置进程的环境变量。将这个字段设置为 NULL 表明继承父（这个）进程的环境变量内容。

6.2.3 选项标志

设置 uv_process_options_t.flags 为下面的标志的按位或，以修改子进程的行为:

- UV_PROCESS_SETUID - 设置子进程的执行 UID 为 uv_process_options_t.uid。
- UV_PROCESS_SETGID - 设置子进程的执行 GID 为 uv_process_options_t.gid。

只有 Unix 才支持修改 UID/GID，在 Windows 平台上，uv_spawn 会因为 UV_ENOTSUP 错误而运行失败。

- UV_PROCESS_WINDOWS_VERBATIM_ARGUMENTS - 在 Windows 平台上生成的 uv_process_options_t.args 的不带引号或转义字符版本。此标志在 Unix 上被忽略。
- UV_PROCESS_DETACHED - 在新会话中的启动子进程，这将使子进程在父进程退出后继续运行下去。参见下面的例子。

6.3 分离进程

传入 UV_PROCESS_DETACHED 标志可用于启动守护进程，或不依赖于父进程的子进程，以使子进程不受父进程的退出影响。

detach/main.c

```
int main() {
    loop = uv_default_loop();

    char* args[3];
    args[0] = "sleep";
    args[1] = "100";
    args[2] = NULL;

    options.exit_cb = NULL;
    options.file = "sleep";
    options.args = args;
    options.flags = UV_PROCESS_DETACHED;

    if (uv_spawn(loop, &child_req, options)) {
        fprintf(stderr, "%s\n", uv_strerror(uv_last_error(loop)));
        return 1;
    }
    fprintf(stderr, "Launched sleep with PID %d\n", child_req.pid);
    uv_unref((uv_handle_t*) &child_req);

    return uv_run(loop, UV_RUN_DEFAULT);
}
```

只要记住，监视器依然监视着子进程，故你的程序不会退出。如果你想更加 射后不理（译者注：不监视子进程，即你的程序与子进程脱离任何关系） 就使用 uv_unref()。

6.4 向进程发送信号

libuv，在 Unix 上封闭了标准的 kill(2) 的系统调用，在 Windows 上也实现了类似的语义，只有 一个警告：SIGTERM、SIGINT 和 SIGKILL 都会导致进程的终止。uv_kill 的函数签名是:

```
uv_err_t uv_kill(int pid, int signum);
```

对于使用 libuv 启动的进程，你可能反而要使用 uv_process_kill，它接收 uv_process_t 监视器作为第一参数，而不是 pid。在这种情况下，**记住调用** uv_close 来关闭监视器。

6.5 信号

TODO: update based on https://github.com/joyent/libuv/issues/668

libuv 提供基于 Unix 信号的封装，同样也有 ome Windows 支持。

为了让信号与 libuv 很好地结合，这个 API 会将信号传递到 所有正在运行的事件循环的所有事件处理器中。使用 `uv_signal_init()` 初始化事件处理器并将它关联到事件循环。要监听某个事件处理器上特定的信号，向 `uv_signal_start()` 传入事件处理器函数。每个事件处理器只可以关联一个信号，`uv_signal_start()` 的后续调用会覆盖之前的关联。使用 `uv_signal_stop()` 来停止监视。这个小示例展示了各种可能性:

signal/main.c

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

void signal_handler(uv_signal_t *handle, int signum)
{
    printf("Signal received: %d\n", signum);
    uv_signal_stop(handle);
}

// two signal handlers in one loop
void thread1_worker(void *userp)
{
    uv_loop_t *loop1 = uv_loop_new();

    uv_signal_t sig1a, sig1b;
    uv_signal_init(loop1, &sig1a);
    uv_signal_start(&sig1a, signal_handler, SIGUSR1);

    uv_signal_init(loop1, &sig1b);
    uv_signal_start(&sig1b, signal_handler, SIGUSR1);

    uv_run(loop1, UV_RUN_DEFAULT);
}

// two signal handlers, each in its own loop
void thread2_worker(void *userp)
{
    uv_loop_t *loop2 = uv_loop_new();
    uv_loop_t *loop3 = uv_loop_new();

    uv_signal_t sig2;
    uv_signal_init(loop2, &sig2);
    uv_signal_start(&sig2, signal_handler, SIGUSR1);

    uv_signal_t sig3;
    uv_signal_init(loop3, &sig3);
    uv_signal_start(&sig3, signal_handler, SIGUSR1);

    while (uv_run(loop2, UV_RUN_NOWAIT) || uv_run(loop3, UV_RUN_NOWAIT)) {
    }
}

```

```
44 int main()
45 {
46     printf("PID %d\n", getpid());
47
48     uv_thread_t thread1, thread2;
49
50     uv_thread_create(&thread1, thread1_worker, 0);
51     uv_thread_create(&thread2, thread2_worker, 0);
52
53     uv_thread_join(&thread1);
54     uv_thread_join(&thread2);
55     return 0;
56 }
```

注解: uv_run(loop, UV_RUN_NOWAIT) 与 uv_run(loop, UV_RUN_ONCE) 类似，只会处理一个事件。如果没有待处理事件，UV_RUN_ONCE 会阻塞，而 UV_RUN_NOWAIT 则会立即返回。我们使用 NOWAIT 以使得一个事件循环不会因为另一个事件循环没有待处理活动而被空置。

向进程发送 SIGUSR1，你会发现事件处理器被调用了 4 次，每个 uv_signal_t 调用一次。事件处理器停止自身的处理，进而整个程序退出。对所有事件处理器采取这种调度方式非常有用。一个使用多重事件循环的服务器只要简单地向每个事件循环添加一个关联 SIGINT 信号的事件处理器就能够在其终止之前确保所有数据都已被安全地保存。

6.6 子进程 I/O

一个正常新建的进程有其自身的文件描述符集合，0、1、2 分别代表 stdin、stdout、stderr。有时你会想和子进程共享文件描述符。例如，你的应用程序可能会启动子命令，且你想将任何错误信息都记录到日志文件中，但要忽略 stdout。对此，你想显示子进程的 stderr。在这种情况下，libuv 支持 继承 文件描述符。在这个示例中，我们调用下面的测试程序:

proc-streams/test.c

```
#include <stdio.h>

int main()
{
    fprintf(stderr, "This is stderr\n");
    printf("This is stdout\n");
    return 0;
}
```

实际程序在只继承 stderr 的情况下运行上述程序。子进程的文件描述符通过 uv_process_options_t 的 stdio 字段设置。首先将 stdio_count 字段设置成要设置的文件描述符数目。uv_process_options_t.stdio 是 uv_stdio_container_t 结构体类型的数组，它的结构是:

```
typedef struct uv_stdio_container_s {
  uv_stdio_flags flags;

  union {
    uv_stream_t* stream;
    int fd;
```

```
  } data;
} uv_stdio_container_t;
```

其中的 flags 有多个值。如果不打算使用它，则设置为 UV_IGNORE。如果前三个 stdio 字段都标志为 UV_IGNORE，则它们会重定向到 /dev/null。

由于想传递现有的文件描述符，我们会使用 UV_INHERIT_FD。然后设置 fd 为 stderr。

proc-streams/main.c

```
int main() {
    loop = uv_default_loop();

    /* ... */

    options.stdio_count = 3;
    uv_stdio_container_t child_stdio[3];
    child_stdio[0].flags = UV_IGNORE;
    child_stdio[1].flags = UV_IGNORE;
    child_stdio[2].flags = UV_INHERIT_FD;
    child_stdio[2].data.fd = 2;
    options.stdio = child_stdio;

    options.exit_cb = on_exit;
    options.file = args[0];
    options.args = args;


    if (uv_spawn(loop, &child_req, options)) {
        fprintf(stderr, "%s\n", uv_strerror(uv_last_error(loop)));
        return 1;
    }

    return uv_run(loop, UV_RUN_DEFAULT);
}
```

若运行 proc-stream，你会发现只显示一行“This is stderr”。试试将 stdout 设置为继承的并查看程序输出。

在流上使用重定向是非常简单的。通过将 flags 设置成 UV_INHERIT_STREAM，再将 data.stream 设置成父进程的流，子进程便可以像操作标准 I/O 那样操作父进程的流了。这可被用于实现类似 CGI 的功能。

CGI 脚本/可执行程序的例子如下：

cgi/tick.c

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

int main() {
    int i;
    for (i = 0; i < 10; i++) {
        printf("tick\n");
        fflush(stdout);
        sleep(1);
    }
```

```
    printf("BOOM!\n");
    return 0;
}
```

CGI 服务器结合的本章及 连接网络 的概念，向每个客户端改送了 10 个 tick 后连接关闭。

cgi/main.c

```
void on_new_connection(uv_stream_t *server, int status) {
    uv_tcp_t *client = (uv_tcp_t*) malloc(sizeof(uv_tcp_t));
    uv_tcp_init(loop, client);
    if (uv_accept(server, (uv_stream_t*) client) == 0) {
        invoke_cgi_script(client);
    }
    else {
        uv_close((uv_handle_t*) client, NULL);
    }
}
```

这里我们简单地接受 TCP 连接并将套接字（流）传递给 `invoke_cgi_script`。

cgi/main.c

```
void invoke_cgi_script(uv_tcp_t *client) {

    /* ... finding the executable path and setting up arguments ... */

    options.stdio_count = 3;
    uv_stdio_container_t child_stdio[3];
    child_stdio[0].flags = UV_IGNORE;
    child_stdio[1].flags = UV_INHERIT_STREAM;
    child_stdio[1].data.stream = (uv_stream_t*) client;
    child_stdio[2].flags = UV_IGNORE;
    options.stdio = child_stdio;

    options.exit_cb = cleanup_handles;
    options.file = args[0];
    options.args = args;

    child_req.data = (void*) client;
    if (uv_spawn(loop, &child_req, options)) {
        fprintf(stderr, "%s\n", uv_strerror(uv_last_error(loop)));
        return;
    }
}
```

CGI 脚本的 `stdout` 重定向到套接字上，这样我们 tick 脚本中的输出内容都被发送到客户端。通过使用进程，我们可以将读/写缓存转移给操作系统，这样做非常便利。只是要警告的是创建进程是昂贵的任务。

6.7 管道

libuv 的 `uv_pipe_t` 结构体对于 Unix 程序员而言有些容易混淆，因为它很容易让人想起 | 和 pipe(7) 。然而 `uv_pipe_t` 与匿名管道无关，更确切地说，它有两个用法:

1. 流 API - 它作为 uv_stream_t 的具体实现以提供 FIFO，本地文件 I/O 的流接口。正如 缓冲区与流 提到的那样，这是通过使用 uv_pipe_open 来实现的。你还可以把它用于 TCP/UDP，不过已经有更方便的函数和结构体来使用它们了。
2. IPC 机制 - uv_pipt_t 可被 Unix Domain Socket 或 Windows Named Pipe 支持以允许多进程通信。这个在下面的内容中会讨论到。

6.7.1 父子进程 IPC

将 uv_stdio_container_t.flags 设置为 UV_CREATE_PIPE 和 UV_READABLE_PIPE 或 UV_WRITABLE_PIPE 的比特位组合，父进程和子进程可通过其建立的管道实现单或双工通信。读/写标志是对子进程而言的。

6.7.2 任意进程 IPC

因为域套接字 [1] 拥有周知的名称和文件系统的位置，所以它们可用于无关联进程间的 IPC。开源桌面环境的 D-BUS 系统将域套接字用于事件通知。不同的应用程序便可在接到在线联系或检测到新硬件时作出反应。MySQL 服务器也在其上运行用于与客户端交互的域套接字。

当使用域套接字时，客户端-服务器模式通常遵循服务器套接字作为创建者/持有者。在初始化设置之后，消息通信方法与 TCP 的没有区别，所以我们会再使用回应服务器的例子。

pipe-echo-server/main.c

```
int main() {
    loop = uv_default_loop();

    uv_pipe_t server;
    uv_pipe_init(loop, &server, 0);

    signal(SIGINT, remove_sock);

    if (uv_pipe_bind(&server, "echo.sock")) {
        fprintf(stderr, "Bind error %s\n", uv_err_name(uv_last_error(loop)));
        return 1;
    }
    if (uv_listen((uv_stream_t*) &server, 128, on_new_connection)) {
        fprintf(stderr, "Listen error %s\n", uv_err_name(uv_last_error(loop)));
        return 2;
    }
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

我们将套接字命名为 echo.sock，这意味着它将在本地目录中被创建。就流 API 而言，现在操作这个套接字的方法跟 TCP 套接字没区别。你可以使用 netcat 测试这个服务器:

```
$ nc -U /path/to/echo.sock
```

想连接到这个域套接字的客户端可使用:

```
void uv_pipe_connect(uv_connect_t *req, uv_pipe_t *handle, const char *name, uv_connect_cb cb);
```

其中的 name 可以是 echo.sock 或类似的名称。

[1] 在本节中的域套接字也代表 Windows 的命名管道。

6.7.3 通过管道发送文件描述符

域套接字最酷的地方在于文件描述符可以在进程之间交换，通过域套接字传递文件描述符的方法来实现。这允许进程将它们的 I/O 转交给其他进程。包含负载均衡服务器、工作进程和其他程序在内的应用就可以对 CPU 资源进行优化了。

> **警告：** 在 Windows 里，只能传递代表 TCP 套接字的文件描述符。

为了演示这个功能，我们会看看回应服务器的实现——工作进程以轮流的方式处理客户端请求。在本书中只涉及这个程序的一点说明及片段，建议阅读其完整的代码来真正理解它。

工作进程非简单，因为文件描述符已由主进程递交给工作进程。

multi-echo-server/worker.c

```
uv_loop_t *loop;
uv_pipe_t queue;


int main() {
    loop = uv_default_loop();

    uv_pipe_init(loop, &queue, 1);
    uv_pipe_open(&queue, 0);
    uv_read2_start((uv_stream_t*)&queue, alloc_buffer, on_new_connection);
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

queue 是在另一端连接到主进程的管道，新的文件描述符通过它发送过去。我人使用 read2 来监听文件描述符。务必要将 ipc 的 uv_pipe_init 参数设置成 1 以表明这个管道将被用于内部进程通信！因为主进程将文件句柄写到工作进程的标准输入流，我们使用 uv_pipe_open 将管道连接到 stdin。

multi-echo-server/worker.c

```
void on_new_connection(uv_pipe_t *q, ssize_t nread, uv_buf_t buf, uv_handle_type pending) {
    if (pending == UV_UNKNOWN_HANDLE) {
        // error!
        return;
    }

    uv_pipe_t *client = (uv_pipe_t*) malloc(sizeof(uv_pipe_t));
    uv_pipe_init(loop, client, 0);
    if (uv_accept((uv_stream_t*) q, (uv_stream_t*) client) == 0) {
        fprintf(stderr, "Worker %d: Accepted fd %d\n", getpid(), client->io_watcher.fd);
        uv_read_start((uv_stream_t*) client, alloc_buffer, echo_read);
    }
    else {
        uv_close((uv_handle_t*) client, NULL);
    }
}
```

虽然 accept 在代码里显得很怪，但它有实际意义。accept 本来是从一个文件描述符（侦听套接字）中获取另一个文件描述符（客户端）。这正是我们在这里所做的。从 queue 中取得文件描述符（client）。从这个角度来看，工作进程就是完成回应服务器做的事情。

现在转向主进程，让我们看看工作进程是如何启动以实现负载均衡。

multi-echo-server/main.c

```
uv_loop_t *loop;

struct child_worker {
    uv_process_t req;
    uv_process_options_t options;
    uv_pipe_t pipe;
} *workers;
```

child_worker 结构体封装了进程及主进程和独立进程之间的管道。

multi-echo-server/main.c

```
void setup_workers() {
    // ...

    // launch same number of workers as number of CPUs
    uv_cpu_info_t *info;
    int cpu_count;
    uv_cpu_info(&info, &cpu_count);
    uv_free_cpu_info(info, cpu_count);

    child_worker_count = cpu_count;

    workers = calloc(sizeof(struct child_worker), cpu_count);
    while (cpu_count--) {
        struct child_worker *worker = &workers[cpu_count];
        uv_pipe_init(loop, &worker->pipe, 1);

        uv_stdio_container_t child_stdio[3];
        child_stdio[0].flags = UV_CREATE_PIPE | UV_READABLE_PIPE;
        child_stdio[0].data.stream = (uv_stream_t*) &worker->pipe;
        child_stdio[1].flags = UV_IGNORE;
        child_stdio[2].flags = UV_INHERIT_FD;
        child_stdio[2].data.fd = 2;

        worker->options.stdio = child_stdio;
        worker->options.stdio_count = 3;

        worker->options.exit_cb = close_process_handle;
        worker->options.file = args[0];
        worker->options.args = args;

        uv_spawn(loop, &worker->req, worker->options);
        fprintf(stderr, "Started worker %d\n", worker->req.pid);
    }
}
```

在启动工作进程时，我们使用美妙的 libuv 函数 uv_cpu_info 来获取 CPU 的数目以便我们能够启动相同数目的工作进程。重复一遍，务必将用于 IPC 通道的管道的第三个参数设置为 1。然后，我们指定子进程的 stdin 为可读管道（就子进程而言）。直到现在还是一帆风顺。工作进程已启动，正等待文件描述符写入它们的管道。

就在 on_new_connection 函数里（TCP 的基础设施已在 main() 中初始化了），我们接受客户端套接字并轮流地传递给工作进程。

multi-echo-server/main.c

```
void on_new_connection(uv_stream_t *server, int status) {
    if (status == -1) {
        // error!
        return;
    }

    uv_pipe_t *client = (uv_pipe_t*) malloc(sizeof(uv_pipe_t));
    uv_pipe_init(loop, client, 0);
    if (uv_accept(server, (uv_stream_t*) client) == 0) {
        uv_write_t *write_req = (uv_write_t*) malloc(sizeof(uv_write_t));
        dummy_buf = uv_buf_init(".", 1);
        struct child_worker *worker = &workers[round_robin_counter];
        uv_write2(write_req, (uv_stream_t*) &worker->pipe, &dummy_buf, 1, (uv_stream_t*) client, NULL
        round_robin_counter = (round_robin_counter + 1) % child_worker_count;
    }
    else {
        uv_close((uv_handle_t*) client, NULL);
    }
}
```

重复一遍，uv_write2 调用处理所有抽象的事件，它只不过是将文件描述符作为正确的参数进行传递。有了这个，我们的多进程回应服务器就能够运作了。

TODO write2/read2 函数是怎么操作缓冲区?

CHAPTER 7

高级事件循环

libuv 提供相当多对事件循环的用户控制功能，且你可以从玩弄多重事件循环中得到有趣的结果。你可以将 libuv 的事件循环嵌入其他基于事件循环的库中 – 想像一下基于 UI 的 Qt，它的事件循环驱动 libuv 执行密集的系统级任务。

7.1 中止事件循环

`uv_stop()` 可用于中止事件循环。事件循环将 在下一次迭代时 停止。这意味着在本次事件循环迭代中已经准备好的事件将依然被处理，所以 `uv_stop()` 不可以用作"切断开关"（译者注：立即中止事件）。当调用了 `uv_stop()`，事件循环 **不会** 阻塞本次迭代的 I/O 操作。这些东西的语义有点难以理解，所以让我们看看当所有控制工作流都出现时的 `uv_run()`。

src/unix/core.c - uv_run

```
1   while (r != 0 && loop->stop_flag == 0) {
2     UV_TICK_START(loop, mode);
3
4     uv__update_time(loop);
5     uv__run_timers(loop);
6     uv__run_idle(loop);
7     uv__run_prepare(loop);
8     uv__run_pending(loop);
9
10    timeout = 0;
11    if ((mode & UV_RUN_NOWAIT) == 0)
12      timeout = uv_backend_timeout(loop);
```

`stop_flag` 由 `uv_stop()` 设置。目前，所有 libuv 的回调函数都在事件循环中调用，那就是为什么在回调函数中调用 `uv_stop()` 依然会导致本次事件循环的发生。libuv 先是更新计时器，然后运行待处理的定时器回调函数、空闲回调函数和准备回调函数，再调用待处理的 I/O 回调函数。如果你在它们其中任何一个里面调用 `uv_stop()`，`stop_flag` 将被设置。这使得 `uv_backend_timeout()` 返回 `0`，这就是为什么事件循环不在 I/O 上阻塞。另一方面，如果你在其中一个检查处理器中调用 `uv_stop()`，I/O 已经执行完毕并且不会受到影响。

当已经计算出一个结果或出现错误，`uv_stop()` 对关闭一个事件循环很有用，因为不用一个接一个地去确保所有的处理器是否已经停止。

这是一个中止事件循环并展示事件循环的本次迭代依然继续进行的简单示例。

uvstop/main.c

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

int64_t counter = 0;

void idle_cb(uv_idle_t *handle, int status) {
    printf("Idle callback\n");
    counter++;

    if (counter >= 5) {
        uv_stop(uv_default_loop());
        printf("uv_stop() called\n");
    }
}

void prep_cb(uv_prepare_t *handle, int status) {
    printf("Prep callback\n");
}

int main() {
    uv_idle_t idler;
    uv_prepare_t prep;

    uv_idle_init(uv_default_loop(), &idler);
    uv_idle_start(&idler, idle_cb);

    uv_prepare_init(uv_default_loop(), &prep);
    uv_prepare_start(&prep, prep_cb);

    uv_run(uv_default_loop(), UV_RUN_DEFAULT);

    return 0;
}
```

7.2 将 libuv 事件循环嵌入到其他库中

CHAPTER 8

实用工具

本章罗列了对常见任务有用的工具和技术。libev man page 已经涉及一些模式，这些模式通过简单地修改 API 就可用于 libuv。它还涉及部分不需要用整个章节专门讨论的 libuv API。

8.1 计时器

计时器会从启动到经过特定的时间后调用回调函数。libuv 计时器也可以被设置成周期性调用，而不是只调用一次。

简单的使用场景是初始化一个监视器并传入 `timeout` 和可选的 `repeat` 来启动它。计时器可在任何时刻停止。

```
uv_timer_t timer_req;

uv_timer_init(loop, &timer_req);
uv_timer_start(&timer_req, callback, 5000, 2000);
```

将启动一个重复执行的计时器，在调用了 `uv_timer_start` 之后，第一次启动 5 秒（即 `timeout`），然后每 2 秒（即 `repeat`）重复执行一次。用:

```
uv_timer_stop(&timer_req);
```

来停止计时器。这个函数也可以在回调函数中安全地使用。

重复执行的间隔可以在任何时刻被修改:

```
uv_timer_set_repeat(uv_timer_t *timer, int64_t repeat);
```

它会 **在可能的时候** 生效。如果这个函数在计时器的回调函数中调用，这意味着:

- 如果计时器不是重复执行的，就已经停止了。请再调用 `uv_timer_start`。
- 如果计时器是重复执行的，则下一个超时时间就已经被安排好了，所以在计时器切换到新的重复执行间隔前旧的重复执行间隔还会再使用一次。

工具函数:

```
int uv_timer_again(uv_timer_t *)
```

只能 用于 **重复执行的计时器**，相当于先中止计时器，将旧的 `repeat` 值设置成初始的 `timeout` 和 `repeat` 后启动计时器。如果计时器还没有启动，则它会调用失败（错误代码为 `UV_EINVAL`）并返回 -1。

在 引用计数部分 有个实际的例子。

8.2 事件循环引用计数

只要有活动的监视器，事件循环就会一直运行。系统运行时会在监视器启动时给事件循环引用计数加 1，而在监视器停止时给事件循环引用减 1。也可以手动修改处理器引用计数:

```
void uv_ref(uv_handle_t*);
void uv_unref(uv_handle_t*);
```

使用这些函数可让事件循环在监视器处于活动状态下退出，或让事件循环使用自定义对象来维持其活动状态。

前者可与间隔计时器使用。你可能会有一个每 X 秒运行一次的垃圾回收器，或你的网络服务可能要定期向别处发送活动信号，但你又不想随着所有清理退出通路的程序流程或错误场景的出现而停止它们。或你想在其他监视器都完成处理任务后程序才退出。在那种情况下，只要在紧接着计时器建立后就 unref 它以便即使只有监视器在运行，uv_run 依然会退出。

后者则用于 node.js 中一些暴露给 JS API 的 libuv 方法。uv_handle_t （所有监视器的超类）被每个 JS 对象创建且可被 ref/unref。

ref-timer/main.c

```
uv_loop_t *loop;
uv_timer_t gc_req;
uv_timer_t fake_job_req;

int main() {
    loop = uv_default_loop();

    uv_timer_init(loop, &gc_req);
    uv_unref((uv_handle_t*) &gc_req);

    uv_timer_start(&gc_req, gc, 0, 2000);

    // could actually be a TCP download or something
    uv_timer_init(loop, &fake_job_req);
    uv_timer_start(&fake_job_req, fake_job, 9000, 0);
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

我们初始化垃圾回收器的定时器，然后立即 unref 它。观察在 9 秒钟之后，当伪作业完成时，程序自动退出，尽管垃圾回收器依然在运行。

8.3 空闲监视器模式

空间监视器的回调函数只在事件循环没有其他待处理事件的时候被调用。在这种情况下，事件循环的每次遍历都会调用它们一次。例如，在空闲周期里你可以对日常应用的性能参数进行汇总以供开发者分析之用，或使用应用的 CPU 时间来执行 SETI 计算 :) 空闲监视器对 GUI 应用程序也有用。如你正使用事件循环来下载一个文件。如果 TCP 套接字正在建立当中而又没有其他事件出现，你的事件循环就会暂停（阻塞），也就意味着你的进度条会停滞不前且用户会以为应用程序已经崩溃了。在这种情况下，给事件循环加入空闲监视器以保持 UI 的可操作性。

idle-compute/main.c

```
uv_loop_t *loop;
uv_fs_t stdin_watcher;
uv_idle_t idler;
char buffer[1024];

int main() {
    loop = uv_default_loop();

    uv_idle_init(loop, &idler);

    uv_fs_read(loop, &stdin_watcher, 1, buffer, 1024, -1, on_type);
    uv_idle_start(&idler, crunch_away);
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

这里我们初始化空闲监视器并加入到我们感兴趣的事件中。 `crunch_away` 现在会重复地被调用直到用户输入内容并回车。然后在事件循环处理输入数据时它会短暂中断，之后它会再次不断地调用空闲回调函数。

idle-compute/main.c

```
void crunch_away(uv_idle_t* handle, int status) {
    // Compute extra-terrestrial life
    // fold proteins
    // computer another digit of PI
    // or similar
    fprintf(stderr, "Computing PI...\n");
    // just to avoid overwhelming your terminal emulator
    uv_idle_stop(handle);
}
```

8.4 向工作线程传递数据

在使用 `uv_queue_work` 时，你通常要给工作线程传入复杂的数据。解决方法是使用 `struct` 并将让 `uv_work_t.data` 指针指向它。一个小小的变型是把这个结构体的第一个成员设置成 `uv_work_t` 本身（称为 baton [1]）。这样只用一个 free 调用就可以清理工作线程请求及所有数据了。

```
struct ftp_baton {
    uv_work_t req;
    char *host;
    int port;
    char *username;
    char *password;
}
```

```
ftp_baton *baton = (ftp_baton*) malloc(sizeof(ftp_baton));
baton->req.data = (void*) baton;
baton->host = strdup("my.webhost.com");
baton->port = 21;
// ...
```

[1] mfp is My Fancy Plugin

```
uv_queue_work(loop, &baton->req, ftp_session, ftp_cleanup);
```

这里我们创建 baton 并将任务放入队列。

现在任务函数可以提取它需要的数据了：

```
void ftp_session(uv_work_t *req) {
    ftp_baton *baton = (ftp_baton*) req->data;

    fprintf(stderr, "Connecting to %s\n", baton->host);
}

void ftp_cleanup(uv_work_t *req) {
    ftp_baton *baton = (ftp_baton*) req->data;

    free(baton->host);
    // ...
    free(baton);
}
```

之后我们释放 baton 时，监视器也随之被释放了。

8.5 轮询外部 I/O

通常第三方库会处理它们自己的 I/O，也会跟踪它们内部的套接字和其他文件。这样它是不可能使用标准流 I/O 操作，但这个库还是可以整合到 libuv 事件循环。所需的全部操作是这个库允许你访问底层的文件描述符且提供处理如同你的应用设计那样小幅度任务的函数。不过一些库不会允许这样的访问，只提供一个标准的阻塞函数来执行全部的 I/O 事务然后仅仅返回结果。将这些库用于事件循环线程是不明智，用 *libuv* 工作队列 代替吧。当然这也意味着对这个库细粒度的控制。

libuv 的 `uv_poll` 部分仅仅是利用操作系统的通知机制来监听文件描述符。在某种程度上说，libuv 自己实现的所有 I/O 操作也被类似 `uv_poll` 的代码支持。无论操作系统什么时候发出正被轮询文件描述符的状态变化通知，libuv 都会调用与其关联的回调函数。

这里我们将过一下简单的下载管理器，会使用到 libcurl 来下载文件。不是让 libcurl 控制整个过程，而是会在 libuv 通知 I/O 可读时用 libuv 事件循环，以及非阻塞、异步 multi 接口来保持下载的进行。

uvwget/main.c - 设置

```
#include <stdio.h>
#include <stdlib.h>
#include <uv.h>
#include <curl/curl.h>

uv_loop_t *loop;
CURLM *curl_handle;
uv_timer_t timeout;

typedef struct curl_context_s {
    if (action == CURL_POLL_IN || action == CURL_POLL_OUT) {
        if (socketp) {
            curl_context = (curl_context_t*) socketp;
        }
        else {
```

```
16              curl_context = create_curl_context(s);
17          }
18          curl_multi_assign(curl_handle, s, (void *) curl_context);
19      }
20
21      switch (action) {
22          case CURL_POLL_IN:
23              uv_poll_start(&curl_context->poll_handle, UV_READABLE, curl_perform);
24              break;
25          case CURL_POLL_OUT:
26              uv_poll_start(&curl_context->poll_handle, UV_WRITABLE, curl_perform);
27              break;
28          case CURL_POLL_REMOVE:
29              if (socketp) {
30                  uv_poll_stop(&((curl_context_t*)socketp)->poll_handle);
31                  destroy_curl_context((curl_context_t*) socketp);
32                  curl_multi_assign(curl_handle, s, NULL);
33              }
34              break;
35          default:
36              abort();
37      }
38
39      return 0;
40  }
41
42  int main(int argc, char **argv) {
43      loop = uv_default_loop();
44
45      if (argc <= 1)
46          return 0;
47
48      if (curl_global_init(CURL_GLOBAL_ALL)) {
49          fprintf(stderr, "Could not init cURL\n");
50          return 1;
51      }
52
53      uv_timer_init(loop, &timeout);
54
55      curl_handle = curl_multi_init();
56      curl_multi_setopt(curl_handle, CURLMOPT_SOCKETFUNCTION, handle_socket);
57      curl_multi_setopt(curl_handle, CURLMOPT_TIMERFUNCTION, start_timeout);
58
59      while (argc-- > 1) {
60          add_download(argv[argc], argc);
61      }
62
63      uv_run(loop, UV_RUN_DEFAULT);
64      curl_multi_cleanup(curl_handle);
65      return 0;
66  }
```

每个库与 libuv 整合的方法都会不同。对于 libcurl，我们可以注册两个回调函数。在套接字状态改变而我们要开始轮询它的时候，套接字回调函数 `handle_socket` 会被调用。`start_timeout` 则由 libcurl 在下一个超时间隔的时间调用，之后无论 I/O 状态如何我们应该让 libcurl 继续工作。这样以便 libcurl 能够处理错误或其他使下载继续进行的事情。

我们下载器将被这样调用:

```
$ ./uvwget [url1] [url2] ...
```

像这样我们添加每个作为 URL 的参数

uvwget/main.c - 添加 URL

```
void add_download(const char *url, int num) {
    char filename[50];
    sprintf(filename, "%d.download", num);
    FILE *file;

    file = fopen(filename, "w");
    if (file == NULL) {
        fprintf(stderr, "Error opening %s\n", filename);
        return;
    }

    CURL *handle = curl_easy_init();
    curl_easy_setopt(handle, CURLOPT_WRITEDATA, file);
    curl_easy_setopt(handle, CURLOPT_URL, url);
    curl_multi_add_handle(curl_handle, handle);
    fprintf(stderr, "Added download %s -> %s\n", url, filename);
}
```

我们让 libcurl 直接把数据写入文件中，但你也可以做更多你想做的事件。

`start_timeout` 会在首次就立即被 libcurl 调用，故东西一开始就设定好了。这只是启动一个在其超时的时候以 `CURL_SOCKET_TIMEOUT` 驱动 `curl_multi_socket_action` 的 libuv timer 。`curl_multi_socket_action` 就是驱动 libcurl的，且我们在套接字状态改变时调用它。但在进入这个函数之前，我们需要在调用 `handle_socket` 时轮询套接字。

uvwget/main.c - 建立轮询

```
int handle_socket(CURL *easy, curl_socket_t s, int action, void *userp, void *socketp) {
    curl_context_t *curl_context;
    if (action == CURL_POLL_IN || action == CURL_POLL_OUT) {
        if (socketp) {
            curl_context = (curl_context_t*) socketp;
        }
        else {
            curl_context = create_curl_context(s);
        }
        curl_multi_assign(curl_handle, s, (void *) curl_context);
    }

    switch (action) {
        case CURL_POLL_IN:
            uv_poll_start(&curl_context->poll_handle, UV_READABLE, curl_perform);
            break;
        case CURL_POLL_OUT:
            uv_poll_start(&curl_context->poll_handle, UV_WRITABLE, curl_perform);
            break;
        case CURL_POLL_REMOVE:
            if (socketp) {
                uv_poll_stop(&((curl_context_t*)socketp)->poll_handle);
```

```
                destroy_curl_context((curl_context_t*) socketp);
                curl_multi_assign(curl_handle, s, NULL);
            }
            break;
        default:
            abort();
    }

    return 0;
}
```

我们对套接字文件描述符 `s` 及其 `action` 感兴趣。对于每个套接字，如果它不存在，我们就创建一个 `uv_poll_t` 句柄，并用 `curl_multi_assign` 关联到套接字上。这样在回调函数被调用时，`socketp` 指针就指向了它。

在下载完成或失败的情况下，libcurl 请求移除轮询。所以我们停止并释放了 poll 句柄。

根据 libcurl 要监听的事件，我们对 `UV_READABLE` 和 `UV_WRITABLE` 启动轮询。现在 libuv 将在套接字已经准备好读或写的时候调用 poll 回调函数。在同一个句柄上可以多次调用 `uv_poll_start`，它只会将事件掩码更新成新的值。`curl_perform` 才是本程序的关键。

uvwget/main.c - 建立轮询

```
void curl_perform(uv_poll_t *req, int status, int events) {
    uv_timer_stop(&timeout);
    int running_handles;
    int flags = 0;
    if (events & UV_READABLE) flags |= CURL_CSELECT_IN;
    if (events & UV_WRITABLE) flags |= CURL_CSELECT_OUT;

    curl_context_t *context;

    context = (curl_context_t*)req;

    curl_multi_socket_action(curl_handle, context->sockfd, flags, &running_handles);

    char *done_url;

    CURLMsg *message;
    int pending;
    while ((message = curl_multi_info_read(curl_handle, &pending))) {
        switch (message->msg) {
            case CURLMSG_DONE:
                curl_easy_getinfo(message->easy_handle, CURLINFO_EFFECTIVE_URL, &done_url);
                printf("%s DONE\n", done_url);

                curl_multi_remove_handle(curl_handle, message->easy_handle);
                curl_easy_cleanup(message->easy_handle);

                break;
            default:
                fprintf(stderr, "CURLMSG default\n");
                abort();
        }
    }
}
```

我们做的第一件事是停止计时数，因为在这期间已经有一些进展。然后根据触发回调函数的事件，我们照样去通知 libcurl。接着我们调用 `curl_multi_socket_action`，传入已经开始处理的套接字及说明什么事件发生的标志。此时，libcurl 就会在内部自行完成细粒度的任务，并尝试尽可能快地返回。libcurl 不断地向其自身的队列里输入传输进度的消息。在我们的例子中，我们只对传输完成感兴趣。所以我们提取出这些消息，并将已经完成传输的句柄清理掉。

8.6 检查和准备监视器

TODO

8.7 加载库

libuv 提供用于动态加载 动态库 的跨平台 API。它可用于实现你自己的插件/扩展/模块系统，且已经在 node.js 实现用于支持库绑定的 `require()`。用法非学简单，只要你的库导出正常的符号即可。在加载第三方库时要注意完好性和安全性检查，否则你的程序会出现异常行为。这个示例实现了一个非常简单的插件系统，它仅仅输出了插件的名称。

让我们先来看看它提供给插件作者的接口吧。

plugin/plugin.h

```
#ifndef UVBOOK_PLUGIN_SYSTEM
#define UVBOOK_PLUGIN_SYSTEM

void mfp_register(const char *name);

#endif
```

plugin/plugin.c

```
#include <stdio.h>

void mfp_register(const char *name) {
    fprintf(stderr, "Registered plugin \"%s\"\n", name);
}
```

同样，你可以增加更多函数来让插件作者能够在你的应用程序中做有用的事情 [2] 。一个使用了这个 API 的救命插件:

plugin/hello.c

```
#include "plugin.h"

void initialize() {
    mfp_register("Hello World!");
}
```

[2] 我首次在 Konstantin Käfer 关于编写 node.js 绑定的优秀幻灯片中接触到 baton 这个词 – http://kkaefer.github.com/node-cpp-modules/#baton

我们的接口定义了所有插件应该要有一个会被应用程序调用的 initialize 函数。这个插件编译为动态库并能够由我们的应用程序在运行的时候加载:

```
$ ./plugin libhello.dylib
Loading libhello.dylib
Registered plugin "Hello World!"
```

这是使用 uv_dlopen 先加载动态库来完成的。然而我们用 uv_dlsym 来访问 initialize 函数并调用它。

plugin/main.c

```
#include "plugin.h"

typedef void (*init_plugin_function)();

int main(int argc, char **argv) {
    if (argc == 1) {
        fprintf(stderr, "Usage: %s [plugin1] [plugin2] ...\n", argv[0]);
        return 0;
    }

    uv_lib_t *lib = (uv_lib_t*) malloc(sizeof(uv_lib_t));
    while (--argc) {
        fprintf(stderr, "Loading %s\n", argv[argc]);
        if (uv_dlopen(argv[argc], lib)) {
            fprintf(stderr, "Error: %s\n", uv_dlerror(lib));
            continue;
        }

        init_plugin_function init_plugin;
        if (uv_dlsym(lib, "initialize", (void **) &init_plugin)) {
            fprintf(stderr, "dlsym error: %s\n", uv_dlerror(lib));
            continue;
        }

        init_plugin();
    }

    return 0;
}
```

uv_dlopen 需要动态库的路径，并设置不透明指针（译者注：句柄）“uv_lib_t“。成功时返回 0，错误时返回 -1。使用 uv_dlerror 来获取错误消息。

uv_dlsym 将第二个参数里的符号存储到第三个参数中的指针上。init_plugin_function 是指向应用程序插件中我们所需要的那个函数的函数指针。

8.8 TTY

文本终端已以一个 相当规范的 的命令集合维持了基本格式一段很长的时间了。这种格式通常用在程序里以改善终端输出的可读性。例如 grep --colour。libuv 提供 uv_tty_t 抽象（流）及相关的函数来实现跨所有平台的 ANSI 转义代码。通过这个，我想让 libuv 将 ANSI 编码转换成等价的 Windows 编译，并提供获取终端信息的函数。

首先用 uv_tty_t 需要读/写的文件描述符来初始化它。这个用:

```
int uv_tty_init(uv_loop_t*, uv_tty_t*, uv_file fd, int readable)
```

实现。

如果 `readable` 为 false，`uv_write` 调用这个流就会被 **阻塞**。

然后最好使用 `uv_tty_set_mode` 设置模式为 正常 (0)，这样会启用大多数 TTY 格式、流控制和其他设置。原始 模式 (1) 也是支持的。

记住在你的程序退出时调用 `uv_tty_reset_mode` 来恢复终端状态。这仅仅是出于礼貌的做法。如果用户将你的命令输出重定向到文件，控制字符不应该写入文件，因为它们妨碍可读性和 `grep`。要检查文件描述符是否为 TTY，调用 `uv_guess_handle`，传入文件描述符并比较返回值和 `UV_TTY`。

这是一个在红色背景中输出白色文本的简单示例：

tty/main.c

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <uv.h>

uv_loop_t *loop;
uv_tty_t tty;
int main() {
    loop = uv_default_loop();

    uv_tty_init(loop, &tty, 1, 0);
    uv_tty_set_mode(&tty, 0);

    if (uv_guess_handle(1) == UV_TTY) {
        uv_write_t req;
        uv_buf_t buf;
        buf.base = "\033[41;37m";
        buf.len = strlen(buf.base);
        uv_write(&req, (uv_stream_t*) &tty, &buf, 1, NULL);
    }

    uv_write_t req;
    uv_buf_t buf;
    buf.base = "Hello TTY\n";
    buf.len = strlen(buf.base);
    uv_write(&req, (uv_stream_t*) &tty, &buf, 1, NULL);
    uv_tty_reset_mode();
    return uv_run(loop, UV_RUN_DEFAULT);
}
```

最后一个 TTY 辅助函数是 `uv_tty_get_winsize()`，它是用来获取终端的宽度和高度的，成功时返回 0。这个小程序通过函数和字符位置转义编码实现一些动画。

tty-gravity/main.c

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <uv.h>
```

```
5
6   uv_loop_t *loop;
7   uv_tty_t tty;
8   uv_timer_t tick;
9   uv_write_t write_req;
10  int width, height;
11  int pos = 0;
12  char *message = "  Hello TTY  ";
13
14  void update(uv_timer_t *req, int status) {
15      char data[500];
16
17      uv_buf_t buf;
18      buf.base = data;
19      buf.len = sprintf(data, "\033[2J\033[H\033[%dB\033[%luC\033[42;37m%s",
20                                pos,
21                                (unsigned long) (width-strlen(message))/2,
22                                message);
23      uv_write(&write_req, (uv_stream_t*) &tty, &buf, 1, NULL);
24
25      pos++;
26      if (pos > height) {
27          uv_tty_reset_mode();
28          uv_timer_stop(&tick);
29      }
30  }
31
32  int main() {
33      loop = uv_default_loop();
34
35      uv_tty_init(loop, &tty, 1, 0);
36      uv_tty_set_mode(&tty, 0);
37
38      if (uv_tty_get_winsize(&tty, &width, &height)) {
39          fprintf(stderr, "Could not get TTY information\n");
40          uv_tty_reset_mode();
41          return 1;
42      }
43
44      fprintf(stderr, "Width %d, height %d\n", width, height);
45      uv_timer_init(loop, &tick);
46      uv_timer_start(&tick, update, 200, 200);
47      return uv_run(loop, UV_RUN_DEFAULT);
48  }
```

转义编码:

编码	含义
2 J	清除屏幕的某个部分，2 为整个屏幕
H	移动光标到指定的位置，默认为左上角
n B	光标向下移动 n 行
n C	光标向右移动 n 列
m	按显示设置输出字符串，这里为绿色 (40+2)，白色文本 (30+7)

正如你看到的那样，这个功能非常有用，可以生成美观的格式化输出，或甚至制作满足你的奇思妙想的电脑游戏。对于更加发烧级的控制，你可以度试 ncurses 。

CHAPTER 9

关于

Nikhil Marathe 在一个不想写程序的下午开始编写这本书。他最近致力于 node-taglib 项目上，但苦于没有一份好的 libuv 文档。虽然已经有参考文档，却没有一份易于理解的教程。本书也由此而生，并致力于其准确性。这就是说，Nikhil 还年轻且经验尚浅，可能会犯些严重的错误。如果你发现什么错误的地方，他希望你能 联系他 。你可以通过 fork 这个 GitHub 代码仓库 并发送 pull 请求来为本书作出贡献。

Nikhil 感谢 Marc Lehmann 编写了易于理解的 libev man page ，其中描述很多关于这两个库的语义。

本书使用 Sphinx 和 vim 制作。

9.1 许可协议

CHAPTER 10

其他格式

本书还提供以下格式：