3.3 GCD 实现

3.3.1 Dispatch Queue

GCD 的 Dispatch Queue 非常方便，那么它究竟是如何实现的呢？

- 用于管理追加的 Block 的 C 语言层实现的 FIFO 队列
- Atomic 函数中实现的用于排他控制的轻量级信号
- 用于管理线程的 C 语言层实现的一些容器

不难想象，GCD 的实现需要使用以上这些工具。但是，如果仅用这些内容便可实现，那么就不需要内核级的实现了[①]。

甚至有人会想，只要努力编写线程管理的代码，就根本用不到 GCD。真的是这样吗？

我们先来回顾一下苹果的官方说明。

通常，应用程序中编写的线程管理用的代码要在系统级实现。

实际上正如这句话所说，在系统级即 iOS 和 OS X 的核心 XNU 内核级上实现。

因此，无论编程人员如何努力编写管理线程的代码，在性能方面也不可能胜过 XNU 内核级所实现的 GCD。

使用 GCD 要比使用 pthreads 和 NSThread 这些一般的多线程编程 API 更好。并且，如果使用 GCD 就不必编写为操作线程反复出现的类似的源代码（这被称为固定源代码片断），而可以在线程中集中实现处理内容，真的是好处多多。我们尽量多使用 GCD 或者使用了 Cocoa 框架 GCD 的 NSOperationQueue 类等 API。

那么首先确认一下用于实现 Dispatch Queue 而使用的软件组件。如表 3-4 所示。

表 3-4 用于实现 Dispatch Queue 而使用的软件组件

组件名称	提供技术
libdispatch	Dispatch Queue
Libc（pthreads）	pthread_workqueue
XNU 内核	workqueue

编程人员所使用 GCD 的 API 全部为包含在 libdispatch 库中的 C 语言函数。Dispatch Queue 通过结构体和链表，被实现为 FIFO 队列。FIFO 队列管理是通过 dispatch_async 等函数所追加的 Block。

---

① 实际上在一般的 Linux 内核中可能使用面向 Linux 操作系统而移植的 GCD。
Portable libdispatch　　https://www.heily.com/trac/libdispatch。

Block 并不是直接加入 FIFO 队列，而是先加入 Dispatch Continuation 这一 dispatch_continuation_t 类型结构体中，然后再加入 FIFO 队列。该 Dispatch Continuation 用于记忆 Block 所属的 Dispatch Group 和其他一些信息，相当于一般常说的执行上下文。

Dispatch Queue 可通过 dispatch_set_target_queue 函数设定，可以设定执行该 Dispatch Queue 处理的 Dispatch Queue 为目标。该目标可像串珠子一样，设定多个连接在一起的 Dispatch Queue。但是在连接串的最后必须设定为 Main Dispatch Queue，或各种优先级的 Global Dispatch Queue，或是准备用于 Serial Dispatch Queue 的各种优先级的 Global Dispatch Queue。

Main Dispatch Queue 在 RunLoop 中执行 Block。这并不是令人耳目一新的技术。

Global Dispatch Queue 有如下 8 种。

- Global Dispatch Queue（High Priority）
- Global Dispatch Queue（Default Priority）
- Global Dispatch Queue（Low Priority）
- Global Dispatch Queue（Background Priority）
- Global Dispatch Queue（High Overcommit Priority）
- Global Dispatch Queue（Default Overcommit Priority）
- Global Dispatch Queue（Low Overcommit Priority）
- Global Dispatch Queue（Background Overcommit Priority）

优先级中附有 Overcommit 的 Global Dispatch Queue 使用在 Serial Dispatch Queue 中。如 Overcommit 这个名称所示，不管系统状态如何，都会强制生成线程的 Dispatch Queue。

这 8 种 Global Dispatch Queue 各使用 1 个 pthread_workqueue。GCD 初始化时，使用 pthread_workqueue_create_np 函数生成 pthread_workqueue。

pthread_workqueue 包含在 Libc 提供的 pthreads API 中。其使用 bsdthread_register 和 workq_open 系统调用，在初始化 XNU 内核的 workqueue 之后获取 workqueue 信息。

XNU 内核持有 4 种 workqueue。

- WORKQUEUE_HIGH_PRIOQUEUE
- WORKQUEUE_DEFAULT_PRIOQUEUE
- WORKQUEUE_LOW_PRIOQUEUE
- WORKQUEUE_BG_PRIOQUEUE

以上为 4 种执行优先级的 workqueue。该执行优先级与 Global Dispatch Queue 的 4 种执行优先级相同。

下面看一下 Dispatch Queue 中执行 Block 的过程。当在 Global Dispatch Queue 中执行 Block 时，libdispatch 从 Global Dispatch Queue 自身的 FIFO 队列中取出 Dispatch Continuation，调用 pthread_workqueue_additem_np 函数。将该 Global Dispatch Queue 自身、符合其优先级的 workqueue 信息以及为执行 Dispatch Continuation 的回调函数等传递给参数。

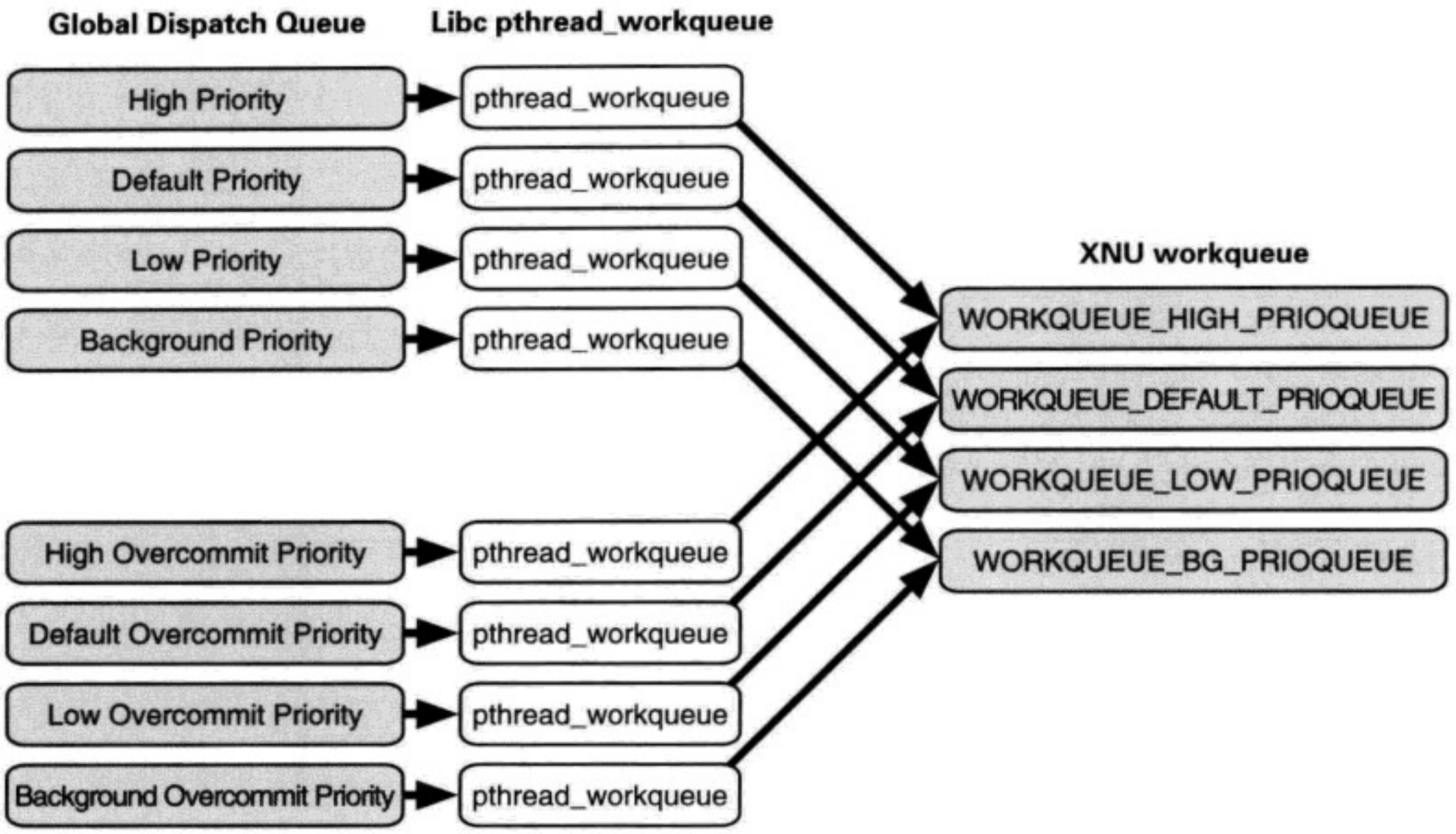

图 3-16　Global Dispatch Queue 与 pthread_workqueue、workqueue 的关系

pthread_workqueue_additem_np 函数使用 workq_kernreturn 系统调用，通知 workqueue 增加应当执行的项目。根据该通知，XNU 内核基于系统状态判断是否要生成线程。如果是 Overcommit 优先级的 Global Dispatch Queue，workqueue 则始终生成线程。

该线程虽然与 iOS 和 OS X 中通常使用的线程大致相同，但是有一部分 pthread API 不能使用。详细信息可参考苹果的官方文档《并列编程指南》的“与 POSIX 线程的互换性”一节。

另外，因为 workqueue 生成的线程在实现用于 workqueue 的线程计划表中运行，所以与一般线程的上下文切换不同。这里也隐藏着使用 GCD 的原因。

workqueue 的线程执行 pthread_workqueue 函数，该函数调用 libdispatch 的回调函数。在该回调函数中执行加入到 Dispatch Continuation 的 Block。

Block 执行结束后，进行通知 Dispatch Group 结束、释放 Dispatch Continuation 等处理，开始准备执行加入到 Global Dispatch Queue 中的下一个 Block。

以上就是 Dispatch Queue 执行的大概过程。

由此可知，在编程人员管理的线程中，想发挥出匹敌 GCD 的性能是不可能的。

3.3.2 Dispatch Source

GCD 中除了主要的 Dispatch Queue 外，还有不太引人注目的 Dispatch Source。它是 BSD 系内核惯有功能 kqueue 的包装。

kqueue 是在 XNU 内核中发生各种事件时，在应用程序编程方执行处理的技术。其 CPU 负

荷非常小，尽量不占用资源。kqueue 可以说是应用程序处理 XNU 内核中发生的各种事件的方法中最优秀的一种。

Dispatch Source 可处理以下事件。如表 3-5 所示。

表 3-5　Dispatch Source 的种类

名称	内容
DISPATCH_SOURCE_TYPE_DATA_ADD	变量增加
DISPATCH_SOURCE_TYPE_DATA_OR	变量 OR
DISPATCH_SOURCE_TYPE_MACH_SEND	MACH 端口发送
DISPATCH_SOURCE_TYPE_MACH_RECV	MACH 端口接收
DISPATCH_SOURCE_TYPE_PROC	检测到与进程相关的事件
DISPATCH_SOURCE_TYPE_READ	可读取文件映像
DISPATCH_SOURCE_TYPE_SIGNAL	接收信号
DISPATCH_SOURCE_TYPE_TIMER	定时器
DISPATCH_SOURCE_TYPE_VNODE	文件系统有变更
DISPATCH_SOURCE_TYPE_WRITE	可写入文件映像

事件发生时，在指定的 Dispatch Queue 中可执行事件的处理。

下面我们使用 DISPATCH_SOURCE_TYPE_READ，异步读取文件映像。

```
__block size_t total = 0;
size_t size = 要读取的字节数
char *buff = (char *)malloc(size);

/*
 * 设定为异步映像
 */
fcntl(sockfd, F_SETFL, O_NONBLOCK);

/*
 * 获取用于追加事件处理的 Global Dispatch Queue
 */
dispatch_queue_t queue =
    dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

/*
 * 基于 READ 事件作成 Dispatch Source
 */
dispatch_source_t source =
    dispatch_source_create(DISPATCH_SOURCE_TYPE_READ, sockfd, 0, queue);

/*
 * 指定发生 READ 事件时执行的处理
 */
dispatch_source_set_event_handler(source, ^{
    /*
     * 获取可读取的字节数
     */
```

```
size_t available = dispatch_source_get_data(source);

/*
 * 从映像中读取
 */
int length = read(sockfd, buff, available);

/*
 * 发生错误时取消 Dispatch Source
 */
if(length < 0){
    /*
     * 错误处理
     */

    dispatch_source_cancel(source);
}

total += length;

if(total == size){

    /*
     * buff 的处理
     */

    /*
     * 处理结束，取消 Dispatch Source
     */
    dispatch_source_cancel(source);
}});

/*
 * 指定取消 Dispatch Source 时的处理
 */
dispatch_source_set_cancel_handler(source, ^{
    free(buff);
    close(sockfd);

    /*
     * 释放 Dispatch Source（自身）
     */
    dispatch_release(source);
});

/*
 * 启动 Dispatch Source
 */
dispatch_resume(source);
```

与上面源代码非常相似的代码，使用在了 Core Foundation 框架的用于异步网络的 API CFSocket 中。因为 Foundation 框架的异步网络 API 是通过 CFSocket 实现的，所以可享受到仅使

用 Foundation 框架的 Dispatch Source（即 GCD）带来的好处。

最后给大家展示一个使用了 DISPATCH_SOURCE_TYPE_TIMER 的定时器的例子。在网络编程的通信超时等情况下可使用该例。

```
/*
 * 指定 DISPATCH_SOURCE_TYPE_TIMER，作成 Dispatch Source。
 *
 * 在定时器经过指定时间时设定 Main Dispatch Queue 为追加处理的 Dispatch Queue
 */
dispatch_source_t timer = dispatch_source_create(
    DISPATCH_SOURCE_TYPE_TIMER, 0, 0, dispatch_get_main_queue());

/*
 * 将定时器设定为 15 秒后。
 * 不指定为重复。
 * 允许迟延 1 秒。
 */
dispatch_source_set_timer(timer,
    dispatch_time(DISPATCH_TIME_NOW, 15ull * NSEC_PER_SEC),
        DISPATCH_TIME_FOREVER, 1ull * NSEC_PER_SEC);

/*
 * 指定定时器指定时间内执行的处理
 */
dispatch_source_set_event_handler(timer, ^{
    NSLog(@"wakeup!");

    /*
     * 取消 Dispatch Source
     */
    dispatch_source_cancel(timer);
});

/*
 * 指定取消 Dispatch Source 时的处理
 */
dispatch_source_set_cancel_handler(timer, ^{
    NSLog(@"canceled");

    /*
     * 释放 Dispatch Source（自身）
     */
    dispatch_release(timer);
});

/*
 * 启动 Dispatch Source
 */
dispatch_resume(timer);
```

看了异步读取文件映像用的源代码和这个定时器用的源代码后，有没有注意到什么呢？实际上 Dispatch Queue 没有“取消”这一概念。一旦将处理追加到 Dispatch Queue 中，就没有方法可

将该处理去除，也没有方法可在执行中取消该处理。编程人员要么在处理中导入取消这一概念，要么放弃取消，或者使用 NSOperationQueue 等其他方法。

Dispatch Source 与 Dispatch Queue 不同，是可以取消的。而且取消时必须执行的处理可指定为回调用的 Block 形式。因此使用 Dispatch Source 实现 XNU 内核中发生的事件处理要比直接使用 kqueue 实现更为简单。在必须使用 kqueue 的情况下希望大家还是使用 Dispatch Source，它比较简单。

通过讲解，大家应该已经理解了主要的 Dispatch Queue 以及次要的 Dispatch Source 了吧。