並列プログラミングガイド

Developer

目次

表、リスト

概要

同時並行性とは、複数の事柄が同時に発生していることを表しています。マルチコアCPUが普及し、プロセッサあたりのコア数も増えてきたため、ソフトウェア開発者も、その利点を活かす新たな手段を取り入れる必要が出てきました。OS XやiOSなどのオペレーティングシステムは、複数のプログラムを並列実行できますが、多くのプログラムはバックグラウンドで動作し、連続したプロセッサ時間を要することはほとんどありません。ユーザの目に性能の違いがはっきり現れ、CPU時間を消費するのは、その時点でフォアグラウンドで動作しているアプリケーションだけなのです。あるアプリケーションが膨大な量の処理をするとしても、一部のコアしか使っていないとすれば、残りは無駄になってしまいます。

従来、アプリケーションに並列性を取り入れようとすれば、スレッドをいくつか生成する必要がありました。残念ながら、スレッド処理を伴うプログラムの開発は非常に困難です。スレッドは低レベルの機能であって、その管理には手間がかかります。アプリケーションの動作に最適なスレッド数は、システム負荷やハードウェアの状況に応じて動的に変化しうるので、スレッド処理の最適な実装は、不可能とまでは言わないにしても、非常に難しいのです。さらに、多くの場合、スレッド間の同期機構が必要で、これもソフトウェアの設計が複雑でリスクの高いものになる要因です。苦労して実装しても、性能を改善できるという保証はありません。

OS XやiOSは、並列タスクの実行に、スレッドベースのシステムやアプリケーションに従来よく見られたものに比べて、より非同期的な方針を取り入れています。アプリケーションは、スレッドを直接生成するのではなく、タスクを定義し、システムにその実行を委ねるだけです。スレッド管理をシステムに委ねることにより、拡張性も格段に改善されました。じかにスレッドを管理していたのではとうてい不可能だったでしょう。アプリケーション開発者は、より簡潔で効率的なプログラミングモデルをも手にしました。

このドキュメントでは、アプリケーションに並列処理を実装する際に必要な、技法や技術について説明します。この技術は、OS XとiOSのどちらにも適用可能です。

この書類の構成

このドキュメントは、次の章で構成されています。

- “並列性とアプリケーション設計” （9 ページ）では、非同期アプリケーション設計の基礎と、カスタムタスクを非同期的に実行する技術の概要を示します。

- “オペレーションキュー”（17 ページ）では、Objective-Cのオブジェクトを使って、タスクをカプセル化し、実行する方法を説明します。
- “ディスパッチキュー”（39 ページ）では、Cベースのアプリケーションで、タスクを並列実行する方法を紹介します。
- “ディスパッチソース”（57 ページ）では、システムイベントを非同期的に処理する方法を解説します。
- “スレッドを使ったコードからの移行”（75 ページ）では、スレッドベースの既存のコードを、より新しい技術に移行するためのヒントと技法を示します。

以上のほかに、「用語解説」を載せてあります。

用語に関する注記

並列性について解説する前に、混乱を避けるため、関連する用語をいくつか定義しておく必要があるでしょう。UNIXシステムや旧OS Xの技術に通じた開発者から見ると、このドキュメントでは、「タスク」、「プロセス」、「スレッド」という用語が、若干違った意味で使われているはずです。このドキュメントでは次のような意味で使います。

- **スレッド**とは、コードを実行する、他とは切り離されたパスのことです。OS Xでは、POSIXのスレッドAPIに準拠して実装されています。
- **プロセス**とは、動作中の実行形式コードのことで、複数のスレッドから成ることもあります。
- **タスク**とは、実行するべき処理を表す、抽象的な概念です。

このドキュメントで使われるほかの重要な用語を含め、きちんとした定義は“用語解説”（86 ページ）を参照してください。

関連項目

ここでは、アプリケーションに並列性を実装する、より望ましい技術を中心に解説します。スレッドの使いかたには触れません。スレッドの使いかたや、ほかのスレッドに関する技術については、『*Threading Programming Guide*』を参照してください。

並列性とアプリケーション設計

コンピュータの黎明期には、実行可能な単位時間あたりの最大作業量は、CPUのクロック速度で決まっていました。しかし、技術が進展し、プロセッサが小型化するにつれ、熱その他の物理的な制約により、クロック速度が抑えられるようになってきました。そこでチップ製造会社は、総合的な性能を高める別の手段を追求し始めました。そして到達したのが、チップあたりのプロセッサコア数を増やす、という解決策だったのです。その結果、CPU速度、あるいはチップサイズや熱特性はそのままで、毎秒の実行命令数を格段に改善できました。残った問題は、増えたコアから性能を引き出すにはどうすればよいか、ということです。

複数のコアが組み込まれているという優位性を活かすためには、複数の処理を同時に実行するソフトウェアが必要です。OS XやiOSのような最新のマルチタスクOSでは、数百ものプログラムを同時に動かせるので、各プログラムを別々のコアに割り振るようスケジューリングすることも可能ではあります。しかし、このプログラムの多くはシステムデーモンやバックグラウンドアプリケーションであって、実処理時間はほとんど消費しません。実際に必要とされるのは、個々のアプリケーションが、増えたコアをより効率的に使えるようにする手段なのです。

アプリケーションがマルチコアを扱う従来の方式は、相応の数のスレッドを生成する、というものでした。しかしコア数が増えると、この方式ではさまざまな問題が生じます。最大の問題は、スレッドを使って実装したコードが、コア数に応じて拡張しにくい、ということです。単にコアと同数のスレッドを生成したからといって、プログラムが適切に動作するとは期待できません。実質的に使用可能なコア数を調べる必要があるのですが、アプリケーションが独自にそれを計算するのは非常に困難です。何とか適正な数がわかったとしても、それだけの数のスレッドを効率的に動かし、互いに干渉しないようプログラムするのが、難しいことに変わりはありません。

以上のような問題があるので、コア数の違いに応じ、その利点をアプリケーションが活かせるようにする手段が必要です。個々のアプリケーションの処理量を、システム状況に応じて動的に変える手段も必要です。さらに、その手段は、開発者の作業がいたずらに増えないよう、できるだけ単純なものでなければなりません。幸い、Appleのオペレーティングシステムには、以上のような問題に対処する手段が組み込まれています。この章では、この手段を実現した技術と、マルチコアの能力を最大限引き出せるようなアプリケーションの設計法について解説します。

スレッド方式の問題点とその解決

スレッド方式は長年にわたって利用されてきましたが、複数のタスクを実行し、しかも拡張性を確保したい、という課題を解決するものではありません。拡張性の高いアプリケーションを実装できるかどうかは、開発者の腕次第、というところがあります。スレッドをいくつ生成するか判断し、システムの状況に応じて動的に増減しなければなりません。さらに、スレッドの生成や維持にかなりのコストがかかる、という問題もあります。

OS XやiOSでは、スレッドに代え、**非同期設計のアプローチ**を採用して並列性の問題を解決しています。非同期で動作する関数自体は、古くからオペレーティングシステムに組み込まれ、ディスクからのデータ読み込みなど、処理時間を要するタスクによく使われています。非同期の関数は、呼び出されると、裏で若干の処理をした後タスクを開始しますが、その完了を待たずに、呼び出し元に制御を返してしまいます。裏で行う処理には、バックグラウンドスレッドを獲得すること、そのスレッド上でタスクを起動すること、タスク完了時に（コールバック関数などで）呼び出し元に通知すること、があります。これまで、必要な機能を備えた非同期関数がなければ、スレッドを生成してその処理を行う非同期関数を、独自に開発する必要がありました。しかしOS XやiOSは、スレッドを自分で管理しなくても、あらゆるタスクを非同期に実行できる技術を備えています。

タスクを非同期に実行する技術のひとつとして、**Grand Central Dispatch（GCD）**というものがあります。通常はアプリケーション中に記述するスレッド管理用のコードを、システムレベルで実装したものです。開発者がしなければならないのは、実行したいタスクを定義し、適切なディスパッチキューに追加することだけです。するとGCDは、必要なスレッドを生成し、そこでタスクを実行するよう適切にスケジューリングします。スレッド管理がシステムの一部として組み込まれているので、全体を見渡してタスクを管理、実行でき、従来のスレッドよりも効率が向上します。

オペレーションキューとは、ディスパッチキューとほとんど同じように振る舞う、Objective-Cのオブジェクトのことです。実行したいタスクを定義してオペレーションキューに追加することにより、スケジューリングや実行を委ねることができます。GCDと同様、オペレーションキューもスレッド管理に必要な処理をすべて行うので、システムが許す限り、迅速かつ効率的にタスクを実行できます。

以下の各セクションでは、ディスパッチキューやオペレーションキューについてさらに詳しく解説し、アプリケーションで利用できるほかの非同期技術についても説明します。

ディスパッチキュー

ディスパッチキューは、カスタムタスクを実行するための、Cベースの機構です。**ディスパッチキュー**は、タスクを直列にも並列にも実行できますが、その順序は常に先入れ先出し方式で決まります（すなわち、キューに追加されたのと同じ順序でタスクを取り出し、実行します）。直列ディスパッチキューの場合、同時に1つのタスクだけを実行します。あるタスクが完了してから、次のタスクを取り出して起動することになります。これに対し並列ディスパッチキューは、タスクの完了を待たずに、できるだけ多くのタスクを起動しようとします。

ディスパッチキューには、ほかにも次のような長所があります。

- 単純でわかりやすいプログラミングインターフェイスを提供します。
- スレッドプールを自動的に、システム全体を見渡して管理できます。
- アセンブリレベルで処理速度を調整しています。
- メモリ効率に優れています（スレッドスタックはアプリケーションメモリ中に長く残らないため）。
- カーネルに過剰な負荷がかかりません。
- タスクをディスパッチキューに非同期に追加しても、デッドロックが発生することはありません。
- 競合が起これば、徐々にスレッドを増減して調整します。
- 直列ディスパッチキューは、ロックその他の同期プリミティブの、より効率的な代替手段として使えます。

ディスパッチキューに登録するタスクは、関数またはブロックオブジェクトの形でカプセル化しなければなりません。**ブロックオブジェクト**とは、OS X v10.6およびiOS 4.0で導入されたC言語レベルの機能で、概念的には関数ポインタに似ていますが、さまざまな利点があります。ブロックそれ自体には（関数と違って）変数のレキシカルスコープを定める働きがなく、関数やメソッドの内部にブロックを定義するようになっているため、ブロック内から、当該関数やメソッドのほかの変数にもアクセスできます。ブロックをそのまま、元のスコープから切り離し、ヒープにコピーすることもできます。ブロックをディスパッチキューに登録する場合、内部的にはこのような処理をしています。このような特徴があるので、非常に動的なタスクを、比較的少ないコード量で実装できるのです。

ディスパッチキューはGrand Central Dispatch技術の一部を成し、Cの実行時ライブラリに組み込まれています。アプリケーションでディスパッチキューを利用する手順については、“ディスパッチキュー”（39 ページ）を参照してください。また、ブロックおよびその長所については、『*Blocks Programming Topics*』を参照してください。

ディスパッチソース

ディスパッチソースとは、特定の型のシステムイベントを非同期的に処理する、Cベースの機構のことです。該当する型のシステムイベントに関する情報をカプセル化し、そのイベントが発生したとき、対応するブロックオブジェクトまたは関数をディスパッチキューに登録する働きがあります。ディスパッチソースで監視できるシステムイベントは次のようなものです。

- タイマー
- シグナルハンドラ
- 記述子に関するイベント

- プロセスに関するイベント
- Machポートイベント
- カスタムイベント（トリガ処理をアプリケーションに組み込んだもの）

ディスパッチソースはGrand Central Dispatch技術の一部を成すものです。アプリケーション中で、ディスパッチソースを使ってイベントを受信する方法については、“ディスパッチソース”（57 ページ）を参照してください。

オペレーションキュー

オペレーションキューとは、並列ディスパッチキューと同等の機能を持つCocoaの機構で、`NSOperationQueue`クラスをとして実装されています。ディスパッチキューがタスクを先入れ先出し順に実行するのに対し、オペレーションキューはほかの因子も考慮して実行順序を決めます。この因子の中でも重要なのは、あるタスクが完了してからでないと実行できない、という関係です。この依存関係は、タスクを定義する際に設定します。実行順序を有向グラフで表すと、かなり複雑なものになるかもしれません。

オペレーションキューに登録するタスクは、`NSOperation`クラスのインスタンスでなければなりません。**オペレーションオブジェクト**とは、Objective-Cのオブジェクトであって、実行したい処理と、それに必要なデータをカプセル化したものです。`NSOperation`は抽象基底クラスなので、通常は、タスクごとにサブクラスを派生させて定義します。もっとも、Foundationフレームワークにはいくつか具象サブクラスが定義されているので、そのインスタンスを生成してタスクを実行することも可能です。

オペレーションオブジェクトは、KVO（Key-Value Observing）通知というものを生成します。これはタスクの進捗状況を監視する手段として有用です。オペレーションキューは常にオペレーションを並列実行しますが、必要ならば依存関係を適切に設定することにより、直列に実行させることも可能です。

オペレーションキューの使いかた、独自のオペレーションオブジェクトの定義方法については、“オペレーションキュー”（17 ページ）を参照してください。

非同期設計に関する技法

並列性を取り入れてコードを設計し直す前に、それが本当に必要なのか、を検討してください。並列性を組み込めば、メインスレッドがいつでもユーザイベントに応答できるので、コードの応答性が増します。また、複数のコアが連携することにより、性能が向上するという効果もあります。その一方で、オーバーヘッドが増え、コード全体の複雑性も増すため、開発やデバッグの負担も大きくなります。

並列性はコードの複雑さを増す要因でもあるので、開発サイクルが進んでから後付けで組み込めるものではありません。アプリケーションが実行するタスクと、そのために用いるデータ構造を、慎重に検討する必要があります。ここで間違うと、処理速度や応答性が、かえって損なわれるおそれがあります。したがって、設計サイクルの初期にある程度の時間をとって、目標を定め、方針を検討する価値はあるでしょう。

アプリケーションによって、求められる要件や、実行するべきタスクは異なります。具体的なアプリケーションや関連するタスクの設計方法について、このドキュメントで説明することはできません。以下のセクションでは、その代わりに、設計を進める上で役立つであろう指針をいくつか示します。

アプリケーションに期待される振る舞いの定義

並列性について検討する前に、アプリケーションに対して想定する望ましい振る舞いを、きちんと定義することから始めるべきでしょう。これは、設計が適切であったかどうか、後で検証するための手がかりになります。また、並列性を取り入れたとき、どの程度の性能向上を期待するか、を検討する材料にもなります。

まずしなければならないのは、アプリケーションが実行するべきタスクを列挙し、各タスクに関係するオブジェクトやデータの構成を決めることです。おそらく最初は、メニュー項目を選択したりボタンを押したりしたときに実行されるタスクを考えるでしょう。それぞれのタスクに、振る舞いや、実行前と実行後の状態を、明確に規定します。このほか、タイマーベースのタスクなど、ユーザの操作を待たずに実行されるタスクも挙げる必要があります。

高レベルのタスクを列挙した後、各タスクの遂行に必要な、個々の手順に分割していきます。このレベルでは、データ構造やオブジェクトに対してどのような操作を施すか、その結果、アプリケーション全体の状態にどのような影響があるか、といった事項を主として検討します。また、オブジェクトやデータ構造どうしの依存関係についても考えなければなりません。たとえば、一連のオブジェクトに対して同じ変更を施す場合、あるオブジェクトに対する変更がほかに影響するか、に注目して検討する価値があるでしょう。それぞれ独立に変更できるのであれば、並列処理が可能かどうか検討する余地があります。

実行可能な処理単位の抽出

アプリケーションのタスクを把握すれば、並列性を取り入れることにより効果が期待できる箇所を特定できるはずです。タスクを構成するステップの順序を変更すると結果が変わってしまうのであれば、おそらくその各ステップは、これまでどおり、直列に実行しなければならないでしょう。逆に結果に影響しないのであれば、並列処理を検討するべきです。いずれの場合も、実行するべきステップを表す、実行可能な処理単位を定義することになります。この処理単位を、ブロックまたはオペレーションオブジェクトの形でカプセル化し、適切なキューにディスパッチします。

ここで挙げた、実行可能な処理単位の数が多くなってしまっても、少なくとも設計の初期段階では、あまり心配する必要はありません。スレッドの生成には必ずコストがかかりますが、ディスパッチキューやオペレーションキューの利点のひとつに、多くの場合、従来型のスレッドよりもコストが格段に小さい、ということがあります。したがって、スレッドの代わりにキューを使えば、処理単位が小さくても、より効率的に実行できます。もちろん、実際の処理性能を必ず測定し、必要ならばタスクの大きさを調整するべきですが、初期段階では、小さすぎて性能が損なわれるのではないか、などと考える必要はありません。

キューの使い分けについて

タスクを個々の処理単位に分割し、ブロックオブジェクトまたはオペレーションオブジェクトにカプセル化したら、次に、当該コードを実行するために使うキューを定義しなければなりません。タスクごとに、作成したブロックやオペレーションオブジェクトをどのような順序で実行すれば、当該タスクが正常に処理されるか検討してください。

タスクをブロックの形で実装した場合、このブロックは、直列、並列いずれのディスパッチキューにも追加できます。特定の順序で実行する必要がある場合は、直列ディスパッチキューに追加してください。そうでない場合は、必要に応じて、並列ディスパッチキューに追加するか、またはディスパッチキューをいくつか用意して使い分けることになります。

一方、オペレーションオブジェクトの形でタスクを実装した場合は、キューの選択よりも、オブジェクトの設定の方が重要です。オペレーションオブジェクトを直列に実行するためには、関係するオブジェクト間の依存関係を設定しなければなりません。依存関係には、あるオペレーションの実行を、それが依存するオブジェクトの処理完了まで抑制する、という効果があります。

効率向上のためのヒント

単にコードをより小さなタスクに分割してキューに追加すること以外にも、キューを使って全体的な効率を改善する手段がいくつかあります。

- **メモリ消費量が問題になる場合は、タスク内で直接、値を計算することを検討してください。**アプリケーションが既に限界までメモリを使っていれば、値を直接計算する方が、あらかじめ計算してメインメモリ中に保存しておいた値を読み込むよりも、高速になる可能性があります。値を直接計算する際には、あるプロセッサコアのレジスタやキャッシュを使うことになりますが、これはメインメモリよりもはるかに高速だからです。もちろん、処理性能を実測した上で、こちらの方が優れている、と判明してから採用してください。
- **設計の早い段階で、直列に実行しているタスクを特定し、並列実行できないかどうか検討してください。**直列に実行している原因が、何らかの共有リソースの都合であるならば、共有リソースを使わなくて済むよう、アーキテクチャを変更するとよいかもしれません。そのリソースを必要とするクライアントごとにコピーを保持する、あるいはリソースそのものがなくても処理できるようにすることを検討するとよいでしょう。

- **ロックの使用を避けてください。** ディスパッチキューやオペレーションキューを利用すれば、ロックが必要になる状況はほとんどありません。共有リソースを競合から保護するには、ロックではなく、正しい順序でタスクを実行できるよう、専用の直列キューを利用する（またはオペレーションオブジェクトの依存関係を設定する）とよいでしょう。
- **できるだけシステムフレームワークを利用してください。** 並列性を活かすためには、システムフレームワークが提供する、組み込みの並列実行機構を活用するのが最善です。多くのフレームワークでは、スレッドその他の技術を内部的に利用して、並列性を実装しています。タスクを定義する際には、やりたいことを並列実行する関数やメソッドが、既存のフレームワークに最初から定義されていないか調べてみるとよいでしょう。このようなAPIを使えば、開発の手間が省け、並列性を最大限活用できます。

処理性能について

オペレーションキュー、ディスパッチキュー、ディスパッチソースは、コードを並列実行しやすくするために提供されています。しかしこの技術により、アプリケーションの効率や応答性が改善される、という保証はありません。必要な効率を確保し、アプリケーションのほかのリソースに過度の負担を与えないようにキューを用いるのは、開発者の責任です。たとえば、1万個のオペレーションオブジェクトを生成してオペレーションキューに登録することも不可能ではありませんが、少なからぬ量のメモリを確保することになるため、ページングが多発して処理性能を損なうおそれがあります。

コードに並列性を取り入れる（キューを使うにせよスレッドによるにせよ）際には、処理性能の改善の度合いを比較するためにも、現状を測定し、基準として残しておく必要があります。変更を施した後、改めて測定し、全体的な効率に改善が見られるかどうか、基準と比較してください。並列性を取り入れたことによりかえって効率や応答性が損なわれる場合は、性能評価ツールを使うなどして、原因を調べるべきでしょう。

処理性能や性能評価ツール、その他、性能関連の話題については、『*Performance Overview*』を参照してください。

並列性およびその他の技術

コードを小さなタスクに分割する方法は、アプリケーションに並列性を取り入れ、改善していく上で、非常に優れています。しかしこの設計方針によっても、あらゆるアプリケーションの要求を満足できるとは限りません。タスクの内容にもよりますが、アプリケーションの全体的な並列性をさらに改善できる、ほかの手段があるかもしれません。このセクションでは、設計の一環として検討するべき、ほかの技術について概説します。

OpenCLと並列性

OS Xでは、**Open Computing Language（OpenCL）**という標準技術が、グラフィックス処理の汎用的な計算に使われています。大量のデータセットに対し、明確に定義された計算を施す場合、OpenCLは優れた技術と言えるでしょう。たとえば、画像のピクセル単位でフィルタ計算を施す、多数の数値に対して同時に複雑な数学計算をする、などといった状況です。すなわち、大量のデータに対して並列に処理を施せる場合に、OpenCLは優れた問題解決手段を提供できるのです。

OpenCLは、大量のデータに対して並列処理を施す場合には優れていますが、より汎用的な計算処理には向いていません。データと必要な作業カーネルの両方を準備してグラフィックスカードに転送し、GPUで処理できるようにするためには、かなりの処理を要するでしょう。同様に、OpenCLの処理結果を取ってくるための処理も膨大です。したがって、システムとのやり取りが必要なタスクには、一般に、OpenCLを使うことはお勧めしません。たとえば、ファイルやネットワークストリームからデータを取得して処理する目的には、OpenCLは向いていないのです。むしろ、グラフィックプロセッサへの転送と計算とが独立に実行できる、自己完結した処理に向いています。

OpenCLおよびその使いかたについては、『*OpenCL Programming Guide for Mac*』を参照してください。

スレッドが向いている状況

オペレーションキューやディスパッチキューは、タスクを並列実行する手段として優れていますが、万能というわけではありません。アプリケーションの内容によっては、独自のスレッドを生成するべき状況もあります。その場合でも、スレッド数はできるだけ少なく抑え、ほかの手段では実装できないタスクにのみ用いてください。

実時間処理を要するコードの実装手段としては、今なおスレッドが優れています。ディスパッチキューは、タスクをできるだけ迅速に実行しようとしますが、実時間で処理しなければならない、という制約を満たすことはできません。バックグラウンドで動作するコードについて、事前の振る舞いを精度よく予測したい場合にも、スレッドは向いています。

スレッドプログラミングでは常にそうであるように、スレッドは慎重に、どうしても必要な箇所に限定して使うようにしてください。スレッドパッケージやその使いかたについては、『*Threading Programming Guide*』を参照してください。

オペレーションキュー

Cocoaでいうオペレーション（操作）とは、非同期に実行するべき処理を、オブジェクト指向の方式でカプセル化する方法のことです。オペレーションは、オペレーションキューと組み合わせても、それ単独でも実行できる設計になっています。オペレーションはObjective-Cベースなので、OS XやiOSの、Cocoaベースのアプリケーションに、広く使われています。

この章では、オペレーションを定義し、使用する方法について説明します。

オペレーションオブジェクトについて

オペレーションオブジェクトは、（Foundationフレームワークにおける）`NSOperation`クラスのインスタンスで、アプリケーションで実行したい処理をカプセル化するために使います。`NSOperation`自身は抽象基底クラスなので、そのサブクラスに、実際の処理内容を記述していくことになります。抽象クラスではありますが、かなりの基盤機能が組み込まれているので、サブクラス側に記述するコード量は最小限で済みます。さらに、Foundationフレームワークには、既存のコードと組み合わせてそのまま使える具象サブクラスが2つ用意されています。表2-1に、そのクラスと、簡単な使いかたを示します。

表 2-1 Foundationフレームワークに付属するオペレーションクラス

クラス	Description
`NSInvocation-Operation`	オブジェクトおよびセレクタをもとにして、アプリケーションからオペレーションオブジェクトを生成するために、そのままの状態で使えるクラスです。このクラスは、必要なタスクを実行するメソッドが、既に開発済みである場合に使います。サブクラスを派生させる必要がないので、より動的なやり方でオペレーションオブジェクトを生成するためにも使えます。 このクラスの使いかたについては、"NSInvocationOperation（起動オペレーション）オブジェクトの生成"（19ページ）を参照してください。

クラス	Description
`NSBlockOperation`	ブロックオブジェクトを並列実行するために、そのままの状態で使えるクラスです。複数のブロックを実行できるので、ブロックオペレーションオブジェクトはグループとして振る舞います。すなわち、関係するブロックすべての実行が完了した時点で、初めてオペレーションは完了した、と見なします。 このクラスの使いかたについては、"NSBlockOperation（ブロックオペレーション）オブジェクトの生成"（20ページ）を参照してください。なお、このクラスが使えるのはOS X v10.6以降です。ブロックについては、『*Blocks Programming Topics*』を参照してください。
`NSOperation`	独自のオペレーションオブジェクトを定義するための基底クラスです。`NSOperation`のサブクラスには、実行方法、状態の通知方法などをオーバーライドして、どのようにでも独自のオペレーションを実装できます。 独自のオペレーションオブジェクトの定義方法については、"独自のオペレーションオブジェクトの定義"（21ページ）を参照してください。

オペレーションオブジェクトはすべて、次のような機能を備えています。

- オペレーションオブジェクト間の依存関係を、有向グラフの形で設定すること。これにより、依存するオペレーションがすべて完了してから実行を開始する、という制御が可能です。依存関係の設定については、"オペレーション間の依存関係の設定"（29ページ）を参照してください。
- 完了ブロックの機能。オペレーションの主たるタスク完了後に実行するブロックです（OS X v10.6以降のみ）。完了ブロックの設定方法については、"完了ブロックのセットアップ"（31ページ）を参照してください。
- KVO通知を監視して、オペレーションの実行状況の変化を把握するための機能。KVO通知を監視する方法については、『*Key-Value Observing Programming Guide*』を参照してください。
- オペレーションの優先度を制御して、相対的な実行順序を調整する機能。詳細については、"オペレーションの実行優先度の変更"（30ページ）を参照してください。
- 実行中にオペレーションをキャンセルする機構。オペレーションをキャンセルする方法については、"オペレーションのキャンセル"（37ページ）を参照してください。また、独自のオペレーションにキャンセル機能を組み込む方法については、"キャンセルイベントへの応答"（23ページ）を参照してください。

オペレーションは、アプリケーションの並列性レベル改善に役立つよう設計されています。さらに、アプリケーションの振る舞いを、明確に区分した単純な「チャンク」に編成し、カプセル化する手段としても優れています。アプリケーションのコードの一部を、メインスレッド上で動かす代わりに、オペレーションオブジェクトをいくつかキューに登録し、対応する処理を、個々のスレッド上で非同期に実行することも可能です。

並列オペレーションと非並列オペレーション

オペレーションは通常、オペレーションキューに追加して実行しますが、この方法しかないわけではありません。オペレーションオブジェクトの`start`メソッドを呼び出す、という形で、直接的に実行することも可能です。もっとも、コードのほかの部分と並列に動作することは保証できません。`NSOperation`クラスの`isConcurrent`メソッドで、`start`メソッドの呼び出し元スレッドと、同期して動作しているかどうか調べることができます。デフォルトの戻り値は`NO`で、これは呼び出し元スレッドと同期して動作していることを表します。

並列オペレーションを実装する、すなわち、呼び出し元スレッドと非同期に動作させるためには、オペレーションを非同期に起動するコードを記述しなければなりません。たとえば、独立したスレッドを生成する、非同期システム関数を呼び出すなど、何らかの方法で、タスクを起動して直ちに（普通はタスク完了前に）制御を戻すというコードを、`start`メソッドに実装するのです。

多くの開発者にとって、並列オペレーションオブジェクトを独自に実装しなければならないことは、ほとんどないでしょう。オペレーションをオペレーションキューに追加する、という方法でいつでも実行しているならば、並列オペレーションを自分で実装する必要はありません。非並列オペレーションをオペレーションキューに登録すると、キューは自分自身でスレッドを生成し、オペレーションを実行します。このように、非並列オペレーションをオペレーションキューに追加する、という方法でも、オペレーションオブジェクトのコードを非同期に実行できるのです。並列オペレーションの定義が必要となるのは、オペレーションキューに追加することなく、非同期にオペレーションを実行したい場合だけです。

並列オペレーションを生成する方法については、"オペレーションを並列実行するための設定"（24ページ）および『*NSOperation Class Reference*』を参照してください。

NSInvocationOperation（起動オペレーション）オブジェクトの生成

`NSInvocationOperation`は、`NSOperation`の具象サブクラスで、指定したオブジェクトの、指定したセレクタを起動する、という機能があります。アプリケーションのタスクごとに、独自のオペレーションオブジェクトをいくつも定義しなくてよい、という効果があります。特に、既存のアプリケー

ションを改訂する場合や、必要なタスクを実行するオブジェクトやメソッドが実装済みである場合に有用です。また、呼び出すべきメソッドが状況に応じて変化する、という場合にも便利です。たとえば、ユーザの入力に応じて、動的に選んだセレクタを実行する、ということが可能です。

起動オペレーションを生成する手順はごく簡単です。このクラスのインスタンスを生成、初期化し、その際、実行するオブジェクトおよびセレクタを渡すだけです。リスト 2-1に、生成処理をコード例を示します。`taskWithData:`メソッドでは、新しい起動オペレーションを生成します。その際、実質的にタスクを実装しているメソッドの名前を渡しています。

リスト 2-1　`NSInvocationOperation`オブジェクトの生成

```
@implementation MyCustomClass
- (NSOperation*)taskWithData:(id)data {
    NSInvocationOperation* theOp = [[NSInvocationOperation alloc] initWithTarget:self
                      selector:@selector(myTaskMethod:) object:data];

    return theOp;
}

// タスクの実質的な処理を行うメソッド
- (void)myTaskMethod:(id)data {
    // タスクを実行
}
@end
```

NSBlockOperation（ブロックオペレーション）オブジェクトの生成

`NSBlockOperation`は`NSOperation`の具象サブクラスで、いくつかのブロックオブジェクトをまとめたラッパとして振る舞います。アプリケーションで既にオペレーションキューを使っており、さらにディスパッチキューも用意するのは望ましくない、という場合に、オブジェクト指向ラッパとしての機能を提供します。また、ブロックオペレーションを使うと、依存関係の管理、KVO通知など、ディスパッチキューにはないさまざまな機能も利用できます。

ブロックオペレーションを生成する場合、通常、初期化の際に少なくとも1つのブロックを追加します。必要ならば、後でさらにブロックを追加することも可能です。`NSBlockOperation`オブジェクトは、実行する段階になると、ブロックをすべて、デフォルトの優先度を設定してオペレーションキュー

に登録します。こうしておいて、ブロックの実行がすべて終わるまで待機します。最後のブロックの実行が終わると、オペレーションオブジェクトは自分自身に、完了した旨の印をつけます。このようにブロックオペレーションでは、実行する複数のブロックを、グループとしてまとめて管理できます。スレッドの合流操作により、複数のスレッドの結果をマージするのと同様の操作と言えるでしょう。もっとも、ブロックオペレーションはそれ自身が独立したスレッド上で動作するので、アプリケーションのほかのスレッドは、ブロックオペレーションの完了を待つことなく、自分自身の処理を続行できる、という点が異なります。

リスト 2-2「NSBlockOperation」に、NSBlockOperationのオブジェクトを生成する簡単なコード例を示します。ブロック自身には引数がなく、戻り値を返すということもありません。

リスト 2-2 NSBlockOperationオブジェクトの生成

```
NSBlockOperation* theOp = [NSBlockOperation blockOperationWithBlock: ^{
      NSLog(@"Beginning operation.\n");
      // 何らかの処理。
   }];
```

ブロックオペレーションのオブジェクトには、生成後も、addExecutionBlock:メソッドでさらにブロックを追加できます。ブロックを直列に実行したい場合は、適切なディスパッチキューに、直接登録してください。

独自のオペレーションオブジェクトの定義

ブロックオペレーションや起動オペレーションでは、アプリケーションの要求に対応できないのであれば、NSOperationから直接サブクラスを派生させ、必要な動作を実装してください。NSOperationクラスは、どんなオペレーションオブジェクトでも実装できるよう、コードのさまざまな箇所が、サブクラスでオーバーライドできるようになっています。さらに、依存関係やKVO通知の管理に必要な、かなりの量の基盤機能が組み込まれています。それでも、オペレーションが意図通りに振る舞うようにするためには、ある程度のコードを記述しなければならないでしょう。自分で実装しなければならないコードの量は、非並列オペレーションか並列オペレーションかによっても違います。

非並列オペレーションの定義方法は、並列オペレーションに比べて単純です。しなければならないのは、主たるタスクを実行すること、キャンセルイベントに適切に応答することだけです。それ以外は基盤機能として最初から組み込まれています。一方、並列オペレーションの場合は、基盤機能のいくつかを、独自のコードで置き換える必要があります。以下のセクションでは、この両方のオブジェクトについて、実装方法を説明します。

主たるタスクの実行

オペレーションオブジェクトには、最低限、次のメソッドを実装しなければなりません。

- 独自の初期化メソッド
- `main`

独自の初期化メソッドは、オペレーションオブジェクトを所定の状態にするため、`main`メソッドはタスクを実行するために必要です。以下のメソッドは、必要に応じて実装してください。

- `main`メソッドから呼び出されてタスクを実行する、独自のメソッド
- データ値を設定し、またはオペレーションの結果を取得するためのアクセスメソッド
- `NSCoding`プロトコルを実装して、オペレーションオブジェクトのアーカイブ保存またはその解除を行うメソッド

リスト 2-3に、`NSOperation`のサブクラスを記述する際の開始用テンプレートを示します（キャンセルイベントに応答する方法は省略し、通常用意するべきメソッドを示します。キャンセル処理については"キャンセルイベントへの応答" （23 ページ）を参照してください）。初期化メソッドは引数としてデータオブジェクトを1つ取り、その参照をオペレーションオブジェクト内に保持します。`main`メソッドは、上記のデータオブジェクトに対して何らかの処理を施し、結果をアプリケーションに返します。

リスト 2-3　単純なオペレーションオブジェクトの定義

```
@interface MyNonConcurrentOperation : NSOperation
@property id (strong) myData;
-(id)initWithData:(id)data;
@end

@implementation MyNonConcurrentOperation
- (id)initWithData:(id)data {
   if (self = [super init])
      myData = data;
   return self;
}

-(void)main {
   @try {
```

```
        // myDataに対して何らかの処理を施し、結果を返す。
    }
    @catch(...) {
        // 例外を再スローしない。
    }
}
@end
```

NSOperationのサブクラスを実装する詳しい例は、『*NSOperationSample*』を参照してください。

キャンセルイベントへの応答

オペレーションの実行を開始すると、完了するか、明示的にキャンセルされるまでの間、タスクの実行が続きます。キャンセルはいつでも（実行開始の前であっても）起こりえます。NSOperationクラスには、クライアントがオペレーションをキャンセルするための手段がありますが、キャンセルイベントを認識しても、直ちに応じなければならないわけではありません。オペレーションを即座に停止してしまうと、確保したリソースを回収できなくなってしまう場合があります。そのためオペレーションオブジェクトは、キャンセルイベントを確認し、オペレーション中であれば、必要な後処理をしてから中断しなければなりません。

キャンセルイベントに対処するため、定期的にisCancelledメソッドを呼び出して、その戻り値がYESであれば直ちに中断し、制御を呼び出し元に返すように実装してください。キャンセルイベントを意識して実装することは、オペレーションの継続期間や、NSOperationのサブクラスに実装したか、いずれかの具象サブクラスを用いたか、にかかわらず重要です。isCancelledメソッド自身は非常に軽量で、頻繁に呼び出しても性能にほとんど影響を及ぼしません。オペレーションオブジェクトを設計する際には、コード中の次の箇所で、isCancelledメソッドを呼び出すことを検討してください。

- 実際の処理を始める直前
- ループ処理ごとに少なくとも1回（1回あたりの処理が比較的長いならばもっと頻繁に）
- オペレーションを中断しやすい箇所

リスト 2-4に、オペレーションオブジェクトのmainメソッド内でキャンセルイベントに応答する、非常に簡単なコード例を示します。isCancelledこの場合、isCancelledメソッドはwhileループを繰り返すごとに呼び出されているので、処理の開始前および一定時間ごとに、いつでも迅速に抜け出せます。

リスト 2-4　キャンセル要求への応答

```
- (void)main {
   @try {
      BOOL isDone = NO;

      while (![self isCancelled] && !isDone) {
          // 何らかの処理を行い、終了したらisDoneをYESにする
      }
   }
   @catch(...) {
      // 例外を再スローしない。
   }
}
```

この例ではクリーンアップ用のコードを省略していますが、実際には、コード内で確保したリソースを、確実に解放してください。

オペレーションを並列実行するための設定

オペレーションオブジェクトは、デフォルトでは、同期的に実行するようになっています。すなわち、その`start`メソッドを呼び出したスレッド内で、タスクを実行します。しかし、非並列オペレーションに対してはオペレーションキューがスレッドを用意するので、ほとんどのオペレーションが非同期に動作します。一方、オペレーションを（オペレーションキューを介さずに）直接実行して、非同期に動作させたいのであれば、そのための作業が必要です。すなわち、オペレーションオブジェクトを、並列オペレーションとして定義する必要があります。

表 2-2に、並列オペレーションを実装するためにオーバーライドするべきメソッドを示します。

表 2-2　並列オペレーションのためにオーバーライドするメソッド

メソッド	Description
`start`	（必須）並列オペレーションはすべて、このメソッドをオーバーライドして、デフォルトの振る舞いを、独自の実装で置き換えなければなりません。オペレーションを直接実行する際には、`start`メソッドを呼び出します。すなわちこのメソッドが、オペレーションの開始点となります。ここでスレッドその他の実行環境を用意し、タスクを実行することになります。オーバーライドしたコードから、`super`を呼び出すことはできません。

メソッド	Description
main	（必要に応じて）通常はここに、オペレーションオブジェクトの本来の目的であるタスクを実装します。startメソッドにタスクを実装することもできますが、ここに実装すれば、セットアップ処理とタスクのコードを明確に分離できます。
isExecuting isFinished	（必須）並列オペレーションは、自分自身の実行環境をセットアップし、その状態を外部のクライアントに通知する必要があります。したがって、タスクが実行中であるか、終了済みであるか、を表す状態情報を管理しなければなりません。このメソッドは、以上の状態情報を返します。 なお、このメソッドは、ほかのスレッドから同時に呼び出されても安全であるように実装しなければなりません。また、このメソッドか返す値が変化したときには、所定のキーパスに対応する、適切なKVO通知を生成する必要があります。
isConcurrent	（必須）これが並列オペレーションである旨を示すため、このメソッドをオーバーライドし、YESを返すように実装します。

以下、MyOperationクラスの実装例を示します。並列オペレーションの実装に、最低限必要な事項を組み込んであります。MyOperationクラスでは、専用のスレッドを生成し、ここで独自のmainメソッドを実行します。mainメソッドの、実際の作業内容については、特に説明しません。コード例の目的は、並列オペレーションとして実装するために必要な「基盤」処理を示すことだからです。

リスト 2-5に、MyOperationクラスのインターフェイスと、実装の一部を示します。MyOperationクラスの、isConcurrent、isExecuting、isFinishedメソッドの実装は、ごく単純です。isConcurrentメソッドでは、これが並列オペレーションであることを示すため、単にYESを返さなければなりません。isExecutingおよびisFinishedは、このクラスで定義したインスタンス変数の値を返すだけです。

リスト 2-5　並列オペレーションの定義

```
@interface MyOperation : NSOperation {
    BOOL        executing;
    BOOL        finished;
}
- (void)completeOperation;
@end

@implementation MyOperation
- (id)init {
```

```
    self = [super init];
    if (self) {
        executing = NO;
        finished = NO;
    }
    return self;
}

- (BOOL)isConcurrent {
    return YES;
}

- (BOOL)isExecuting {
    return executing;
}

- (BOOL)isFinished {
    return finished;
}
@end
```

リスト2-6に、MyOperationのstartメソッドを示します。ここでは、必ず実行しなければならない、最小限の処理だけを実装しています。単に新しいスレッドを起動し、mainメソッドを呼び出すように設定します。また、executingメンバー変数を更新し、それを反映するため、isExecutingキーパスに対応するKVO通知を生成しています。この処理が終わると、新たにデタッチしたスレッドに実際のタスクを任せて、呼び出し元に制御を返します。

リスト 2-6　startメソッド

```
- (void)start {
   // タスクを起動する前に、キャンセルされていないかどうか確認する。
   if ([self isCancelled])
   {
      // キャンセルされていれば、オペレーションを完了状態に移行する。
      [self willChangeValueForKey:@"isFinished"];
```

```
        finished = YES;
        [self didChangeValueForKey:@"isFinished"];
        return;
    }

    // キャンセルされていなければ、タスクの実行を開始する。
    [self willChangeValueForKey:@"isExecuting"];
   [NSThread detachNewThreadSelector:@selector(main) toTarget:self withObject:nil];
    executing = YES;
    [self didChangeValueForKey:@"isExecuting"];
}
```

リスト 2-7に、MyOperationクラスの、残りの実装を示します。リスト 2-6 （26 ページ）に示したように、mainメソッドは新しいスレッドの入り口点です。オペレーションオブジェクトの本来の目的である処理を行い、終了後、独自に定義したcompleteOperationメソッドを呼び出しています。completeOperationメソッドでは、オペレーションの状態変化を反映するため、isExecutingおよびisFinishedキーパスに対応するKVO通知を生成します。

リスト 2-7　処理完了時にオペレーションの状態を更新

```
- (void)main {
    @try {

        // オペレーション本来の処理をここで実行する。

        [self completeOperation];
    }
    @catch(...) {
        // 例外を再スローしない。
    }
}

- (void)completeOperation {
    [self willChangeValueForKey:@"isFinished"];
    [self willChangeValueForKey:@"isExecuting"];
```

```
    executing = NO;
    finished = YES;

    [self didChangeValueForKey:@"isExecuting"];
    [self didChangeValueForKey:@"isFinished"];
}
```

オペレーションがキャンセルされた場合でも、KVOオブザーバには、処理が完了した旨を通知しなければなりません。オペレーションオブジェクトは、ほかのオペレーションオブジェクトとの依存関係が設定されている場合、当該オブジェクトの`isFinished`キーパスを監視しています。依存するすべてのオブジェクトから、完了した旨の通知を受け取った後に、処理を開始するようになっているのです。したがって、完了通知を生成しないでおくと、依存される側のオペレーションが実行されないままになってしまいます。

KVOの規約準拠

`NSOperation`クラスは、次のキーパスについて、KVO（Key-Value Observing）の規約に従っています。

- `isCancelled`
- `isConcurrent`
- `isExecuting`
- `isFinished`
- `isReady`
- `dependencies`
- `queuePriority`
- `completionBlock`

`start`メソッドをオーバーライドした、あるいは`NSOperation`オブジェクトに対して、`main`をオーバーライドする以外にも大きなカスタマイズを施した場合、上記のキーパスについて、KVOの規約に準拠しなければなりません。`start`メソッドをオーバーライドする際、`isExecuting`と`isFinished`には、特に意識して対応してください。このメソッドを実装し直すと、多くの場合、影響を受けることになるキーパスだからです。

`isReady`ほかのオペレーションオブジェクトとの依存関係に加え、それ以外の何かとの依存関係も管理したい場合は、isReadyメソッドもオーバーライドして、その依存関係が満たされるまでは`NO`を返すようにしなければなりません（独自の依存関係を実装し、なおかつ、`NSOperation`クラスが提供する

デフォルトの依存関係管理システムも利用するのであれば、`isReady`の中から`super`を呼び出してください）。オペレーションオブジェクトの準備状況に変化があったときには、`isReady`キーパスに対応するKVO通知を生成して、その旨を通知します。`addDependency:`または`removeDependency:`メソッドをオーバーライドしていない限り、`dependencies`キーパスに対応するKVO通知は考慮しなくて構いません。

`NSOperation`のほかのキーパスに対応するKVO通知も生成できますが、その必要が生じることはほとんどないでしょう。オペレーションをキャンセルするためには、はじめから用意されている`cancel`メソッドを呼び出すだけで済みます。同様に、オペレーションオブジェクトでキューの優先度情報を変更しなければならないことも、ほとんどないでしょう。最後に、並列処理するかどうかを動的に変更する場合を除き、`isConcurrent`キーパスに対応するKVO通知も必要ありません。

KVO（Key-Value Observing）について、およびカスタムオブジェクトでこれを処理する方法については、『*Key-Value Observing Programming Guide*』を参照してください。

オペレーションオブジェクトの、実行時の振る舞いのカスタマイズ

生成したオペレーションオブジェクトに対して、キューに追加する前に、必要な設定を行います。このセクションで説明する設定は、`NSOperation`のサブクラスとして実装したか、既存のサブクラスを使っているかにかかわらず、オペレーションオブジェクトすべてに適用されます。

オペレーション間の依存関係の設定

依存関係は、各オペレーションオブジェクトを直列に（順に）実行する手段として使えます。ほかのオペレーションに依存している場合、それらすべてが実行完了するまで、実行を開始できません。2つのオペレーションオブジェクト間に、単純に1対1の依存関係を定義するだけでなく、有向グラフの形でオブジェクト間の複雑な依存関係を設定することも可能です。

2つのオペレーションオブジェクト間に依存関係を設定するためには、`NSOperation`の`addDependency:`メソッドを使います。あるオペレーションオブジェクトから、引数として指定する依存先オペレーションに向かう、一方向の依存関係を設定するメソッドです。これは、依存先オペレーションオブジェクトがすべて実行完了するまで、実行を開始できないことを表します。依存関係は、同じキューに登録したオペレーションの範囲内に限定されません。オペレーションオブジェクトがそれぞれ自分自身の依存関係を管理しているので、オペレーション間に依存関係を設定した上で、それぞれ異なるキューに登録することも、問題なく可能です。ただし、依存先をたどって行くと自分自身に戻ってしまうような依存関係は設定できません。関係するオペレーションが永久に実行されなくなってしまいます。

依存先オペレーションがすべて完了すると、実行可能な状態になります（`isReady`メソッドの振る舞いを変更している場合は、その条件に従ってオペレーションの状態が決まります）。当該オペレーションオブジェクトがキューにあれば、いつでも実行を開始できるわけです。一方、（キューを介さずに）直接実行する場合、`start`メソッドをどの時点で呼び出すか、は開発者に任されています。

> **Important:** 依存関係を設定するのは、オペレーションを実行する前、あるいはオペレーションキューに登録する前でなければなりません。実行が始まってから依存関係を設定しても、実行を停止することはできないのです。

依存関係は、各オペレーションオブジェクトが、状態が変化すればいつでも適切なKVO通知を送る、という規約に基づいて実現されています。オペレーションオブジェクトの振る舞いをカスタマイズする場合、適切なKVO通知を生成するようコード中に記述しなければ、依存関係に問題が生じるかもしれません。KVO通知およびオペレーションオブジェクトについては、"KVOの規約準拠"（28 ページ）を参照してください。依存関係の設定については、『*NSOperation Class Reference*』を参照してください。

オペレーションの実行優先度の変更

キューに追加したオペレーションの実行順序は、第1にそれが実行可能な状態かどうか、第2に相対的な優先度によって決まります。実行可能かどうかは、ほかのオペレーションとの依存関係によって決まるのに対し、優先度はオペレーションオブジェクト自身の属性です。オペレーションオブジェクトを生成した当初はすべて「標準」の優先度になっていますが、必要に応じ、`setQueuePriority:`メソッドで増減できます。

優先度が実行順序が反映されるのは、同じオペレーションキューに入っているオペレーションどうしの間に限ります。複数のオペレーションキューがある場合、ほかのキューとは関係なく、それぞれが優先度を判断することになります。したがって、キューが別々であれば、優先度の低い方が先になることもありえます。

優先度は、依存関係の代替機能ではありません。実行可能な状態になっているオペレーションの間だけで、オペレーションキューが実行を開始する順序を決めるものです。たとえば、あるキューに高優先度と低優先度のオペレーションがあって、どちらも実行可能な状態であれば、高優先度の方が先になります。しかし、高優先度のオペレーションが実行可能な状態になっておらず、低優先度の方だけが実行可能であれば、先に低優先度のオペレーションが実行されます。あるオペレーションが完了するまで実行を待って欲しい、という状況では、依存関係の仕組みを使わなければなりません（"オペレーション間の依存関係の設定"（29 ページ）を参照）。

基盤となるスレッドの優先度変更

OS X v10.6以降では、オペレーションが動作基盤としているスレッドの実行優先度を設定できます。システムのスレッド処理方針はカーネルが管理していますが、一般に優先度が高いスレッドは、低いものに比べ、実行される機会が多くなります。スレッドの優先度は、オペレーションオブジェクト側で、0.0から1.0までの浮動小数点数で指定してください。0.0は最も低い優先度、1.0は最も高い優先度です。明示的に指定しなければ、オペレーションは、デフォルトの優先度である0.5で動作します。

オペレーションのスレッドの優先度を設定するためには、オペレーションオブジェクトの`setThreadPriority:`メソッドを、キューに追加する（または直接実行する）前に呼び出さなければなりません。オペレーションを実行する段になると、デフォルトの`start`メソッドは、指定に従い、現在動作しているスレッドの優先度を修正するようになっています。この新しい優先度は、オペレーションの`main`メソッドが動作している間のみ有効です。ほかのコード（オペレーションの完了ブロックを含む）は、デフォルトのスレッド優先度で動作します。ただし、並列オペレーションを生成している（したがって`start`メソッドをオーバーライドしている）場合は、スレッド優先度を自分で設定しなければなりません。

完了ブロックのセットアップ

OS X v10.6以降では、主たるタスクの実行終了後、完了ブロックを実行するよう、オペレーションを設定できます。完了ブロックでは、主たるタスクで行うべきとは考えられない処理を、何でも実行できます。たとえば、オペレーションが完了した旨を関係するクライアントに通知する、という処理が考えられます。あるいは並列オペレーションオブジェクトの場合、完了ブロックで最後のKVO通知を生成するとよいかもしれません。

完了ブロックの設定には、`setCompletionBlock:`の`NSOperation`メソッドを使います。このメソッドに渡すブロックは、引数も戻り値もないものでなければなりません。

オペレーションオブジェクトの実装に関するヒント

オペレーションオブジェクトの実装は比較的容易ですが、実際にコードを記述する際に知っておくべき事項がいくつかあります。以下のセクションでは、オペレーションオブジェクトのコードを記述する上で考慮するべき事項を解説します。

オペレーションオブジェクトにおけるメモリの管理

以下のセクションでは、オペレーションオブジェクトで良好にメモリ管理を行う上で大切な事項を説明します。Objective-Cのプログラムにおけるメモリ管理についての一般論は、『*Advanced Memory Management Programming Guide*』を参照してください。

スレッドに属するストレージの回避

多くのオペレーションはスレッド上で実行されますが、非並列オペレーションの場合、このスレッドは普通、オペレーションキューが用意します。この場合、スレッドはキューが所有するのであって、オペレーション側から自由に操作できるものではない、ということに注意してください。特に、自分が生成したものでも、管理できるものでもないスレッドに、データを関連づけることはできません。オペレーションキューが管理するスレッドは、システムやアプリケーションの都合に合わせて生成、解放されます。したがって、スレッドに属するストレージを介してオペレーション間でデータをやり取りすると、信頼性が損なわれ、正しく処理できないおそれがあります。

オペレーションオブジェクトの場合、スレッドに属するストレージを使わなければならない理由は何もありません。オペレーションオブジェクトを初期化する際には、オブジェクトに対して、ジョブに必要なリソースをすべて提供するはずです。したがって、オペレーションオブジェクト自身が、状況に応じてストレージを用意します。出し入れするデータはすべて、アプリケーション側に返すか必要なくなるまで、このストレージに保存しておいてください。

必要ならばオペレーションオブジェクトへの参照を保持すること

オペレーションオブジェクトは非同期に動作するので、生成し、キューに追加してしまえば、あとは「忘れて」しまってよい、ということにはなりません。これもオブジェクトであることに変わりはないので、開発者が必要に応じてその参照を管理してください。これは特に、オペレーションの完了後、ここから結果データを検索したい場合に重要です。

オペレーションへの参照を保持しておくべきなのは、後になってキューに問い合わせて、オブジェクトを取得できるとは限らないからです。キューは、できるだけ迅速にオペレーションをディスパッチ、実行しようとしています。多くの場合、オペレーションをキューに追加すると、ほぼ直後にその実行が始まります。キューに問い合わせて、オペレーションオブジェクトへの参照を取得しようとしても、その頃にはオペレーションが完了し、キューから削除されてしまっているかもしれません。

エラーや例外の取り扱い

オペレーションは本質的に、アプリケーション内で独立したエンティティなので、エラーや例外が発生した場合、その中で処理する必要があります。OS X v10.6以降では、`start`クラスに定義されたデフォルトの`NSOperation`メソッドは、例外を捕捉しないようになっています（OS X v10.5では、例外を捕捉して捨てるようになっていました）。独自に開発するコードでは、例外を捕捉し、捨てるようにしてください。また、エラーコードを確認し、必要ならばアプリケーションの適切な部分に通知しなければなりません。さらに、`start`メソッドを置き換えるのであれば、独自の実装の中で、同じように例外を捕捉して、スレッドのスコープから離れないようにする必要があります。

さまざまなエラーのうち、対処しなければならないのは次のようなものです。

- UNIXの`errno`の形で返されるエラーコードの検査と処理。

- メソッドや関数から明示的に返されるエラーコードの検査。
- 独自に作成したコード、またはほかのシステムフレームワークから発生する例外の捕捉。
- `NSOperation`クラス自身が、次のような状況でスローした例外の捕捉。
    - オペレーションが実行可能な状態でないのに、`start`メソッドが呼び出された場合
    - オペレーションが既に実行中または完了している（おそらくキャンセルされたため）のに、再度`start`メソッドが呼び出された場合
    - 既に実行中または完了しているオペレーションに、完了ブロックを追加しようとしたとき
    - キャンセルされたはずの`NSInvocationOperation`オブジェクトに結果を問い合わせたとき

カスタムコード中で例外やエラーを捕捉した場合は、アプリケーション中の必要な部分に、エラーを伝える必要があります。`NSOperation`クラスには、エラーを表す結果コードや例外を、アプリケーションのほかの部分に渡す、専用のメソッドはありません。したがって、このような情報が重要であれば、そのための処理コードを記述してください。

オペレーションオブジェクトの適切な処理量の判断

オペレーションキューにはいくらでも多くのオペレーションを追加できますが、おのずと限度があります。ほかのオブジェクトと同様、`NSOperation`クラスのインスタンスもメモリを消費し、その実行にはコストがかかります。ちょっとした処理しかしないオペレーションオブジェクトを数万個単位で生成したとすれば、実質的な処理よりもディスパッチ処理の方に時間がかかってしまいます。また、アプリケーションにメモリの制約があれば、数万個単位のオペレーションオブジェクトをメモリ中に確保すると、性能が急速に落ちていきます。

オペレーションを効率的に動かすために大切なのは、必要な処理量と、コンピュータの稼働率とのバランスを取ることです。ひとつひとつのオペレーションの処理量を、合理的な範囲に保つ必要があります。たとえば、100個のオペレーションオブジェクトを生成し、値が異なるだけでまったく同じ100個のタスクを実行する代わりに、オペレーションオブジェクトを10個だけ生成し、それぞれに10個の値を渡して処理させる方がよいかもしれません。

また、大量のオペレーションを同時にキューに追加するのも、処理が追い付かないほどの速度で連続して追加するのも、避けた方がよいでしょう。キューがオペレーションオブジェクトであふれてしまうようであれば、オブジェクトを分割して生成することも検討してください。分割したある一群が処理を終えた時点で、完了ブロックを使って、新たに一群のオペレーションオブジェクトを生成するよう、アプリケーションに通知する方式です。大量の処理がある場合、キューにはできるだけ多くのオペレーションを追加して、コンピュータが常にビジー状態であるようにしたいかもしれませんが、同時に多数のオペレーションを生成しすぎて、メモリ不足に陥るのは望ましくありません。

もちろん、生成するオペレーションオブジェクトの適切な個数、したがってそれぞれの処理量は、アプリケーションによって異なるでしょう。InstrumentsやSharkのようなツールを使って、効率と速度の最適なバランスを見つけるようにしてください。Instruments、Shark、その他の性能計測ツールで何を計測できるか、については、『*Performance Overview*』を参照してください。

オペレーションの実行

最終的にアプリケーションは、オペレーションを実行して、必要な処理を行うことになります。このセクションでは、オペレーションを実行するいくつかの方式と、実行時にオペレーションの実行状況を管理する方法を紹介します。

オペレーションをオペレーションキューに追加

オペレーションを実行する最も簡単な方法は、`NSOperationQueue`クラスのインスタンスである、オペレーションキューを使うというものです。オペレーションキューの生成と維持は、アプリケーション側で行います。キューはいくつでも生成できますが、同時に実行できるオペレーションの数には、おのずと限界があります。オペレーションキューはシステムと連携して、並列オペレーションの個数を、コア数やシステム負荷に応じた適切な範囲に制限します。したがって、キューを追加生成したからといって、多数のオペレーションを実行するようになるとは限りません。

アプリケーション内でキューを生成する方法は、次のように、普通のオブジェクトと同じです。

```
NSOperationQueue* aQueue = [[NSOperationQueue alloc] init];
```

オペレーションをキューに追加するためには、`addOperation:`メソッドを使います。OS X v10.6以降では、複数のオペレーションをまとめて`addOperations:waitUntilFinished:`メソッドで追加する方法、あるいは、ブロックオブジェクトをキューに、（対応するオペレーションオブジェクトを使わず）直接、`addOperationWithBlock:`メソッドで追加する方法もあります。上記のメソッドはいずれも、オペレーションをキューに追加し、処理を開始するようキューに通知します。多くの場合、オペレーションをキューに追加するとすぐに実行が始まりますが、何らかの理由で遅延が生じることもあります。たとえば、依存するオペレーションがあって、それがまだ完了していない場合です。また、オペレーションキュー自身が一時的に停止していたり、実行可能な並列オペレーションの最大数に達していたりする場合にも、遅延が生じます。オペレーションをキューに追加するコードの、基本的な構文を以下に示します。

```
[aQueue addOperation:anOp]; // 単一のオペレーションを追加

[aQueue addOperations:anArrayOfOps waitUntilFinished:NO]; // 複数のオペレーションを追
加
```

```
[aQueue addOperationWithBlock:^{
    /* 何らかの処理を実行*/
}];
```

Important: キューに追加した後で、オペレーションオブジェクトを修正することはできません。キューで待機中、いつ実行が始まるかわからないので、その間に依存関係やデータを変更すると不具合が起こりえます。オペレーションの状態を知りたい場合は、`NSOperation`クラスの該当するメソッドを使って、実行中か、待機中か、既に完了しているかを調べてください。

`NSOperationQueue`クラスはオペレーションの並列実行を想定して設計されていますが、同じキューでは同時に1つのオペレーションしか実行しないよう強制することも可能です。`setMaxConcurrentOperationCount:`メソッドで、オペレーションキューオブジェクトが同時に実行できる並列オペレーションの最大数を設定できます。この値を1とすれば、同時には1つのオペレーションしか実行しないようになります。とはいえその順序は、各オペレーションが実行可能な状態かどうか、あるいは優先度など、ほかの要因によって決まります。したがって、その振る舞いは、Grand Central Dispatchの直列ディスパッチキューとまったく同じにはなりません。オペレーションオブジェクトの実行順序が重要であれば、キューに追加する前に、依存関係を設定するとよいでしょう。依存関係の設定については、"オペレーション間の依存関係の設定"（29 ページ）を参照してください。

オペレーションキューの使いかたについては、『*NSOperationQueue Class Reference*』を参照してください。直列ディスパッチキューについては、"直列ディスパッチキューの生成"（45 ページ）を参照してください。

オペレーションの直接実行

オペレーションオブジェクトの実行にはオペレーションキューを使うのが便利ですが、キューを使わずに直接実行することも可能です。しかしその場合、コードを記述する上で注意するべき事項がいくつかあります。特に、オペレーションが実行可能な状態になってから、`start`メソッドで開始する、という順序を守ることが大切です。

オペレーションは、`isReady`メソッドが`YES`を返すようになるまで、実行できません。`isReady`メソッドは、依存関係に関する、`NSOperation`クラスの管理機構に組み込まれています。依存するオペレーションがすべて完了してからでないと、実行を開始できません。

オペレーションを直接実行する際には、必ず`start`メソッドを使ってください。`main`その他のメソッドを使わないのは、`start`メソッドが、実行しても安全かどうか、事前に確認するようになっているからです。特に、デフォルトの`start`メソッドは、依存関係を正しく処理できるよう、KVO通知を生成します。また、キャンセルされたオペレーションは実行を回避します。さらに、実行可能な状態になっていないのに実行しようとすれば、例外を投げるようになっています。

アプリケーションで並列オペレーションオブジェクトを定義する際には、起動する前に、`isConcurrent`メソッドで状態を確認するとよいかもしれません。このメソッドが`NO`を返した場合は、現在のスレッド内で同期的に実行するか、別にスレッドを生成して実行するか、という選択肢があります。しかし、その判断基準の実装は、開発者に任されています。

リスト 2-8に、上記のような判断をした上でオペレーションを直接実行する、簡単なコード例を示します。例に挙げたメソッドが`NO`を返した場合、タイマーを設定し、しばらくしてから再度呼び出して、実行を試みるとよいでしょう。`YES`を返すようになる（オペレーションがキャンセルされた、など）まで、タイマーの設定を繰り返します。

リスト 2-8　オペレーションオブジェクトを直接実行

```
- (BOOL)performOperation:(NSOperation*)anOp
{
   BOOL        ranIt = NO;

   if ([anOp isReady] && ![anOp isCancelled])
   {
      if (![anOp isConcurrent])
         [anOp start];
      else
         [NSThread detachNewThreadSelector:@selector(start)
                   toTarget:anOp withObject:nil];
      ranIt = YES;
   }
   else if ([anOp isCancelled])
   {
      // 開始前にキャンセルされている場合は、
      //  オペレーションを完了状態に移行する。
      [self willChangeValueForKey:@"isFinished"];
      [self willChangeValueForKey:@"isExecuting"];
      executing = NO;
      finished = YES;
      [self didChangeValueForKey:@"isExecuting"];
      [self didChangeValueForKey:@"isFinished"];

      // ranItをYESと設定して、今後再び、
```

```
        // 同じオペレーションを引数として呼び出されないようにする。
        ranIt = YES;
    }
    return ranIt;
}
```

オペレーションのキャンセル

オペレーションオブジェクトは、オペレーションキューに追加するとその実効所有者がキューになるため、削除できません。キューから外したい場合は、キャンセル操作を施すことになります。ある特定のオペレーションオブジェクトをキャンセルしたい場合は`cancel`メソッドを使います。キューに登録したオペレーションオブジェクトをまとめてキャンセルしたい場合は、キューオブジェクトの`cancelAllOperations`メソッドを呼び出してください。

オペレーションをキャンセルしてよいのは、その処理がもはや必要でなくなった場合に限ります。キャンセルコマンドを発行すると、オペレーションオブジェクトは「キャンセル」状態になり、実行されなくなります。キャンセルされたオペレーションも「完了」したものと見なされるので、これに依存するオブジェクトには、依存関係を解消する適切なKVO通知が送られます。キャンセル操作の多くは、アプリケーションの終了、ユーザのキャンセル指示などといった重要なイベントによるもので、キューにあるオペレーションすべてを対象とするものでしょう。個別にキャンセルすることはあまりないはずです。

オペレーションの完了まで待機

高い処理性能を引き出すためには、オペレーションをできるだけ非同期のものとして設計し、オペレーションの実行中、アプリケーションがいつでも追加の処理ができるようにしてください。オペレーションを生成するコードが、その結果も処理するのであれば、`NSOperation`の`waitUntilFinished`メソッドを用いて、オペレーション完了までコードをブロックする、という方法もあります。しかし一般には、このメソッドの利用は避ける方がよいでしょう。現在のスレッドをブロックする方法は、状況によっては便利かもしれませんが、直列に動作するコード部分が多くなり、全体の並列性が損なわれます。

Important: 特にアプリケーションのメインスレッドでは、オペレーションを待機しないようにしてください。待機するとしても、二次スレッドまたはほかのオペレーションに限るべきです。メインスレッドをブロックすると、ユーザイベントに応答できないため、アプリケーションそのものがハングしたように見えてしまいます。

特定のオペレーションだけでなく、キューのすべてのオペレーションが完了するのを待機することも可能です。この目的には、`NSOperationQueue`の`waitUntilAllOperationsAreFinished`メソッドを使います。なお、キュー全体の完了を待機している間でも、アプリケーションのほかのスレッドがキューにオペレーションを追加すれば、待機期間がその分延びることに注意してください。

キューの一時停止と再開

オペレーションの実行を一時的に停止したい場合は、対応するオペレーションキューを、`setSuspended:`メソッドで一時停止してください。ただし、既に実行中のオペレーションが、タスクの途中で中断することはありません。新しいオペレーションの実行が始まらない、というだけです。進行中の処理を一時停止するようユーザが指示したとしても、ゆくゆくは再開することになると想定し、キューを停止するにとどめる、という実装もありえます。

ディスパッチキュー

Grand Central Dispatch（GCD）のディスパッチキューは、タスクを実行するためのツールとして、強力な機能を持っています。ディスパッチキューを使えば、任意のコードブロックを、呼び出し元と非同期的に、あるいは同期的に実行できます。独立したスレッド上で実行していたようなタスクは、ディスパッチキューを使っても、ほとんどすべて実行できます。しかも、スレッドを使う方法に比べ、より使いやすく効率もよい、という利点があります。

この章では、ディスパッチキューの概要を説明し、これを使ってアプリケーションの一般的なタスクを実行する方法を解説します。スレッドを使っていたコードをディスパッチキューで実装し直す際には、“スレッドを使ったコードからの移行” （75 ページ）に挙げるヒントも参照してください。

ディスパッチキューについて

ディスパッチキューは、アプリケーションのタスク群を非同期かつ並列に実行する手段として、容易に利用できます。**タスク**とは、アプリケーションが実行するべき、何らかの処理を表す言葉です。何らかの計算をする、データ構造を生成/修正する、ファイルからデータを読み込んで加工する、などといった処理がタスクに当たります。タスクの定義は、そのためのコードを関数またはブロックオブジェクトの形で記述し、ディスパッチキューに追加することにより行います。

ディスパッチキューは、オブジェクトに似た構造を持っており、登録されたタスクを管理する働きがあります。ディスパッチキューはいずれも、先入れ先出し方式のデータ構造です。すなわち、タスクはキューに追加した順に実行されることになります。GCDには最初からいくつかディスパッチキューが用意されていますが、それとは別に、目的に応じて生成することも可能です。表3-1に、アプリケーションで利用可能なディスパッチキューの型と、その使いかたを示します。

表 3-1 ディスパッチキューの型

型	Description
直列	直列キュー（あるいは**プライベートディスパッチキュー**）は、タスクを同時にはひとつずつ、追加された順に実行します。タスクは、ディスパッチキューが管理する、独立したスレッド（タスクによって異なりうる）上で動作します。あるリソースに同期アクセスするためによく使われます。 直列キューは、必要に応じていくつでも生成し、ほかのキューとは並列に動かすことができます。たとえば4つの直列キューを用意すれば、それぞれは同時にひとつずつタスクを実行しますが、全体としては4つのタスクが並列に動作することになります。直列キューの生成手順については、“直列ディスパッチキューの生成”（45 ページ）を参照してください。
並列	並列キュー（あるいは**グローバルディスパッチキュー**の一種）は、複数のタスクを並列に実行しますが、各タスクが実行を開始する順序は、キューに追加された順序どおりになります。タスクは、ディスパッチキューが管理する、独立したスレッド上で動作します。ある時点で動作しているタスクの数は、システムの状況に応じて変わります。 iOS 5以降、キューの型として`DISPATCH_QUEUE_CONCURRENT`を指定することにより、自分で並列ディスパッチキューを生成できるようになりました。さらに、アプリケーションで自由に利用できるグローバルな並列キューが4つ、あらかじめ用意されています。グローバルな並列キューを取得する方法については、“グローバルな並列ディスパッチキューの取得”（44 ページ）を参照してください。
メインディスパッチキュー	アプリケーションのどこからでも利用できる直列キューで、アプリケーションのメインスレッド上でタスクを実行します。アプリケーションに実行ループがあればこれと連動し、キューに登録されたタスクとほかのイベントの処理を、交互に行います。メインスレッド上でタスクを実行するので、アプリケーションの主要な同期ポイントとしてもよく使われます。 メインディスパッチキューはあらかじめ生成された状態になっていますが、適切に破棄するのはアプリケーション側の責任です。このキューの管理方法については、“メインスレッド上でタスクを実行”（52 ページ）を参照してください。

アプリケーションに並列性を導入する手段としてはスレッドもありますが、ディスパッチキューにはいくつか利点があります。端的な利点として、キューを用いるプログラミングモデルは非常に扱いやすい、ということがあります。スレッドの場合、実行したい本来の処理と、スレッド自身の生成や管理の、両方を意識しながらコードを記述しなければなりません。一方、ディスパッチキューを使えば、本来の処理にのみ集中して記述でき、スレッドの生成や管理について気にする必要はありません。これはシステムが代行してくれるからです。この方式には、アプリケーションがそれぞれスレッドを管理するよりも効率的になる、という利点があります。リソースの状況やシステムの状態に応じ、動的にスレッド数を調整できるのです。さらにタスクの起動も、スレッドをそれぞれ生成する方式に比べ、迅速になります。

ディスパッチキューを用いるようコードを書き換えるのは難しそうに思えるかもしれませんが、多くの場合、スレッドを使うよりも記述は容易です。重要なのは、自己完結したタスクを設計し、非同期的に動作できるようにすることです（これはスレッドでもディスパッチキューでも同じです）。しかしディスパッチキューには、振る舞いを事前に予測できる、という利点があります。同じ共有リソースにアクセスする2つのタスクが異なるスレッド上で動作していれば、どちらが先にこのリソースを書き換えるか予測できないので、両者が同時に書き換えようとすることのないよう、ロックをかける必要があります。一方、ディスパッチキューの場合、タスクを直列ディスパッチキューに追加するだけで、両者が同時に書き換えようとすることがない旨を保証できます。このような、キューを用いた同期機構は、ロックよりも効率的です。ロックの場合、競合の有無にかかわらずカーネルトラップが必要であり、それにはコストがかかります。一方、ディスパッチキューは主としてアプリケーションのプロセス空間で動作し、本当に必要な場合にカーネルの機能を呼び出すだけで済むからです。

直列キューに入れた2つのタスクが並列に動作することは決してないのに対し、2つのスレッドが同時にロックを獲得することがありうるとすれば、スレッドにより実現していた並列性はまったく発揮されなくなるか、そうでなくても大きく損なわれてしまいます。さらに重要なのは、スレッドモデルの場合2つのスレッドを生成する必要があるので、カーネル空間とユーザ空間のメモリを消費する、ということです。ディスパッチキューの場合はこのようなメモリ消費がなく、キューがタスクを実行するスレッドは常にビジー状態で、ブロックされることはありません。

ディスパッチキューについては次の事項にも注意してください。

- ディスパッチキューは登録されたタスクを、ほかのディスパッチキューに登録されたタスクとは並列に実行します。直列に実行するのは、同じディスパッチキューに登録されたタスクの範囲内だけです。
- システムは同時に実行するタスク数を調整します。したがって、100個のタスクを、100個のキューにそれぞれ追加したとしても、すべてが並列に動作するとは限りません（実質的なコア数が100以上の場合を除く）。
- どのタスクを先に起動するか判断する際には、キューの優先度も考慮します。直列キューの優先度の設定については、“キューのクリーンアップ関数の組み込み”（47 ページ）を参照してください。
- タスクはキューに追加する時点で実行可能な状態になっていなければなりません（Cocoaのオペレーションオブジェクトを使ったことがある方は、そのモデルとは動作が違っているので注意してください）。
- プライベートディスパッチキューは、参照カウントで管理されるオブジェクトです。コードを記述する際には、キューを保持するだけでなく、キューにディスパッチソースをアタッチしたら、参照カウンタも増やしてください。その後、使用済みになったディスパッチソースはすべて確実にキャンセルすること、保持と解放の関数を対にして呼び出すことも大切です。キューの保持と解放については、“ディスパッチキューのメモリ管理”（46 ページ）を参照してください。ディスパッチソースについては、“ディスパッチソースについて”（57 ページ）を参照してください。

ディスパッチキューを操作するインターフェイスについては、『*GrandCentralDispatch(GCD)Reference*』を参照してください。

キューに関係する技法

Grand Central Dispatchはディスパッチキュー以外にも、キューを用いるコードが記述しやすいよう、いくつかの技法を提供しています。表3-2に、その技法の概要と、詳細を説明したページを示します。

表 3-2 ディスパッチキューを用いる技法

テクノロジー	Description
ディスパッチグループ	一連のブロックオブジェクトをまとめて、処理が完了したかどうかを監視する仕組みです（ブロック群の監視は、同期的にでも非同期的にでも可能です）。複数のタスクとの依存関係がある（すべて完了しなければ実行できない）場合に、同期機構として役立ちます。グループの使いかたについては、“キューに追加されたタスクのグループを待機”（54 ページ）を参照してください。
ディスパッチセマフォ	従来からあるセマフォと似ていますが、より効率的に動作します。カーネル機能を呼び出すのは、セマフォが使えないため、呼び出し元のスレッドをブロックしなければならない場合だけだからです。セマフォが使えれば、カーネル呼び出しは起こりません。ディスパッチセマフォの使用例は、“ディスパッチセマフォを使って有限のリソースの使いかたを規制”（53 ページ）を参照してください。
ディスパッチソース	所定の型のシステムイベントに応じて通知を生成します。プロセス通知、シグナル、記述子などのイベントを監視するために有用です。ディスパッチソースは、イベントが発生すると、タスクコードを非同期的にディスパッチキューに登録して処理を委ねます。ディスパッチソースの生成や使いかたについては、“ディスパッチソース”（57 ページ）を参照してください。

ブロックによるタスクの実装

ブロックオブジェクトはCベースの言語機能で、C、Objective-C、C++のコード中で使えます。自己完結した処理の単位を容易に定義できる仕組みです。関数ポインタと同じように見えるかもしれませんが、実際にはオブジェクトによく似たデータ構造を用いて実現されており、生成や管理はコンパイラが行います。コンパイラは、ブロックのコード（および関係するデータ）をパッケージ化し、ヒープ上に置いたり、アプリケーション内で受け渡ししたりできる形式でカプセル化します。

ブロックには、レキシカルスコープ外からも変数にアクセスできる、という利点があります。関数やメソッドの内部にブロックを定義した場合、そのブロックは、ある面では従来からあるようなコードブロックと同じように振る舞います。親スコープで定義された変数の値を読み取れる、という点もそのひとつです。ブロックがアクセスした変数は、ヒープ上にある、ブロックのデータ構造内にコピーされ、以降はこちらにアクセスできるようになります。ブロックをディスパッチキューに追加して（非同期に）実行する場合、親スコープの変数の値を更新することはできません。しかし、同期して実行する場合は、変数名に`__block`キーワードを前置して記述することにより、親スコープの変数を更新する（データを返す）ことも可能です。

ブロックは、コード中に、関数ポインタと同様の構文で記述します。ただし、ブロック名には、アスタリスク（`*`）ではなくカレット（`^`）を前置します。関数ポインタと同じように、引数を渡したり、戻り値を受け取ったりすることができます。リスト 3-1に、コード中にブロックを定義し、同期して実行する方法を示します。変数`aBlock`をブロックとして宣言しています。int型の引数を1つ取り、戻り値はありません。次に、このプロトタイプに合致する実際のブロックを`aBlock`に代入し、インラインで宣言します。最終行ではブロックを直接実行し、指定された整数値を標準出力に印字しています。

リスト 3-1　簡単なブロックの例

```
int x = 123;
int y = 456;

// ブロックの宣言と代入
void (^aBlock)(int) = ^(int z) {
    printf("%d %d %d\n", x, y, z);
};

// ブロックを実行する
aBlock(789);   // 「123 456 789」と出力
```

ブロックを設計する際に考慮するべきガイドラインを以下に示します。

- ディスパッチキューを使って非同期に実行するブロックの場合、親である関数やメソッドのスカラ変数を参照し、ブロック内で使うのは安全です。しかし、大きな構造体やポインタベースの変数で、呼び出しコンテキストで確保、削除されるものは、この方法では参照できません。ブロックが実行されるまでの間に、ポインタの参照先メモリが無効になっている可能性があるからです。もちろん、メモリ（またはオブジェクト）を自分自身で確保し、メモリの所有権を明示的にブロックに渡すのであれば安全です。

- ディスパッチキューは、追加されたブロックをコピーしておき、実行終了後に解放するようになっています。したがって、キューに追加する際、明示的にブロックをコピーする必要はありません。
- 小さなタスクを実行する場合、キューは生のスレッドに比べて効率的であると言っても、ブロックを生成してキュー上で実行する、というオーバーヘッドはあります。ブロックの処理がごくわずかであれば、キューにディスパッチするよりもインラインで実行する方が、コストがかからないかもしれません。「ごくわずか」の処理量に相当するかどうか、性能計測ツールで実際に処理時間を計測し、比較してみるとよいでしょう。
- ブロックが動作するスレッドの側にデータをキャッシュして、（同じスレッド上で実行される）ほかのブロックからアクセスさせることはできません。同じキューに入れたタスク間でデータを共有したい場合は、ディスパッチキューのコンテキストポインタにデータを格納してください。ディスパッチキューのコンテキストデータにアクセスする方法については、“独自のコンテキスト情報をキューに格納”（47 ページ）を参照してください。
- ブロック内でObjective-Cのオブジェクトを数個以上生成する場合は、ブロックのコードの一部を@autoreleaseブロックで囲み、メモリ管理を委ねるとよいでしょう。GCDのディスパッチキューにも自動解放プールはありますが、このプールがいつ破棄されるか、何の保証もありません。アプリケーションが使用できるメモリ量に制約がある場合、独自に自動解放プールを生成すれば、自動解放されたオブジェクトのメモリを定期的に解放できることになります。

ブロックについて、特にその宣言方法や使いかたについては、『*Blocks Programming Topics*』を参照してください。ブロックをディスパッチキューに追加する手順については、“タスクをキューに追加”（49 ページ）を参照してください。

ディスパッチキューの生成と管理

タスクをキューに追加する前に、どんな種類のキューをどのように使うか、決めておかなければなりません。ディスパッチキューはタスクを、直列にも並列にも実行できます。また、キューの特別な使いかたを想定しているならば、それに応じて属性を設定することになるでしょう。以降の各セクションでは、ディスパッチキューを生成、設定する方法を示します。

グローバルな並列ディスパッチキューの取得

並列ディスパッチキューは、並列に実行できる複数のタスクがある場合に有用です。並列キューもキューの一種なので、タスクは先入れ先出しの順序で取り出されます。しかし、前のタスクが完了していなくても、次のタスクを取り出せるのが特徴です。同時に実行可能なタスク数は可変で、アプリケーションの状態に応じて動的に変わります。この数を決める要因として、プロセッサのコア数、ほかのプロセスが実行した処理量、ほかの直列ディスパッチキュー上にあるタスクの優先度など、さまざまなものが関与します。

各アプリケーションには、4つの並列ディスパッチキューが初めから用意されています。いずれもアプリケーション全体からアクセスでき、優先度以外はまったく同じです。グローバルという性質上、明示的に生成することはありません。初めから用意されているものを、次の例のように、dispatch_get_global_queue関数で取得してください。

```
dispatch_queue_t aQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
 0);
```

デフォルト優先度の並列キューだけでなく、高優先度、低優先度のキューも、引数として定数DISPATCH_QUEUE_PRIORITY_HIGHまたはDISPATCH_QUEUE_PRIORITY_LOWを渡すことにより取得できます。さらに、定数DISPATCH_QUEUE_PRIORITY_BACKGROUNDを渡すことにより、バックグラウンドキューも取得できます。名前が示すように、高優先度の並列キュー上にあるタスクは、デフォルト優先度または低優先度の並列キュー上にあるタスクよりも先に実行されます。同様に、デフォルト優先度のキュー上にあるタスクは、低優先度の並列キュー上のタスクよりも先に実行されます。

> **注意:** dispatch_get_global_queue関数の第2引数は、将来の拡張に備えて予約されています。当面は常に0を渡すようにしてください。

ディスパッチキューは参照カウントで管理されるオブジェクトですが、グローバルな並列キューの場合、保持や解放の必要はありません。グローバル（アプリケーション全体で有効）という性質上、保持や解放をする関数を呼び出しても無視されます。したがって、このキューの参照をどこかに格納しておく必要もありません。必要に応じてdispatch_get_global_queue関数を呼び出せば、いつでもキューの参照を取得できます。

直列ディスパッチキューの生成

直列キューは、タスクを指定どおりの順序で実行したい場合に有用です。同時には1つのタスクしか実行せず、常にキューの先頭から次のタスクを取り出すようになっています。共有リソースやミュータブルな（生成後も状態を変更できる）データ構造を保護するため、ロックの代わりに使うことも可能です。ロックと違い、タスクの実行順序も制御できます。さらに、タスクを直列キューに非同期に登録している限り、デッドロックのおそれもありません。

並列キューと違い、初めから用意されてはいないので、必要な直列キューは自分で生成、管理しなければなりません。いくらでも必要な個数の直列キューを生成できますが、同時に多数のタスクを実行したいというだけの目的で、大量に生成することは避けてください。その目的にはグローバル並列キューが向いています。直列キューを生成する際は、リソースを保護する、アプリケーションの何らかの振る舞いを同期するなど、個々のキューの目的を明確にしてください。

リスト 3-2に、直列キューを生成する手順を示します。dispatch_queue_create関数には引数として、キュー名と、キュー属性の集合を渡します。デバッガや性能計測ツールは、タスクの実行状況を追跡する際に役立つよう、キュー名を表示するようになっています。キュー属性は将来の拡張に備えて予約されているものなので、当面はNULLを指定してください。

リスト 3-2　直列キューの生成

```
dispatch_queue_t queue;
queue = dispatch_queue_create("com.example.MyQueue", NULL);
```

独自に生成するキューに加え、システムが自動的に生成し、アプリケーションのメインスレッドに対応づける直列キューがあります。このキューの取得方法については、“標準的なキューを実行時に取得”（46 ページ）を参照してください。

標準的なキューを実行時に取得

Grand Central Dispatchには、標準的なディスパッチキューに、アプリケーション側からアクセスする関数が用意されています。

- dispatch_get_current_queue関数は、デバッグ用として、あるいは「現在の」キューが何かを調べるために使います。ブロックオブジェクト内から呼び出すと、当該ブロックが登録されている（そしておそらくこのブロックを実行している）キューが返されます。ブロック外から呼び出せば、アプリケーションのデフォルトの並列キューが返されます。
- dispatch_get_main_queue関数は、アプリケーションのメインスレッドに対応づけられた、直列ディスパッチキューを返します。このキューは、Cocoaアプリケーションや、メインスレッド上でdispatch_main関数を呼び出すアプリケーション、あるいは（CFRunLoopRef型またはNSRunLoopオブジェクトで）実行ループを構成するアプリケーションでは、自動的に生成されます。
- dispatch_get_global_queue関数は、共有並列キューを取得するために使います。詳細については、“グローバルな並列ディスパッチキューの取得”（44 ページ）を参照してください。

ディスパッチキューのメモリ管理

ディスパッチキューその他のディスパッチオブジェクトは、参照カウントで管理されるデータ型です。直列ディスパッチキューを生成した時点では、参照カウンタの値は1になっています。この値は、必要に応じ、dispatch_retain関数、dispatch_release関数で増減できます。参照カウンタの値が0になると、システムが非同期的にキューを破棄します。

キューその他のディスパッチオブジェクトは、使用中にもかかわらず破棄されてしまわないようにするため、適切に保持、解放する必要があります。メモリ管理されたCocoaオブジェクトの一般的なルールとして、コードに渡されたキューを使う場合、事前にキューを保持した上で使用し、必要がなくなった時点で解放しなければなりません。この基本パターンを守っていれば、キューの使用中は確実に、メモリ上に置かれたままになっています。

> **注意:** グローバルなディスパッチキュー（並列ディスパッチキュー、メインスレッドに対応づけられたディスパッチキューなど）を保持、解放する必要はありません。しようとしても無視されます。

アプリケーションにガベージコレクションの機構を組み込んだ場合でも、ディスパッチキューその他のディスパッチオブジェクトは、それとは別に保持、解放する必要があります。Grand Central Dispatchは、使っていないメモリを回収する方法として、ガベージコレクションのモデルを採用してはいません。

独自のコンテキスト情報をキューに格納

ディスパッチオブジェクト（ディスパッチキューを含む）はすべて、独自のコンテキストデータを結びつけて格納できます。データの設定/取得には、`dispatch_set_context`関数、`dispatch_get_context`関数をそれぞれ使います。システムがこのデータを使うことはありません。また、適当な時点でデータを割り当て、解放するのは、開発者の責任です。

キューに関しては、コンテキストデータとして、Objective-Cのオブジェクトその他のデータ構造を指すポインタを格納しておけば、キューやその意図する使いかたが識別しやすくなります。キューを破棄する際にコンテキストデータを解放（あるいは関連づけを解除）するためには、ファイナライザ関数を使うとよいでしょう。ファイナライザ関数でコンテキストデータを解放するコード例を、リスト3-3（48 ページ）に示します。

キューのクリーンアップ関数の組み込み

生成した直列ディスパッチキューに、ファイナライザ関数の形で、このキューを破棄する際に実行する独自のクリーンアップ処理を組み込むことができます。ディスパッチキューは参照カウントで管理されるオブジェクトであり、`dispatch_set_finalizer_f`関数で、参照カウンタの値が0に達したときに実行する関数を指定できます。この関数で、キューに結びつけられたコンテキストデータをクリーンアップします。関数が呼び出されるのは、コンテキストポインタが`NULL`でない場合に限ります。

リスト 3-3に、独自に用意したファイナライザ関数と、キューを生成してこのファイナライザ関数を組み込むコード例を示します。キューはファイナライザ関数を使って、キューのコンテキストポインタに格納されたデータを解放します（コードから参照されている`myInitializeDataContextFunction`

関数、myCleanUpDataContextFunction関数は、データ構造体の内容を初期化、クリーンアップするものと想定します）。ファイナライザ関数の引数であるコンテキストポインタには、キューに結びつけられたデータオブジェクトが収容されています。

リスト 3-3　キューのクリーンアップ関数の組み込み

```
void myFinalizerFunction(void *context)
{
    MyDataContext* theData = (MyDataContext*)context;

    // 構造体の内容をクリーンアップ
    myCleanUpDataContextFunction(theData);

    // 構造体自身を解放。
    free(theData);
}

dispatch_queue_t createMyQueue()
{
    MyDataContext*  data = (MyDataContext*) malloc(sizeof(MyDataContext));
    myInitializeDataContextFunction(data);

    // キューを生成しコンテキストデータを結びつけ。
    dispatch_queue_t serialQueue =
dispatch_queue_create("com.example.CriticalTaskQueue", NULL);
    if (serialQueue)
    {
        dispatch_set_context(serialQueue, data);
        dispatch_set_finalizer_f(serialQueue, &myFinalizerFunction);
    }

    return serialQueue;
}
```

タスクをキューに追加

タスクを実行するためには、適切なディスパッチキューにディスパッチする必要があります。ディスパッチする方法には、同期/非同期の2とおりがあるほか、タスクを単独で、あるいはグループにまとめてディスパッチできます。タスクをキューに入れてしまえば、さまざまな制約や、既存のタスクの状況を考慮し、できるだけ早期に実行するのは、キュー側の責任になります。このセクションでは、タスクをキューにディスパッチする方法をいくつか紹介し、それぞれの長所を解説します。

タスクを単独でキューに追加

タスクをキューに追加する方法には、非同期/同期の2とおりがあります。可能な限り、非同期に追加する方法を推奨します。それには`dispatch_async`関数、`dispatch_async_f`関数を使います。ブロックオブジェクトや関数をキューに追加する時点で、実際にいつ実行されるかを予測する手段はありません。したがって、ブロックや関数を非同期に追加すれば、コードの実行をスケジューリングし、そのまま呼び出しスレッド上で別の処理を続行できることになります。この振る舞いが特に重要なのは、（おそらく何らかのユーザイベントに応答して）アプリケーションのメインスレッドからタスクをスケジューリングする場合です。

タスクは可能な限り非同期に追加するべきですが、競合条件その他、同期にまつわるエラーを回避するため、同期的に追加しなければならない場合もあります。その場合は`dispatch_sync`関数、`dispatch_sync_f`関数を使ってください。すると、当該タスクの実行完了まで、現在のスレッドの実行がブロックされます。

> **Important:** `dispatch_sync`、`dispatch_sync_f`関数を、この関数に渡そうとしているのと同じキュー上で動作しているタスクから呼び出してはなりません。特に直列キューの場合、デッドロックを回避するためこれを厳守する必要がありますが、並列キューであってもできるだけ避けてください。

次のコード例では、ブロックの形で定義したタスクを、非同期に、あるいは同期的にディスパッチしています。

```
dispatch_queue_t myCustomQueue;
myCustomQueue = dispatch_queue_create("com.example.MyCustomQueue", NULL);

dispatch_async(myCustomQueue, ^{
    printf("Do some work here.\n");
});

printf("The first block may or may not have run.\n");
```

```
dispatch_sync(myCustomQueue, ^{
    printf("Do some more work here.\n");
});
printf("Both blocks have completed.\n");
```

タスク終了時に完了ブロックを実行

キューに登録されたタスクは、その性質上、当該タスクの生成元コードとは独立して動作します。しかしアプリケーション側では、タスクが終了したらその結果を知りたいので、終了した旨の通知が欲しいことが多いでしょう。従来の非同期プログラミングではコールバック関数の形で実装していましたが、ディスパッチキューでは完了ブロックを使います。

完了ブロックも処理を記述したコード単位で、タスク完了時にキューにディスパッチされます。通常は呼び出し元が用意し、タスク開始時に引数として渡します。タスクのコードが実行しなければならないのは、処理終了時に、所定のブロックまたは関数を、指定されたキューに登録することだけです。

リスト 3-4に、ブロックを使って実装した、平均値を計算する関数の例を示します。平均値関数に渡す、末尾2つの引数で、結果を報告する際に使うキューとブロックを指定します。計算終了後、当該ブロックに結果を渡し、キューに追加するようになっています。キューが誤って解放されてしまわないよう、最初にキューを保持し、完了ブロックをディスパッチした後に解放している点が重要です。

リスト 3-4　タスク終了時に完了ブロックを実行

```
void average_async(int *data, size_t len,
   dispatch_queue_t queue, void (^block)(int))
{
   // 渡されたキューを保持して、
   // 完了ブロックが呼び出される前に、
   // 使えなくなってしまわないようにする。
   dispatch_retain(queue);

   // デフォルトの並列キュー上で処理を行い、
   // 指定されたブロックを、その結果を引数として呼び出す。
   dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0),
^{
      int avg = average(data, len);
      dispatch_async(queue, ^{ block(avg);});
```

```
        // 終了後、キューを解放する
        dispatch_release(queue);
    });
}
```

ループの並列実行

並列ディスパッチキューを使って性能を改善できる箇所のひとつとして、繰り返し回数が固定のループがあります。たとえば、ある処理を繰り返す、次のようなforループがあるとします。

```
for (i = 0; i < count; i++) {
    printf("%u\n",i);
}
```

繰り返しそれぞれの処理が他と独立であり、順序がそれほど重要でないならば、これをdispatch_applyまたはdispatch_apply_f関数の呼び出しに置き換えることができます。この関数は、指定されたブロックまたは関数を、ループの繰り返しごとに、キューに登録します。並列キューにディスパッチすれば、ループの何回分かを同時に実行できることになります。

dispatch_apply関数やdispatch_apply_f関数には、直列キュー、並列キューのどちらでも指定できます。並列キューを渡せばループの何回分かを同時に実行でき、これが最も標準的な使いかたです。直列キューも指定可能で、これが適している場合もありますが、元のループに比べ、性能改善効果はありません。

Important: 普通のforループと同様、dispatch_apply関数やdispatch_apply_f関数も、繰り返しが終わるまでは制御が戻りません。したがって、キューのコンテキストで既に実行中のコードから呼び出す際には、注意が必要です。引数として指定するのが直列キューであって、当該コードを実行しているのと同じものであれば、デッドロックに陥るからです。

実質的に現在のスレッドをブロックすることになるので、メインスレッドからこの関数を呼び出す場合にも注意が必要です。この場合、イベントループが、イベントに迅速に応答できなくなるおそれがあります。ループが相当の処理時間を要するものであれば、別のスレッドからこの関数を呼び出す方がよいでしょう。

リスト 3-5」に、先に挙げたforループをdispatch_applyの構文で置き換えた例を示します。dispatch_apply関数に渡すブロックは、ループの繰り返し回数を表す引数を取るものとします。この値は、ブロックを実行する初回には0、その次の実行時には1となります。最終回にはcount - 1となります（countは反復数）。

リスト 3-5　forループの反復を並列実行

```
dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
 0);


dispatch_apply(count, queue, ^(size_t i) {
   printf("%u\n",i);
});
```

反復されるタスクコードは、ある程度以上の処理を行うものでなければなりません。ブロックや関数をキューにディスパッチすると、一般に、そのスケジューリングにある程度のオーバーヘッドがかかります。ちょっとした処理しか行わないとすれば、オーバーヘッド分がかなりの割合を占め、性能改善効果が損なわれてしまいます。テスト中、このような状況に気がついた場合は、ストライディングという手法で、ループ1回あたりの処理量を増やすとよいでしょう。元のループの何回分かをひとつのブロックにまとめ、それに応じて反復回数を減らすというやり方です。たとえば100回反復するループは、4回分ごとにブロックにまとめると、反復回数が25になります。実装例については"ループコードの改善"（79 ページ）を参照してください。

メインスレッド上でタスクを実行

Grand Central Dispatchには、アプリケーションのメインスレッド上でタスクを実行する、専用のディスパッチキューが組み込まれています。メインスレッド上で（`CFRunLoopRef`型または`NSRunLoop`オブジェクトで）実行ループを構成するアプリケーションの場合、このキューは自動的に生成され、終了時には破棄されます。Cocoaアプリケーション以外で、明示的に実行ループを使わずに実装したい場合は、`dispatch_main`関数で明示的にメインディスパッチキューを破棄する必要があります。タスクをキューに追加することは可能ですが、この関数を呼び出さなければ、実行されることはありません。

アプリケーションのメインスレッドで用いるディスパッチキューは、`dispatch_get_main_queue`関数で取得できます。このキューに追加したタスクは、メインスレッド自身の上で、直列に実行されます。したがってこのキューを、アプリケーションのほかの部分で実行される処理との同期点として使えます。

Objective-Cのオブジェクトをタスクで使用

GCDはCocoaのメモリ管理技術にもはじめから対応しているので、ディスパッチキューに登録するブロック中も、Objective-Cのオブジェクトは自由に使えます。ディスパッチキューはそれぞれ独自に自動解放プールを管理しており、自動解放されたオブジェクトは、ある時点で確実に解放されます。どの時点で解放されるか、については何の保証もありません。

アプリケーションが使用できるメモリ量に制約がある場合、ブロックが自動解放の対象となるオブジェクトを相当数生成するならば、独自の自動解放プールを生成する以外に、オブジェクトを適切な時点で確実に解放する手段はありません。ブロック内で数百個以上のオブジェクトを生成するのであれば、自動解放プールを複数用意するか、一定期間ごとにプールを破棄する必要があるかもしれません。

自動解放プールおよびObjective-Cのメモリ管理機構については、『*Advanced Memory Management Programming Guide*』を参照してください。

キューの一時停止と再開

キューを一時停止することにより、一時的にブロックオブジェクトの実行を停めることができます。ディスパッチキューの一時停止には`dispatch_suspend`関数、再開には`dispatch_resume`関数を使います。`dispatch_suspend`を呼び出すと、キューの「一時停止」状態を表す参照カウンタの値が増え、`dispatch_resume`を呼び出すと減ります。参照カウンタの値が0にならない限り、キューは一時停止したままになります。したがって、一時停止と再開の関数を対にして呼び出すようにしないと、想定どおりにブロックの処理が再開されません。

Important: 一時停止や再開の呼び出しは非同期であり、ブロックの実行中には効果がありません。キューを一時停止しても、既に実行中のブロックが停止するわけではないのです。

ディスパッチセマフォを使って有限のリソースの使いかたを規制

ディスパッチキューに登録されているタスクが、有限のリソースにアクセスする場合、ディスパッチセマフォを使って、同時にアクセスするタスク数を規制するとよいかもしれません。ディスパッチセマフォの動作は、一般的なセマフォと同じですが、ひとつ違いがあります。リソースが使用可能であれば、従来型のシステムセマフォに比べ、短時間でディスパッチセマフォを取得できる、という点です。これは、Grand Central Dispatchがこの場合に、カーネルの機能を使わないからです。カーネル機能を呼び出す必要があるのは、リソースが使えない状態で、これが明け渡される（セマフォにV操作が行われる）まで、スレッドを停める必要がある場合に限ります。

ディスパッチセマフォを使う手順を以下に示します。

1. セマフォを（`dispatch_semaphore_create`関数で）生成する際、使用可能なリソースの個数（正の整数）を初期値として指定できます。
2. 各タスクは、リソースを使う必要が生じると、セマフォに対して`dispatch_semaphore_wait`を呼び出して要求を送り、待機します（P操作）。

3. 制御が返ってきたら、リソースを獲得して処理を行います。
4. 処理が終了したらリソースを解放し、その旨をdispatch_semaphore_signal関数でセマフォに伝えます（V操作）。

この手順でうまく処理できる例として、ファイル記述子というリソースを考えてみましょう。使用可能なファイル記述子の数は、アプリケーションごとに決まっています。大量のファイルを処理するタスクであっても、この数を超えるファイルを同時に開くことはできません。そこで、セマフォを利用して、ファイル処理コードで同時に使うファイル記述子の数を制限します。これを実装するコードの基本部分を以下に示します。

```
// プールの初期サイズを指定してセマフォを生成
dispatch_semaphore_t fd_sema = dispatch_semaphore_create(getdtablesize() / 2);

// ファイル記述子が使用可能になるまで待機
dispatch_semaphore_wait(fd_sema, DISPATCH_TIME_FOREVER);
fd = open("/etc/services", O_RDONLY);

// 終了したらファイル記述子を解放
close(fd);
dispatch_semaphore_signal(fd_sema);
```

セマフォを生成する際には、使用可能なリソースの数を指定します。これがセマフォのカウント変数の初期値になります。リソースが必要になると、セマフォに要求を送って（dispatch_semaphore_wait関数を呼び出して）待機します。この関数は、カウント変数の値を1減らします。結果が負になると、カーネルに対し、スレッドをブロックするよう要求するようになっています。リソースを使い終わると、dispatch_semaphore_signal関数を呼び出します。この関数はカウント変数を1増やします。リソースが使えるようになるのを待機している（ブロックされた）タスクがあれば、そのうちのひとつがブロックを解除されます。

キューに追加されたタスクのグループを待機

ディスパッチグループは、一連のタスクがすべて実行完了するまで、スレッドをブロックする手段となります。この振る舞いは、所定のタスクがすべて完了するまで、処理を進めることができない場合に有用です。たとえば、何らかのデータを計算するタスクをいくつかディスパッチした後、すべて完了するまで待ち、その結果をもとに処理を進める、といった使いかたが考えられます。あるいは、ス

レッドの合流操作に代わる機能としても使えます。子スレッドをいくつか起動し、それぞれを合流する代わりに、対応するタスクをそれぞれディスパッチグループに追加し、グループ全体が終了するまで待機する、ということです。

リスト3-6に、グループをセットアップし、タスクをディスパッチし、結果が得られるまで待機する、基本的な流れを示します。dispatch_async関数でタスクをキューにディスパッチする代わりに、dispatch_group_async関数を使っています。この関数は、タスクをグループに関連づけた後、キューに追加して実行を委ねます。次に、タスクグループが完了するまで待機するため、このグループを引数としてdispatch_group_wait関数を呼び出しています。

リスト 3-6　非同期タスクの待機

```
dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
 0);
dispatch_group_t group = dispatch_group_create();


// タスクをグループに追加
dispatch_group_async(group, queue, ^{
    // 何らかの非同期処理
});


// タスク実行中にもほかの処理を実行。


// 処理をこれ以上続行できない場合は、
// グループ上で待機して、現在のスレッドをブロックする。
dispatch_group_wait(group, DISPATCH_TIME_FOREVER);


// 必要なくなったのでグループを解放する。
dispatch_release(group);
```

ディスパッチキューとスレッドの安全性

ディスパッチキューの話題でスレッドの安全性について触れるのは奇妙に思われるかもしれませんが、実は密接に関連する話です。アプリケーションに並列性を組み込む際、常に意識しておくべき事項がいくつかあります。

- ディスパッチキューそのものはスレッドセーフです。すなわち、システム上のどのスレッドからでも、タスクをディスパッチキューに登録できます。事前にロックしたり、キューへのアクセスを同期したりする必要はありません。
- `dispatch_sync`関数を呼び出す場合、引数として渡したキュー上で動作しているタスクから呼び出す形であってはなりません。キューがデッドロックに陥るおそれがあります。現在のキューにディスパッチする必要がある場合は、`dispatch_async`関数で非同期に追加してください。
- ディスパッチキューに登録するタスクからロックをかけることは避けてください。タスクからロックをかけること自体は安全ですが、ロックを獲得する際、当該ロックが使用できない状態であれば、直列キュー全体がブロックされるおそれがあります。並列キューの場合も同様に、ロックを待機していると、代わりにほかのタスクが実行されなくなることがあります。コードのある箇所と別の箇所とを同期する必要がある場合は、ロックではなく直列ディスパッチキューを使ってください。
- タスクの動作基盤であるスレッドについても、各種の情報を取得できますが、それは避けた方がよいでしょう。ディスパッチキューとスレッドの互換性については、“POSIXスレッドとの互換性” （83 ページ）を参照してください。

スレッドを使って実装したコードをディスパッチキューで書き換えるためのヒントは、“スレッドを使ったコードからの移行” （75 ページ）を参照してください。

ディスパッチソース

タスクが動作基盤としているシステムとのやり取りには、無視できないほどの時間が取られうることを予期しておかなければなりません。カーネルその他のシステム階層に属する機能を呼び出せば、コンテキストの切り替えが発生することもあって、アプリケーションプロセス内に閉じた呼び出しに比べて相応のコストがかかります。そのため、多くのシステムライブラリは、システムに対して要求を登録した後、処理終了まで別の処理を続行できるように、非同期インターフェイスを提供しています。Grand Central Dispatchは、これと同じような振る舞いになるよう実装されているので、ブロックとディスパッチキューを使って要求を登録して、その後は別の処理を続行し、処理が終了してから結果を受け取ることができます。

ディスパッチソースについて

ディスパッチソースは、ある低レベルのシステムイベントの処理を仲介する基本的なデータ型です。Grand Central Dispatchは次のような型のディスパッチソースに対応しています。

- **タイマーディスパッチソース**は、定期的に通知を生成します。
- **シグナルディスパッチソース**は、UNIXシグナルが届いたときに通知します。
- **記述子ディスパッチソース**は、ファイルやソケットが次のような状況になった場合に通知します。
    - データを読み取れるようになったとき
    - データを書き込めるようになったとき
    - ファイルシステム上で、ファイルが削除、移動、改名されたとき
    - ファイルのメタ情報が変化したとき
- **プロセスディスパッチソース**は、プロセスに関する、次のようなイベントを通知します。
    - プロセスが終了したとき
    - プロセスが`fork`や`exec`などのシステムコールを呼び出したとき
    - シグナルがプロセスに送信されたとき
- **Machポートディスパッチソース**はMachに関するイベントを通知します。
- **カスタムディスパッチソース**は、開発者が定義し、トリガをかけるものです。

ディスパッチソースは、システム関係のイベント処理によく使われる、非同期コールバック関数の代替となるものです。ディスパッチソースを生成した後、監視対象のイベント、ディスパッチキュー、イベントを処理するコードを設定します。このコードは、ブロックオブジェクトまたは関数の形で定義します。該当するイベントが届くと、ディスパッチソースはこのブロックまたは関数を所定のディスパッチキューに登録し、実行を委ねます。

キューに直接登録するタスクと違い、いったん設定してしまえば、随時発生する継続的なイベントソースとして振る舞います。明示的に解除しない限り、ディスパッチソースは、あるディスパッチキューに結びつけられた状態になっています。その間は、該当するイベントが発生する都度、対応するタスクコードをディスパッチキューに登録します。タイマーのように一定期間ごとに発生するイベントもありますが、多くは、ある条件が満たされたとき、突発的に発生します。そのため、ディスパッチソースは、自分自身に結びつけられたディスパッチキューを保持して、未処理のイベントが残っているかも知れない間は、誤って解放されてしまわないようにしています。

ディスパッチキューにイベントが滞留し、処理しきれなくなってしまわないよう、ディスパッチソースには、イベントを「併合」する仕組みが実装されています。あるイベントをキューから取り出して処理する前に次のイベントが届いた場合、旧イベントのデータに新イベントのデータを併合して、2つのイベントを順に処理した場合と実質的に同じ結果になるような、1つのイベントにまとめてしまうのです。イベントの型によって、旧イベントを完全に新イベントで置き換えてしまう場合や、保持する情報を更新する場合があります。たとえばシグナルディスパッチソースの場合、最新のシグナルに関するデータのみを残します。ただし、付加情報として、イベントハンドラを最後に起動して以降に送信された、シグナルの個数も報告します。

ディスパッチソースの生成

ディスパッチソースの生成処理では、イベントのソースとディスパッチソース自身の、両方を生成します。イベントのソースとは、イベントの処理に必要となる、OSレベルのデータ構造のことです。たとえば記述子ディスパッチソースの場合は、ファイルやソケットをオープンして記述子を取得します。プロセスディスパッチソースであれば、対象プログラムのプロセスIDを取得することになります。イベントソースを取得すれば、対応するディスパッチソースを、次の手順で生成できます。

1. ディスパッチソースを、`dispatch_source_create`関数で生成します。
2. ディスパッチソースについて、次のような設定をします。
    - イベントハンドラをディスパッチソースに組み込みます（“イベントハンドラの記述と組み込み”（59 ページ）を参照）。
    - タイマーディスパッチソースの場合、タイマー情報を`dispatch_source_set_timer`関数で設定します（“タイマーの生成”（64 ページ）を参照）。
3. 必要ならば、キャンセルハンドラをディスパッチソースに組み込みます（“キャンセルハンドラの組み込み”（62 ページ）を参照）。

4. dispatch_resume関数を呼び出して、イベントの処理を開始します（"ディスパッチソースの一時停止と再開"（74 ページ）を参照）。

ディスパッチソースは、実際に使う前にいくつか設定が必要なので、dispatch_source_create関数は、生成したディスパッチソースを一時停止状態にして返すようになっています。この状態では、イベントを受け取るだけで、それ以上の処理はしません。この間に、イベントハンドラを組み込み、イベント処理に必要な設定をします。

以降の各セクションでは、ディスパッチソースの各種設定について説明します。ディスパッチソースの型に応じて必要な設定をする詳しい例は、"ディスパッチソースの例"（64 ページ）を参照してください。ディスパッチソースの生成および設定に用いる関数については、『*Grand Central Dispatch (GCD) Reference*』を参照してください。

イベントハンドラの記述と組み込み

ディスパッチソースが生成するイベントを処理するためには、イベントハンドラを定義する必要があります。イベントハンドラは関数またはブロックオブジェクトの形で記述し、dispatch_source_set_event_handler関数またはdispatch_source_set_event_handler_f関数でディスパッチソースに組み込みます。イベントが届くと、ディスパッチソースはこのイベントハンドラを、所定のディスパッチキューに登録して処理を委ねます。

イベントハンドラの本体では、届いたイベントは何であれ処理しなければなりません。イベントハンドラがキューに追加され、イベント処理を待機している状態で、新たにイベントが届いた場合、ディスパッチソースは2つのイベントを併合します。そのため、イベントハンドラは一般に、最新のイベントのみを処理することになります。もっとも、ディスパッチソースの型によっては、併合されたほかのイベントの情報も取得できることがあります。一方、イベントハンドラの処理が始まってから、新たにイベントが届いた場合は、その処理が終了するまで、ディスパッチソースがそのまま保持しています。処理が終了してから、新しいイベントを処理するイベントハンドラを、改めてキューに登録します。

関数ベースのイベントハンドラは、引数として、ディスパッチソースオブジェクトを指すコンテキストポインタを1つ取り、値は返しません。ブロックベースのイベントハンドラは、引数も戻り値もありません。

```
// ブロックベースのイベントハンドラ
void (^dispatch_block_t)(void)


// 関数ベースのイベントハンドラ
void (*dispatch_function_t)(void *)
```

イベントハンドラでは、イベントに関する情報を、ディスパッチソース自身から取得できます。関数ベースのイベントハンドラの場合、引数としてディスパッチソースを指すポインタが渡されます。一方、ブロックベースのイベントハンドラの場合、自分自身でこのポインタを獲得しなければなりません。と言っても、ディスパッチソースを収容した変数を、単に参照するだけのことです。たとえば次のコード断片では、変数sourceを参照していますが、これはブロックのスコープ外で宣言されたものです。

```
dispatch_source_t source = dispatch_source_create(DISPATCH_SOURCE_TYPE_READ,
                                  myDescriptor, 0, myQueue);
dispatch_source_set_event_handler(source, ^{
   // データを変数sourceから取得する。
   // この変数は親コンテキスト（ブロックの外）で宣言されたもの。
   size_t estimated = dispatch_source_get_data(source);

   // 記述子から読み込みを続行...
});
dispatch_resume(source);
```

ブロック内からブロック外の変数にアクセスできるため、非常に柔軟な記述が可能です。もちろんこの変数は、ブロック内では読み込み専用です。条件によってはブロック外の変数を更新することも可能ですが、ディスパッチソースに結びつけられたイベントハンドラでは、できないと考えた方がよいでしょう。ディスパッチソースはイベントハンドラを非同期に実行するので、実際に実行される時点までに、ブロック外の変数の定義スコープが消えてしまっている可能性があります。ブロック内から外部の変数にアクセスする方法については、『*Blocks Programming Topics*』を参照してください。

表 4-1に、イベントハンドラから呼び出して、イベントに関する情報を取得する関数を示します。

表 4-1 ディスパッチソースからデータを取得する関数

関数	解説
`dispatch_source_-get_handle`	この関数は、ディスパッチソースが管理する、イベントのソース（OSレベルのデータ構造）を返します。 記述子ディスパッチソースの場合、ディスパッチソースに対応する（ファイルやソケットの）記述子を表す、`int`型の値を返します。 シグナルディスパッチソースの場合、直近のシグナル番号を表す、`int`型の値を返します。 プロセスディスパッチソースの場合、監視対象プロセスを表す、`pid_t`型のデータ構造体を返します。 Machポートディスパッチソースの場合、`mach_port_t`型のデータ構造体を返します。 その他のディスパッチソースの場合、この関数の戻り値は未定義です。
`dispatch_source_-get_data`	イベントに関連づけられた、未処理になっているデータに関する情報を返します。 ファイルからデータを読み込む記述子ディスパッチソースの場合、読み込み可能なバイト数を返します。 ファイルにデータを書き込む記述子ディスパッチソースの場合、書き込める空間があるならば正の整数を返します。 ファイルシステムの動作を監視する記述子ディスパッチソースの場合、発生したイベントの型を表す定数を返します。どのような定数があるか、については、列挙型`dispatch_source_vnode_flags_t`を調べてください。 プロセスディスパッチソースの場合、発生したイベントの型を表す定数を返します。どのような定数があるか、については、列挙型`dispatch_source_proc_flags_t`を調べてください。 Machポートディスパッチソースの場合、発生したイベントの型を表す定数を返します。どのような定数があるか、については、列挙型`dispatch_source_machport_flags_t`を調べてください。 カスタムディスパッチソースの場合、既存のデータから生成された新しいデータ値と、`dispatch_source_merge_data`関数に渡される新しいデータを返します。

関数	解説
dispatch_source_-get_mask	ディスパッチソースの生成時に使われたイベントフラグを返します。 プロセスディスパッチソースの場合、ディスパッチソースが受信するイベントのマスクを返します。どのような定数があるか、については、列挙型dispatch_source_proc_flags_tを調べてください。 送信権限のあるMachポートディスパッチソースの場合、要求されたイベントのマスクを返します。どのような定数があるか、については、列挙型dispatch_source_mach_send_flags_tを調べてください。 カスタムORディスパッチソースの場合、データを併合するために用いるマスクを返します。

ディスパッチソースの型に応じたイベントハンドラを記述し、組み込む例については、“ディスパッチソースの例”（64 ページ）を参照してください。

キャンセルハンドラの組み込み

キャンセルハンドラは、ディスパッチソースを解放する前のクリーンアップ処理に使います。大部分の型のディスパッチソースでは、キャンセルハンドラはなくても構いません。必要なのは、何らかのデータの更新処理を組み込んでいて、解放時にも更新しなければならない場合に限ります。もっとも、記述子ディスパッチソースの場合は記述子をクローズするため、Machポートディスパッチソースの場合はMachポートを解放するために、キャンセルハンドラを用意する必要があります。これを怠ると、コード中あるいはシステムのほかの箇所で、記述子やMachポートが誤って再利用され、微妙なバグが入り込むおそれがあります。

キャンセルハンドラはいつ組み込んでも構いませんが、通常はディスパッチソースの生成時に行います。組み込みには、ブロックオブジェクトの形で実装した場合はdispatch_source_set_cancel_handler関数、関数の形で実装した場合はdispatch_source_set_cancel_handler_f関数を使います。ディスパッチソース用にオープンした記述子をクローズする、簡単なキャンセルハンドラの例を示します。fdは記述子を表す、ブロック外の変数です。

```
dispatch_source_set_cancel_handler(mySource, ^{
    close(fd); // 先にオープンしたファイル記述子をクローズする。
});
```

キャンセルハンドラを組み込んだディスパッチソースの完全なコード例は、“記述子からのデータ読み込み”（66 ページ）を参照してください。

ターゲットキューの変更

ディスパッチソースを生成する際には、イベントを実行するキューと、キャンセルハンドラを指定しますが、このキューはいつでも、`dispatch_set_target_queue`関数で変更できます。これを利用して、ディスパッチソースのイベントを処理する優先度を変更することも可能です。

ディスパッチソースのキューを変更する操作は非同期のオペレーションであり、ディスパッチソースは、できるだけ迅速に変更しようと試みます。イベントハンドラが既にキューに入っており、処理を待っている状態であれば、変更前のキュー上で実行されます。しかし、変更した時刻の前後に届いたイベントが、どちらのキューで処理されるかは不定です。

カスタムデータをディスパッチソースに関連づけ

Grand Central Dispatchにおけるほかの多くのデータ型と同様、`dispatch_set_context`関数を使うと、カスタムデータもディスパッチソースに関連づけることができます。コンテキストポインタを使って、イベントハンドラがイベントの処理に必要とするデータを格納しておけます。カスタムデータをコンテキストポインタに格納した場合、キャンセルハンドラを組み込んで（“キャンセルハンドラの組み込み”（62 ページ）を参照）、ディスパッチソースが必要としなくなった時点でデータを解放する必要があります。

イベントハンドラをブロックの形で実装した場合、ブロック外の局所変数にもアクセスできます。これを利用すれば、ディスパッチソースのコンテキストポインタにデータを格納しなくても済むかもしれませんが、この機能は慎重に使うようにしてください。ディスパッチソースはアプリケーション中に長期にわたって生存するので、変数の値がポインタである場合、そのアクセスには注意が必要です。ポインタが指すデータがいつでも解放されうるとすれば、別にコピーしておくか、適切に保持して、解放されないようにしなければなりません。いずれにしても、キャンセルハンドラを用意して、用が済んだらデータを解放しなければならないでしょう。

ディスパッチソースのメモリ管理

ほかのディスパッチオブジェクトと同様、ディスパッチソースも参照カウントで管理されます。生成時の参照カウンタ値は1です。`dispatch_retain`および`dispatch_release`関数で保持、解放できます。参照カウンタが0になると、システムは自動的に、ディスパッチソースのデータ構造体を解放します。

ディスパッチソースの所有権は、使いかたによって、ディスパッチソース自身に持たせることも、外部で管理することも可能です。外部で管理する場合、ほかのオブジェクトまたはコード単位がディスパッチソースを所有し、必要なくなった時点で解放する責任を負います。内部的に管理する場合、ディスパッチソースは自分自身を所有し、適切な時点で解放します。外部で管理する方式が一般的ですが、自律的なディスパッチソースを生成し、外部とのやり取りなしでコードの振る舞いを管理したい場合は、内部的に管理するとよいでしょう。たとえばディスパッチソースを、単一のグローバルイベントにのみ応答するよう設計する場合、そのイベントを処理して直ちに終了することになります。

ディスパッチソースの例

以降の各セクションでは、典型的なディスパッチソースを生成、設定する方法を示します。個々の型のディスパッチソースの設定については、『*Grand Central Dispatch (GCD) Reference*』を参照してください。

タイマーの生成

タイマーディスパッチソースは、一定時間間隔でイベントを生成します。定期的に実行するタスクを起動するために使えます。たとえば、ゲームその他のグラフィックを多用するアプリケーションでは、画面やアニメーションの更新にタイマーを使えるかもしれません。また、頻繁に更新されるサーバについて、新しい情報がないか定期的に確認するためにも、タイマーを設定し、生成されるイベントを利用するとよいでしょう。

タイマーディスパッチソースはすべてインターバルタイマーであり、いったん生成すれば、指定した間隔で定期的にイベントを送信し続けます。生成時に指定する値として、許容差があります。システムはこれを参照して、タイマーイベントを生成する時間の精度を調整します。許容差が大きければ、電源を管理し、コアをスリープ状態から復帰する処理が、制御しやすくなります。たとえばシステムは、タイマーの発動時刻を若干前後して、ほかのシステムイベントと揃えようとするかもしれません。したがって、可能であれば許容差を指定するべきでしょう。

> **注意:** 許容差を0としても、ナノ秒単位の精度が確保されるとは期待しないでください。システムはできるだけ指定に従おうとしますが、精度は保証できません。

コンピュータがスリープ状態になると、タイマーディスパッチソースはすべて一時停止します。スリープ状態が解除されれば、タイマーディスパッチソースも自動的に再開します。タイマーの設定にもよりますが、このように一時停止すれば、次に発動する時刻にも影響が現れます。タイマーディスパッチソースのセットアップに、`dispatch_time`関数または`DISPATCH_TIME_NOW`定数を使った場合、発動時刻の判断には、デフォルトのシステムクロックが使われます。しかしこのクロックは、スリープ状態の間は進みません。これに対し、`dispatch_walltime`関数でセットアップした場合、タイマーディスパッチソースは実時間に従って発動時刻を追跡します。これは発動間隔が比較的長い場合に向いています。イベント時刻間の揺らぎが大きくならないからです。

リスト4-1に、30秒ごと（許容差は1秒）に発動するタイマーの例を示します。タイマー間隔が比較的長いので、`dispatch_walltime`関数で生成しています。起動した直後に初回の発動が起こり、その後は30秒ごとに発動を繰り返します。`MyPeriodicTask`および`MyStoreTimer`は、タイマーの振る舞いを実装するカスタム関数、アプリケーション側の適当な箇所にタイマーを格納するカスタム関数を表します。

リスト 4-1　タイマーディスパッチソースの生成

```
dispatch_source_t CreateDispatchTimer(uint64_t interval,
              uint64_t leeway,
              dispatch_queue_t queue,
              dispatch_block_t block)
{
   dispatch_source_t timer = dispatch_source_create(DISPATCH_SOURCE_TYPE_TIMER,
                                                    0, 0, queue);
   if (timer)
   {
     dispatch_source_set_timer(timer, dispatch_walltime(NULL, 0), interval, leeway);
      dispatch_source_set_event_handler(timer, block);
      dispatch_resume(timer);
   }
   return timer;
}

void MyCreateTimer()
{
   dispatch_source_t aTimer = CreateDispatchTimer(30ull * NSEC_PER_SEC,
                               1ull * NSEC_PER_SEC,
                               dispatch_get_main_queue(),
                               ^{ MyPeriodicTask(); });

   // 後の使用に備えて適当な箇所に格納しておく。
    if (aTimer)
    {
        MyStoreTimer(aTimer);
    }
}
```

タイマーベースのイベントを受け取る方法としては、タイマーディスパッチソースを使うのが標準的ですが、ほかの選択肢もあります。所定の時間後に1回だけブロックを実行したいのであれば、`dispatch_after`関数や`dispatch_after_f`関数が使えます。`dispatch_async`関数とほぼ同じ動作ですが、ブロックをキューに登録する時刻を指定できます。この時刻は、（現在時刻からの）相対値でも、絶対値でも構いません。

記述子からのデータ読み込み

ファイルやソケットからデータを読み込むためには、これをオープンし、`DISPATCH_SOURCE_TYPE_READ`型のディスパッチソースを生成する必要があります。イベントハンドラには、記述子から内容を読み込み、処理する機能を実装します。ファイルの場合、ここでファイルデータ（またはその一部）を読み込み、適当なデータ構造体を生成して格納し、アプリケーションに渡します。ネットワークソケットの場合は、新たに受信したネットワークデータを処理することになります。

データを読み込む際は、記述子を非ブロッキングモードに設定しなければなりません。`dispatch_source_get_data`関数で、読み込めるデータの量を調べることができますが、この値は、実際にデータを読み込むまでの間に変わってしまうことがあります。ブロッキングモードになっていると、ファイルが切り詰められた、あるいはネットワークエラーが発生したとき、イベントハンドラの処理中に読み込みが停止（ストール）してしまい、ディスパッチキューがほかのタスクをディスパッチできなくなるおそれがあります。その結果、直列キューであればデッドロックに陥り、並列キューの場合でも、起動できる新しいタスクの個数が減ってしまうのです。

リスト 4-2に、ファイルからデータを読み込むよう、ディスパッチソースを設定する例を示します。この例でイベントハンドラは、指定されたファイルの内容をすべてバッファに読み込み、（別途定義する）カスタム関数を呼び出してデータを処理しています（この関数の呼び出し元では、読み込み終了後、戻り値のディスパッチソースを使ってキャンセルします）。読み込むデータがない場合でもディスパッチキューがブロックしてしまわないよう、この例では`fcntl`関数を使って、ファイル記述子を非ブロッキングモードに設定しています。ディスパッチソースに組み込まれたキャンセルハンドラは、読み込み終了後、ファイル記述子を確実にクローズします。

リスト 4-2　ファイルからのデータ読み込み

```
dispatch_source_t ProcessContentsOfFile(const char* filename)
{
    // ファイルを読み込み用にオープンする。
    int fd = open(filename, O_RDONLY);
    if (fd == -1)
        return NULL;
    fcntl(fd, F_SETFL, O_NONBLOCK);  // 非ブロッキングモードにする
```

```
   dispatch_queue_t queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
   dispatch_source_t readSource = dispatch_source_create(DISPATCH_SOURCE_TYPE_READ,
                                    fd, 0, queue);
   if (!readSource)
   {
      close(fd);
      return NULL;
   }

   // イベントハンドラを組み込む
   dispatch_source_set_event_handler(readSource, ^{
      size_t estimated = dispatch_source_get_data(readSource) + 1;
      // データをテキストバッファに読み込む。
      char* buffer = (char*)malloc(estimated);
      if (buffer)
      {
         ssize_t actual = read(fd, buffer, (estimated));
         Boolean done = MyProcessFileData(buffer, actual);  // データを処理する

         // 終了後、バッファを解放する。
         free(buffer);

         // これ以上データがなければ、ディスパッチソースをキャンセルする。
         if (done)
            dispatch_source_cancel(readSource);
      }
    });

   // キャンセルハンドラを組み込む
   dispatch_source_set_cancel_handler(readSource, ^{close(fd);});

   // ファイルの読み込みを開始する。
   dispatch_resume(readSource);
```

```
    return readSource;
}
```

この例のカスタム関数MyProcessFileDataは、必要なファイルデータをすべて読み込んでしまい、ディスパッチソースをキャンセルしても構わない状態になったかどうかを判断します。デフォルトでは、記述子からデータを読み込むディスパッチソースは、読み込むべきデータが残っている間、イベントハンドラを繰り返しスケジューリングするようになっています。ソケット接続がクローズされたり、ファイルの末尾に到達したりすると、ディスパッチソースは自動的に、イベントハンドラのスケジューリングを停止します。ディスパッチソースが必要ないとわかっているならば、直接キャンセルしてしまっても構いません。

記述子へのデータ書き込み

データをファイルやソケットにデータを書き込む処理も、読み込む場合とよく似ています。記述子を書き込み用に設定した後、DISPATCH_SOURCE_TYPE_WRITE型のディスパッチソースを生成します。ディスパッチソースを生成してしまえば、システム側からイベントハンドラが呼び出されるので、そこでファイルやソケットにデータを書き込めます。書き込み終了後、dispatch_source_cancel関数でディスパッチソースをキャンセルしてください。

データを書き込む際は、記述子を非ブロッキングモードに設定しなければなりません。dispatch_source_get_data関数で、書き込み可能な容量を調べることはできますが、その値は参考程度であって、実際に書き込む時点までの間に変わっていることもありえます。ブロッキングもーどになっていると、エラーが発生したとき、イベントハンドラの処理中に書き込みが停止（ストール）してしまい、ディスパッチキューがほかのタスクをディスパッチできなくなるおそれがあります。その結果、直列キューであればデッドロックに陥り、並列キューの場合でも、起動できる新しいタスクの個数が減ってしまうのです。

リスト4-3に、ディスパッチソースを使ってファイルにデータを書き込む、基本的な流れを示します。ファイルを生成し、そのファイル記述子をイベントハンドラに渡しています。ファイルに書き込むデータはMyGetData関数で取得します。この関数に、書き込むべきデータを生成する処理を記述してください。書き込み終了後、イベントハンドラはディスパッチソースをキャンセルして、再び呼び出されることがないようにします。ディスパッチソースは、その所有者が解放する責任を負います。

リスト 4-3　ファイルへのデータ書き込み

```
dispatch_source_t WriteDataToFile(const char* filename)
{
    int fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC,
                  (S_IRUSR | S_IWUSR | S_ISUID | S_ISGID));
```

```
    if (fd == -1)
        return NULL;
    fcntl(fd, F_SETFL); // 書き込み中はブロック。

    dispatch_queue_t queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
    dispatch_source_t writeSource =
dispatch_source_create(DISPATCH_SOURCE_TYPE_WRITE,
                           fd, 0, queue);
    if (!writeSource)
    {
        close(fd);
        return NULL;
    }

    dispatch_source_set_event_handler(writeSource, ^{
        size_t bufferSize = MyGetDataSize();
        void* buffer = malloc(bufferSize);

        size_t actual = MyGetData(buffer, bufferSize);
        write(fd, buffer, actual);

        free(buffer);

        // 終了後、ディスパッチソースをキャンセルして解放。
        dispatch_source_cancel(writeSource);
    });

    dispatch_source_set_cancel_handler(writeSource, ^{close(fd);});
    dispatch_resume(writeSource);
    return (writeSource);
}
```

ファイルシステムオブジェクトの監視

ファイルシステムオブジェクトの状態変化を監視するため、DISPATCH_SOURCE_TYPE_VNODE型のディスパッチソースをセットアップします。ファイルが削除、書き込み、改名されたときに、通知を受け取ることができます。また、ある種のメタ情報（ファイル長、リンク数など）が変化したときにも、警告が届きます。

> **注意:** ディスパッチソースに指定するファイル記述子は、イベント処理中、オープンしたままになっていなければなりません。

リスト 4-4に、ファイルの改名を監視し、何らかの処理を行う例を示します（「何らかの処理」は、この例で呼び出されている、MyUpdateFileName関数に実装するものとします）。この記述子はディスパッチソースでしか使わないので、もう使わなくなった時点でクローズする、キャンセルハンドラを組み込んであります。この例で生成するファイル記述子は、基盤となるファイルシステムのオブジェクトに対応するものなので、同じディスパッチソースで、何回改名されてもその都度検出できます。

リスト 4-4　ファイルの改名を監視

```
dispatch_source_t MonitorNameChangesToFile(const char* filename)
{
    int fd = open(filename, O_EVTONLY);
    if (fd == -1)
        return NULL;

    dispatch_queue_t queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
    dispatch_source_t source = dispatch_source_create(DISPATCH_SOURCE_TYPE_VNODE,
                fd, DISPATCH_VNODE_RENAME, queue);
    if (source)
    {
        // 後の使用に備えてファイル名をコピーしておく。
        int length = strlen(filename);
        char* newString = (char*)malloc(length + 1);
        newString = strcpy(newString, filename);
        dispatch_set_context(source, newString);
```

```
        // 改名を処理するイベントハンドラを組み込む
        dispatch_source_set_event_handler(source, ^{
              const char*  oldFilename = (char*)dispatch_get_context(source);
              MyUpdateFileName(oldFilename, fd);
        });

        // 記述子およびファイル名を保存した文字列を解放する
        // キャンセルハンドラを組み込む。
        dispatch_source_set_cancel_handler(source, ^{
            char* fileStr = (char*)dispatch_get_context(source);
            free(fileStr);
            close(fd);
        });

        // イベント処理を開始する。
        dispatch_resume(source);
    }
    else
        close(fd);

    return source;
}
```

シグナルの監視

UNIXシグナルを使えば、ドメイン外からアプリケーションを操作できます。アプリケーションは、致命的エラー（不正な命令など）や、重要な情報の通知（子プロセスの終了など）といった、各種のシグナルを受信できます。従来は、`sigaction`関数を使ってシグナルハンドラを組み込み、シグナルを受け取ったらできるだけ即座に、同期的に処理していました。シグナルが届いた旨を知れば充分で、実際に処理する必要はないのであれば、シグナルディスパッチソースを使って非同期的に処理できます。

シグナルディスパッチソースは、`sigaction`関数で組み込む、同期的なシグナルハンドラに代わるものではありません。同期的なシグナルハンドラは、実際にシグナルを捕捉し、アプリケーションが停止しないようにすることができます。これに対し、シグナルディスパッチソースができるのは、シグナルが届いた旨を通知することだけです。しかも、あらゆる型のシグナルを受け取れるわけではありません。特に`SIGILL`、`SIGBUS`、`SIGSEGV`の各シグナルは、監視の対象外です。

シグナルディスパッチソースは、ディスパッチキュー上で非同期に実行されるため、同期的なシグナルハンドラでは避けられない制約のいくつかを回避できます。たとえば、シグナルディスパッチソースのイベントハンドラからは、どのような関数でも呼び出せます。その代償として、シグナルが届いてから、ディスパッチソースのイベントハンドラが呼び出されるまでに、若干の遅延が生じることがあります。

リスト4-5に、SIGHUPシグナルを処理するよう、シグナルディスパッチソースを設定する例を示します。ディスパッチソースのイベントハンドラは、MyProcessSIGHUP関数を呼び出します。実際にはここに、シグナルを処理するコードを実装することになります。

リスト 4-5　シグナルを監視するブロックの組み込み

```
void InstallSignalHandler()
{
   // シグナルが届いてもアプリケーションが停止しないようにする。
   signal(SIGHUP, SIG_IGN);


   dispatch_queue_t queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
   dispatch_source_t source = dispatch_source_create(DISPATCH_SOURCE_TYPE_SIGNAL,
 SIGHUP, 0, queue);


   if (source)
   {
      dispatch_source_set_event_handler(source, ^{
         MyProcessSIGHUP();
      });


      // シグナルの処理を開始する
      dispatch_resume(source);
   }
}
```

カスタムフレームワーク用のコードを開発する場合、シグナルディスパッチソースを使えば、これにリンクされたアプリケーションとは独立にシグナルを監視できる、という利点があります。ほかのディスパッチソースや同期シグナルハンドラがアプリケーションに組み込まれていても、干渉することはないのです。

同期シグナルハンドラの実装、シグナル名のリストについては、`signal`のmanページを参照してください。

プロセスの監視

プロセスディスパッチソースを使えば、あるプロセスの振る舞いを監視し、適切に応答できます。親プロセスはこれを使って、自身が生成した子プロセスの実行状況を監視できます。たとえば子プロセスの終了を監視する、という使いかたが可能です。同様に、子プロセスも親プロセスを監視し、親プロセスが終了したら自分も終了する、という制御ができます。

リスト 4-6に、親プロセスの停止を監視するディスパッチソースを組み込む手順を示します。親プロセスが終了すると、ディスパッチソースはある内部状態情報を設定し、子プロセスに対して終了するよう通知します（アプリケーションでは、終了状態を表す適切なフラグを設定する、`MySetAppExitFlag`関数を実装する必要があるかもしれません）。ディスパッチソースは自律的に動作しており、したがって自分自身を所有しているので、プログラムの停止を予測して、自分自身をキャンセルし、解放することもできます。

リスト 4-6　親プロセスの終了を監視

```
void MonitorParentProcess()
{
   pid_t parentPID = getppid();


   dispatch_queue_t queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
   dispatch_source_t source = dispatch_source_create(DISPATCH_SOURCE_TYPE_PROC,
                                                     parentPID, DISPATCH_PROC_EXIT,
 queue);
   if (source)
   {
      dispatch_source_set_event_handler(source, ^{
         MySetAppExitFlag();
         dispatch_source_cancel(source);
         dispatch_release(source);
      });
      dispatch_resume(source);
   }
}
```

ディスパッチソースのキャンセル

ディスパッチソースは、dispatch_source_cancel関数で明示的にキャンセルするまで、アクティブなままです。キャンセルすれば新しいイベントの送信は止まり、この操作は取り消せません。したがって通常は、次のように、キャンセルしたら即座に解放することになります。

```
void RemoveDispatchSource(dispatch_source_t mySource)
{
   dispatch_source_cancel(mySource);
   dispatch_release(mySource);
}
```

ディスパッチソースのキャンセルは非同期的なオペレーションです。dispatch_source_cancel関数の呼び出した後、新たにイベント処理が始まることはありませんが、既に処理中のイベントはそのまま続行されます。イベント処理がすべて終了した後、キャンセルハンドラがあればそれを実行します。

キャンセルハンドラでは、ディスパッチソースに代わって獲得したメモリを解放し、リソースをクリーンアップできます。記述子やmachポートを使っている場合は、これをクローズあるいは破棄する、キャンセルハンドラを用意しなければなりません。それ以外のディスパッチソースでは、キャンセルハンドラがなくても構いません。もっとも、ディスパッチソースでメモリやデータを確保している場合は必要です。たとえば、ディスパッチソースのコンテキストポインタにデータを格納している場合です。キャンセルハンドラについては、“キャンセルハンドラの組み込み” （62 ページ）を参照してください。

ディスパッチソースの一時停止と再開

ディスパッチソースのイベントの送信を、dispatch_suspendおよびdispatch_resumeメソッドで、一時停止し、再開することができます。このメソッドは、ディスパッチオブジェクトの「一時停止」カウンタを増減します。したがって、dispatch_suspendとdispatch_resumeは、対にして呼び出さなければなりません。

一時停止期間中に発生したイベントは、再開までの間、蓄積されています。再開時には、そのイベントをすべて送信するのではなく、1つのイベントに併合してから送信するようになっています。たとえばファイル名の変更を監視している場合、送信されるのは最終のイベントだけです。このようにイベントを併合すれば、再開時に多数のイベントがキューにたまるのを防ぎ、アプリケーションに対する負荷を軽減できます。

スレッドを使ったコードからの移行

スレッドで実装した既存のコードは、さまざまな方法で、Grand Central Dispatchやオペレーションオブジェクトを使って書き直せます。どんな場合でもスレッドを排除できるわけではありませんが、この書き直しができれば、処理性能（およびコードの簡潔性）が格段に向上します。特に、ディスパッチキューやオペレーションキューをスレッドの代わりに使えば、次のような点で好都合です。

- アプリケーション側のメモリ空間にスレッド用のスタックを格納する必要がないので、メモリ効率が改善されます。
- スレッドを生成、設定するコードを記述する必要がありません。
- スレッドを管理、スケジューリングするコードを記述する必要がありません。
- コードを簡潔に記述できます。

この章では、スレッドで実装した既存のコードと同等の処理を、ディスパッチキューやオペレーションキューを使って書き直すための、ヒントやガイドラインを紹介します。

スレッドをディスパッチキューで置き換え

スレッドをディスパッチキューに置き換える方法を考えるにあたっては、現状のアプリケーションでスレッドをどのように使っているか、をまず理解しておく必要があります。

- **単一タスクスレッド**。単一のタスクを実行するスレッドを生成し、タスクが終了したら解放します。
- **ワーカスレッド**。それぞれ特定のタスクを想定した、ワーカスレッドをいくつか生成します。タスクは各スレッドに周期的にディスパッチします。
- **スレッドプール**。汎用スレッドのプールを生成し、それぞれに対して実行ループをセットアップします。タスクが発生するごとに、プールからスレッドを1つ獲得してタスクをディスパッチします。空いているスレッドがなければタスクをキューに入れて、空きスレッドができるまで待機します。

それぞれ大きく異なる技法のようにも見えますが、実際には同じ原理に基づく変化形に過ぎません。いずれの場合もスレッドは、アプリケーションが実行するべき、何らかのタスクのために使われます。違いは、スレッドを管理し、タスクをキューに入れるために用いるコードだけです。ディスパッチキューやオペレーションキューを使えば、スレッド管理やスレッド間通信のコードをすべて排除し、本来実行したいタスクの開発に集中できます。

上記のスレッドモデルのいずれかを使っているならば、アプリケーションが実行するタスクの種類に関しては、かなりよくわかっていると考えてよいでしょう。独自に用意したスレッドにタスクを登録する代わりに、オペレーションオブジェクトまたはブロックオブジェクトにタスクをカプセル化し、適切なキューにディスパッチする方法を試してみてください。特に競合が起こらない（ロックが不要な）タスクについては、次のように置き換えるとよいでしょう。

- 単一タスクスレッドの場合は、ブロックオブジェクトまたはオペレーションオブジェクトにタスクをカプセル化し、並列キューに登録します。
- ワーカスレッドについては、直列キューと並列キューのどちらが適しているか、検討が必要です。ある一連のタスクの実行を同期するためにワーカスレッドを使っているならば、直列キューを選んでください。一方、相互に依存しない任意のタスクを実行するために使っているならば、並列キューにします。
- スレッドプールの場合は、タスクをブロックオブジェクトまたはオペレーションオブジェクトにカプセル化し、並列キューにディスパッチして実行を委ねます。

もちろん、このような単純な置き換えではうまくいかない場合もあります。実行するタスクが共有リソースを巡って競合するならば、まずこの競合を解消するか、最小限にとどめるよう試みてください。コードをリファクタリングし、あるいはアーキテクチャを再考することにより、共有リソースを巡る相互依存性を排除できるならば、ぜひそうするべきです。しかし、それが不可能、あるいは効果的でないとしても、キューにはなお、さまざまな利点があります。最大の利点は、コードがどのように実行されるか、より明確に予測できることでしょう。したがって、ロックその他の重い同期機構を使わなくても、コードの実行を同期する手段があります。ロックの代わりにキューを使っても、多くの場合、同じタスクを実行できるのです。

- 所定の順序で実行しなければならないタスクは、直列ディスパッチキューに登録してください。オペレーションキューの方が使いやすいならば、オペレーションオブジェクトの依存関係を設定して、確実に所定の順序で実行されるようにするとよいでしょう。
- 共有リソースを保護するためにロックを使っているならば、直列キューを生成し、リソースを書き換えるタスクはここで実行してください。直列キューは、ロックに代わる同期機構として利用できます。ロックを排除する技法については、“ロックベースのコードの排除” （77 ページ）を参照してください。

- スレッドの合流機能により、バックグラウンドタスクの完了を待機しているならば、代わりにディスパッチグループを使えないか検討してください。また、`NSBlockOperation`オブジェクトやオペレーションオブジェクトの依存関係を使っても、グループ全体が完了するまで待機する、という動作を実現できます。実行中のタスクのグループを追跡する方法については、“スレッドの合流機能の置き換え”（80 ページ）を参照してください。
- プロデューサ／コンシューマアルゴリズムにより、有限のリソースのプールを管理しているならば、“プロデューサ／コンシューマモデルによる実装の書き直し”（81 ページ）に紹介している実装に置き換えられないか、検討するとよいでしょう。
- スレッドを使って記述子の読み書きをし、あるいはファイル操作を監視しているならば、ディスパッチソースを使ってください（“ディスパッチソース”（57 ページ）を参照）。

重要なのは、どんな場合でもキューがスレッドの代替品になるとは限らない、ということです。キューによる非同期プログラミングモデルは、遅延が多少起こっても問題にならない状況に適しています。キューに登録したタスクの実行優先度を設定することはできますが、高優先度だからといって、指定した時刻どおりに実行される保証はありません。したがって、音声や画像の再生など、遅延を最小限に抑えたい状況では、スレッドの方が優れています。

ロックベースのコードの排除

スレッドを用いたコードでは、スレッド間で共有するリソースへのアクセスを同期する手段として、ロックが古くから使われています。しかしロックの使用にはコストがかかります。競合が起こらない状況でも、ロックを用いれば性能が若干損なわれます。競合が起これば、ロックが解除されるまでの間、いくつかのスレッドがブロックされ、その時間も不確定になる可能性があります。

ロックベースのコードをキューで置き換えれば、このような負担の多くを軽減できるばかりでなく、コードが簡潔になるという利点もあります。ロックを使って共有リソースを保護する代わりに、キューを生成し、逐次的にリソースにアクセスするようにします。こうすれば、ロックの場合のような負担はかかりません。カーネル割り込み（トラップ）を使ってミューテックスを獲得する必要もないのです。

タスクをキューに登録する際、判断しなければならない重要事項として、同期方式か非同期方式か、ということがあります。非同期に登録すれば、タスク実行中も、現スレッドはそのまま処理を続行できます。一方、同期的に登録すると、タスク完了まで現スレッドの実行はブロックされます。用途に応じて使い分ける必要がありますが、可能であれば非同期に登録する方が、有利なことが多いでしょう。

以下のセクションでは、ロックベースの既存のコードを、キューを使って同等のコードに置き換える方法を説明します。

非同期ロックの実装

非同期ロックは共有リソースを保護する手段として有効です。リソースを書き換えるコードがブロックされることはありません。何らかの処理の副作用として、あるデータ構造を書き換えようとする場合に向いています。従来型のロックを使って実装すると、通常、共有リソースのロックを獲得し、書き換え処理を実行し、ロックを解除してからタスク本来の処理に戻る、という手順になります。しかしディスパッチキューを使えば、書き換え処理を非同期に実行できるので、呼び出し元ではそれが終わるまで待機する必要がありません。

リスト 5-1に、非同期ロックと同等の処理を実装する例を示します。この例では、保護されるべきリソースの側が、自分自身で直列ディスパッチキューを定義しています。呼び出し元コードは、リソースに施す変更処理を記述したブロックオブジェクトを、このキューに登録します。キュー自身がブロックを直列に実行するので、リソースの書き換え処理は、ブロックが登録された順になります。しかし、タスクは非同期に実行されるので、呼び出し元スレッドはブロックされません。

リスト 5-1　保護されるリソースを非同期に書き換え

```
dispatch_async(obj->serial_queue, ^{
    // クリティカルセクション
});
```

クリティカルセクションの同期実行

あるタスクが完了するまで、現コードが続行できないようにしたいならば、`dispatch_sync`関数で、当該タスクを同期的に登録してください。この関数は、タスクをディスパッチキューに追加し、当該タスクが完了するまで現スレッドをブロックします。ディスパッチキュー自身は、必要に応じ、直列でも並列でも構いません。しかし、この関数は現スレッドをブロックするので、どうしても必要な場合にのみ使うようにしてください。リスト 5-2に、`dispatch_sync`を使って、コード中のクリティカルセクションをラップする技法を示します。

リスト 5-2　クリティカルセクションの同期実行

```
dispatch_sync(my_queue, ^{
    // クリティカルセクション
});
```

既に直列キューを使って共有リソースを保護しているならば、これに同期的にディスパッチしても、非同期の場合に比べて保護の度合いが高まるわけではありません。それでも同期的にディスパッチする理由があるとすれば、クリティカルセクションが完了するまで、現コードの進行を停めることができる、という点だけです。たとえば、共有リソースから何らかの値を取得してすぐに使いたいのであ

れば、こうする必要があるでしょう。クリティカルセクションが完了するまで待つ必要はない、あるいは、完了したら同じ直列キューに追加のタスクを登録するだけでよい場合は、非同期に登録する方法を推奨します。

ループコードの改善

コード中にループがあり、反復する処理がそれぞれ独立であれば、`dispatch_apply`関数または`dispatch_apply_f`関数を使って実装し直すことを検討してください。この関数は、反復される各回の処理をディスパッチキューに登録し、実行を委ねます。並列キューに登録すれば、各回の処理を並列に実行できます。

`dispatch_apply`や`dispatch_apply_f`は同期的な関数で、反復処理がすべて完了するまで、現スレッドをブロックします。並列キューに登録した場合、その実行順序は保証されません。反復されるそれぞれのスレッドがブロックされ、ほかのスレッドより先に終わるか後に終わるかも不定になる可能性があります。したがって、各ループのブロックオブジェクトや関数は、再入可能でなければなりません。

リスト 5-3に、`for`ループを、ディスパッチベースの同等のコードに置き換える例を示します。`dispatch_apply`や`dispatch_apply_f`に渡すブロックや関数は、何回目の反復か、を表す整数値を引数として受け取るものとします。この例では、単に反復回数をコンソールに出力しているだけです。

リスト 5-3　`for`ループの置き換え（ストライディングなし）

```
queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
dispatch_apply(count, queue, ^(size_t i) {
   printf("%u\n", i);
});
```

ごく単純な例ですが、ディスパッチキューを使ってループを置き換える、基本的な流れは理解できるでしょう。性能改善の面でも優れた技法ですが、実際に使う際には、考慮しなければならない事項があります。ディスパッチキューのオーバーヘッドは非常に小さいのですが、それでもループの処理をそれぞれスレッド上にスケジューリングするためには、ある程度のコストがかかります。したがって、そのコストに見合うよう、ループではそれなりの量の処理をしなければなりません。どの程度の処理量が必要か、正確な値は、性能計測ツールで測定してみるとよいでしょう。

ループの1回あたりの処理量を増やす簡単な手法として、ストライディングというものがあります。元のループの何回分かを、1回でまとめて実行する、という方法です。`dispatch_apply`関数に指定する反復回数は、それに応じて減らします。リスト 5-4に、ストライディングを取り入れてリスト 5-3 （79 ページ）のループを書き換えた例を示します。リスト 5-4では、ストライド数（この例では

137）と同じ回数、ブロック内でprintfを呼び出しています（ストライド数は実際のコードに応じて適宜調整してください）。反復回数をストライド値で割った余りの分は、最後にインラインで実行します。

リスト 5-4　ループを置き換えたコードにストライディングを適用

```
int stride = 137;
dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
 0);


dispatch_apply(count / stride, queue, ^(size_t idx){
    size_t j = idx * stride;
    size_t j_stop = j + stride;
    do {
       printf("%u\n", (unsigned int)j++);
    }while (j < j_stop);
});


size_t i;
for (i = count - (count % stride); i < count; i++)
   printf("%u\n", (unsigned int)i);
```

ストライディングには、性能面ではっきりとした利点があります。特に、ストライド数に対して、元の反復数が多い場合に顕著です。並列にディスパッチするブロック数が少ないほど、ブロック本体の実行に費やせる時間が多くなるからです。もっとも、どんな性能指標でもそうですが、実際のコードに応じて、最も効率的なストライド数になるよう調整する必要があるでしょう。

スレッドの合流機能の置き換え

スレッドの合流機能を使えば、スレッドをいくつか分岐させ、それがすべて完了するまで現スレッドは待機する、ということができます。これを実装するため、親スレッドは子スレッドを、**合流可能なスレッド**として生成します。親スレッドは、子スレッドから処理結果を取得しなければ、これ以上処理を進めることができない、という状態になると、合流により停止状態になります。その結果、親スレッドは、子スレッドのタスク完了までブロックされます。その後、親スレッドは子スレッドから結果を受け取り、自分自身の処理を続行します。複数の子スレッドを合流する場合は、ひとつずつ順に行います。

ディスパッチグループを使えば、スレッドの合流と同様の処理を実装できますが、ほかにも利点があります。ディスパッチグループも、いくつかの子タスクが完了するまで、あるスレッドの実行をブロックする手段である点は同じです。しかしスレッドの合流とは、子タスクをすべて、同時に待機できる点が異なります。さらに、ディスパッチキューを使って実行することも、効率改善に寄与しています。

合流可能なスレッドで実装していた処理を、ディスパッチグループを使って実行する場合、その手順は次のようになります。

1. ディスパッチグループを生成します（`dispatch_group_create`関数）。
2. タスクをグループに追加します（`dispatch_group_async`関数または`dispatch_group_async_f`関数）。グループに登録する各タスクは、従来、合流可能なスレッド上で実行していたものです。
3. 現スレッドがこれ以上処理を進められなくなった時点で、`dispatch_group_wait`関数を呼び出して、グループが実行完了するまで待機します。この関数は、グループに属するタスクがすべて完了するまで、現スレッドをブロックします。

タスクの実装にオペレーションオブジェクトを使っているならば、依存関係を利用しても、スレッドの合流と同等の処理が可能です。親スレッドがタスク完了まで待機する、という構成の代わりに、親のコードそのものをオペレーションオブジェクトに移してしまいます。次に、親となるオペレーションオブジェクトと、従来は合流可能なスレッド上で実行していた処理を行う子のオペレーションオブジェクトとの間に、依存関係を設定します。こうすれば、親のオペレーションオブジェクトは、子のオペレーションオブジェクトがすべて完了するまで処理を続行できません。

ディスパッチグループの使用例については、“キューに追加されたタスクのグループを待機”（54ページ）を参照してください。オペレーションオブジェクト間の依存関係の設定については、“オペレーション間の依存関係の設定”（29ページ）を参照してください。

プロデューサ／コンシューマモデルによる実装の書き直し

プロデューサ／コンシューマモデルを使うと、動的に作られる有限個数のリソースを管理できます。プロデューサはリソース（または処理）を生成し、コンシューマ（複数でも可）はリソース（または処理）の準備が整うのを待って消費します。このモデルを実装する標準的な機構として、条件変数やセマフォがあります。

条件変数を用いる実装では、プロデューサスレッドは次のように動作します。

1. 条件変数に結びつけられたミューテックスをロックします（`pthread_mutex_lock`関数）。
2. リソースまたは処理を生産します。これがコンシューマによって消費されることになります。

3. 消費するべきリソースまたは処理がある旨を表す条件変数を、シグナルとして通知します（pthread_cond_signal関数）。
4. ミューテックスのロックを解除します（pthread_mutex_unlock関数）。

一方、対応するコンシューマスレッドは、次のように動作します。

1. 条件変数に結びつけられたミューテックスをロックします（pthread_mutex_lock関数）。
2. 次の処理を行うwhileループを設定します。
    a. リソースまたは処理の有無を確認します。
    b. 実行するべき処理がない（あるいは使えるリソースがない）場合は、pthread_cond_waitを呼び出して、シグナルの形で通知されるまで、現スレッドをブロックします。
3. 処理（またはリソース）を取得します。これはプロデューサが生産し、供給するものです。
4. ミューテックスのロックを解除します（pthread_mutex_unlock関数）。
5. 処理を実行します。

ディスパッチキューを使えば、プロデューサとコンシューマの実装を、単一の関数呼び出しの形で簡潔に記述できます。

```
dispatch_async(queue, ^{
    // 処理を行う。
});
```

プロデューサは、リソースまたは処理を生産した後、これをキューに追加して処理を委ねるだけで済みます。先に示したコードで、キューの種類だけは、状況に応じて使い分ける必要があります。プロデューサが生成したタスクを、所定の順序で実行する必要がある場合は、直列キューを使ってください。一方、並列に実行しても構わないならば、並列キューに追加して、同時にできるだけ多数を実行できるようにするとよいでしょう。

セマフォを使ったコードの置き換え

共有リソースへのアクセスを制限するためにセマフォを使っている場合は、代わりにディスパッチセマフォが使えないか検討してください。従来型のセマフォは、状態を調べるために、必ずカーネル呼び出しが必要になります。一方、ディスパッチセマフォは、ユーザ空間内で迅速に状態を調べることができます。カーネル機能を呼び出すのは、共有リソースが使えない状態のため、呼び出し元スレッドをブロックする必要が生じた場合のみです。そのため、競合がない場合、従来型のセマフォに比べて処理が迅速です。それ以外の振る舞いは従来型のセマフォと変わりません。

ディスパッチセマフォの使用例は、“ディスパッチセマフォを使って有限のリソースの使いかたを規制”（53 ページ）を参照してください。

実行ループの置き換え

いくつかのスレッド上で実行される処理を実行ループで管理している場合、キューを使った方が、実装も保守も容易です。独自の実行ループをセットアップする場合、その動作基盤であるスレッドと、実行ループ自身の、両方を考慮しなければなりません。実行ループのコードでは、実行ループソースをいくつかセットアップし、このソースに届くイベントを処理するコールバック関数を記述することになります。この代わりに、直列キューを生成してタスクをディスパッチするだけでも、同等の処理が可能です。その場合、スレッドと実行ループを生成するコードは、次の1行だけになってしまいます。

```
dispatch_queue_t myNewRunLoop = dispatch_queue_create("com.apple.MyQueue", NULL);
```

キューは追加されたタスクを自動的に実行するので、その管理のために特別なコードを記述する必要はありません。スレッドの生成や設定も、実行ループソースの生成やアタッチも不要です。さらに、キュー上で新しい種類の処理を行うことも、単にタスクを追加するだけでできてしまいます。同じことを実行ループで実現しようとすれば、既存の実行ループソースを修正するか、別に生成しなければなりません。

実行ループの標準的な使いかたとして、ネットワークソケットから非同期に届くデータを処理する、というものがあります。しかし、その代わりに、ディスパッチソースを適当なキューにアタッチする、という方法でも実装できます。ディスパッチソースには、従来の実行ループソースよりも多くのデータ処理機能が備わっています。タイマーイベントやネットワークポートイベントの処理だけでなく、ファイルの読み書きや、ファイルシステムオブジェクト、プロセス、シグナルの監視も可能です。さらに、独自のディスパッチソースを定義し、コードのほかの部分から非同期にトリガをかけることもできます。ディスパッチソースの設定については、“ディスパッチソース”（57 ページ）を参照してください。

POSIXスレッドとの互換性

Grand Central Dispatchは、タスクとそれが動作するスレッドの関係を管理しているので、一般に、タスクコードからPOSIXスレッドルーチンを呼び出すのは避けるべきです。何らかの理由でこれを呼び出す必要がある場合は、どのルーチンを呼び出すか、慎重に検討してください。このセクションでは、キューに追加したタスクから、どのルーチンならば呼び出しても安全か、どのルーチンはそうでないか、について説明します。以下に挙げるリストは、完全なものではありませんが、何が安全で何がそうでないか、おおよそのところはわかるでしょう。

一般にアプリケーションは、自身が生成したオブジェクトやデータ構造体以外を、削除したり書き換えたりしてはなりません。したがって、ディスパッチキューを使って実行されるブロックオブジェクトが、次の関数を呼び出すことはできません。

```
pthread_detach
pthread_cancel
pthread_join
pthread_kill
pthread_exit
```

タスク実行中にスレッドの状態を変えるのは構いませんが、タスクから復帰する前に、元の状態に戻す必要があります。したがって次の関数は、最後に元の状態に戻すのであれば、呼び出しても安全です。

```
pthread_setcancelstate
pthread_setcanceltype
pthread_setschedparam
pthread_sigmask
pthread_setspecific
```

ブロックの実行基盤となるスレッドは、起動する都度、毎回同じものになるとは限りません。したがってアプリケーションでは、次の関数の戻り値が、ブロックを起動する都度同じになる、と想定することはできません。

```
pthread_self
pthread_getschedparam
pthread_get_stacksize_np
pthread_get_stackaddr_np
pthread_mach_thread_np
pthread_from_mach_thread_np
pthread_getspecific
```

Important: ブロックでは、その内部でスローされた言語レベルの例外を捕捉し、上位に伝わらないようにする必要があります。ブロックの実行中に発生するその他のエラーは、ブロック内で同様に処理するか、アプリケーションのほかの部分に通知するために使ってください。

このセクションで述べた、POSIXスレッドおよび関数については、`pthread`のmanページを参照してください。

用語解説

アプリケーション（application）　プログラム（program）のうち、グラフィカルインターフェイスを使ってユーザとやりとりするもの。

非同期設計アプローチ（asynchronous design approach）　アプリケーションの構成法のひとつ。いくつかのコードブロックを、メインスレッドまたはその他の実行スレッド上で、並列に実行するモデル。非同期タスクは、これを起動するスレッドと実行されるスレッドが別なので、プロセッサリソースを効率的に使って迅速に処理できます。

ブロックオブジェクト（block object）　インラインコードとデータをカプセル化したCの構成体。これを単位として、後で（コード記述上の位置とは異なる時点で）実行できます。ブロックの形で記述したタスクは、現スレッド上でインラインで、あるいはディスパッチキューを使って独立したスレッド上で実行できます。詳細については、『*Blocks Programming Topics*』を参照してください。

並列操作（concurrent operation）　オペレーションオブジェクトのうち、`start`メソッドの呼び出し元スレッド以外でタスクを実行するもの。通常、独自のスレッドをセットアップするか、処理を行う独立したスレッドをセットアップするインターフェイスを呼び出します。

条件変数（condition）　リソースへのアクセスを同期するために使う構成体。ある条件を満たすまで待機するスレッドは、ほかのスレッドがシグナルとして明示的に条件を伝えるまで、処理を続行できません。

クリティカルセクション（critical section）　コードのうち、同時には1つのスレッドでしか実行されない部分。

カスタムソース（custom source）　アプリケーションが定義したイベントの処理に使われるディスパッチソース（dispatch source）。アプリケーションが生成するイベントに応じて、別に用意したイベントハンドラを呼び出します。

記述子（descriptor）　ファイル、ソケットその他のシステムリソースにアクセスするために使う、抽象的な識別子。

ディスパッチキュー（dispatch queue）　Grand Central Dispatch（GCD）が提供する、アプリケーションのタスクを実行するための構造体。GCDでは、直列または並列にタスクを実行するディスパッチキューを用意しています。

ディスパッチソース（dispatch source）　Grand Central Dispatch（GCD）が提供するデータ構造体で、システム関連のイベントを処理するために生成するもの。

記述子ディスパッチソース（descriptor dispatch source）　ファイル関連のイベントを処理するために使うディスパッチソース（dispatch source）。ファイルを読み書きできるようになった、あるいはファイルシステムに変更が施された場合に、別途用意したイベントハンドラを呼び出します。

動的共有ライブラリ（dynamic shared library） アプリケーションのプロセス空間に、動的にロードされる実行形式バイナリ。アプリケーション自体に静的にリンクされるライブラリと対照して用いる用語。

フレームワーク（framework） 動的共有ライブラリと、これをサポートするリソースやヘッダファイルを組にして提供され、アプリケーションの基盤として機能するソフトウェア。詳細については、『*Framework Programming Guide*』を参照してください。

グローバルディスパッチキュー（global dispatch queue） ディスパッチキュー（dispatch queue）が提供し、アプリケーションが自由に使えるGrand Central Dispatch（GCD）。生成する必要も、保持、解放する必要もありません。システムに組み込まれた関数を使って獲得できます。

Grand Central Dispatch（GCD） 非同期タスクを並列に実行する技術。OS X v10.6以降、iOS 4.0以降で利用できます。

入力ソース（input source） スレッドに送信される非同期イベントのソース。ポートベースのものと直接トリガをかけるものがあり、スレッドの実行ループにアタッチする必要があります。

合流可能なスレッド（joinable thread） スレッドのうち、終了時にリソースが直ちに回収されないもの。明示的にデタッチするか、ほかのスレッドに合流してからでないと、リソースを回収できません。合流される側のスレッドに戻り値を返します。

ライブラリ（library） 低レベルのシステムイベントを監視する、UNIXの機能。詳細については、`kqueue`のmanページを参照してください。

Machポートディスパッチソース（Mach port dispatch source） Machポートに届くイベントを処理するために使うディスパッチソース（dispatch source）。

メインスレッド（main thread） スレッド（thread）のうち、プロセス生成時に自動的に生成されるもの。プログラムのメインスレッドが終了すれば、プロセスも終了します。

ミューテックス（mutex） 共有リソースに対する相互排除アクセスを可能にするロック。同時には1つのスレッドだけが保持できます。ほかのスレッドが保持しているミューテックスを獲得しようとすると、そのスレッドは、ロックを獲得できるまでの間スリープ状態になります。

Open Computing Language（OpenCL） グラフィックプロセッサ上で汎用的な計算処理を行う、標準ベースの技術。詳細については、『*OpenCL Programming Guide for Mac*』を参照してください。

オペレーションオブジェクト（operation object） `NSOperation`クラスのインスタンス。タスクに関連するコードとデータを、実行可能な単位にラップします。

オペレーションキュー（operation queue） `NSOperationQueue`クラスのインスタンス。オペレーションオブジェクトの実行を管理します。

プライベートディスパッチキュー（private dispatch queue） ディスパッチキュー（dispatch queue）のうち、明示的に生成、保持、解放するもの。

プロセス（process） アプリケーションやプログラムの実行時インスタンス。ほかのプログラムとは独立した、自分だけの仮想メモリ空間とシステムリソース（ポート権限を含む）を持ち

ます。プロセスにはスレッドが少なくとも1つある（メインスレッド）ほか、いくつでも追加で生成できます。

プロセスディスパッチソース（process dispatch source） プロセス関連のイベントを処理するために使うディスパッチソース（dispatch source）。指定したプロセスの状態が変化した場合に、別途用意したイベントハンドラを呼び出します。

プログラム（program） 何らかのタスクを実行できる、コードとリソースを組み合わせたもの。グラフィカルアプリケーションもプログラムの一種ですが、単にプログラムと言った場合、グラフィカルユーザインターフェイスはなくても構いません。

再入可能（reentrant） あるスレッド上で動作中に、別のスレッド上で新たに起動しても安全なコード。

実行ループ（run loop） イベントを受信して適切なハンドラにディスパッチする、イベント処理ループ。

実行ループモード（run loop mode） 入力ソース、タイマーソース、実行ループオブザーバを一式集めたものに、名前を与えて識別できるようにしたもの。実行ループをある「モード」で動かすと、関係するソースやオブザーバのみを監視するようになります。

実行ループオブジェクト（run loop object） `NSRunLoop`クラスまたは`CFRunLoopRef`不透過型のインスタンス。スレッドにイベント処理ループを実装するためのインターフェイスを提供します。

実行ループオブザーバ（run loop observer） 実行ループの各実行フェーズにおける通知の受け手。

セマフォ（semaphore） 共有リソースへのアクセスを制限する、保護された変数。ミューテックスや条件変数はセマフォの一種です。

シグナル（signal） ドメイン外からプロセスを操作する、UNIXの機構。システムはシグナルを使って、「不正な命令が実行された」など、重要なメッセージをアプリケーションに伝達します。詳細については、`signal`のmanページを参照してください。

シグナルディスパッチソース（signal dispatch source） UNIXシグナルを処理するために使うディスパッチソース（dispatch source）。プロセスがUNIXシグナルを受信したときに、別途用意したイベントハンドラを呼び出します。

タスク（task） 実行するべき処理の量。ほかの技術（特にCarbon Multiprocessing Services）ではこの用語を別の意味で使っていますが、一般的には、実行するべき処理を何らかの形で定量化した、抽象的な概念を意味するものとして使います。

スレッド（thread） プロセスにおける、実行フローの単位。各スレッドには独自のスタック空間がありますが、それ以外のメモリは、同じプロセス内のほかのスレッドと共有します。

タイマーディスパッチソース（timer dispatch source） 周期的に発生するイベントの処理に使われるディスパッチソース（dispatch source）。一定時間ごとに、別途用意したイベントハンドラを呼び出します。

書類の改訂履歴

この表は「*並列プログラミングガイド*」の改訂履歴です。

日付	メモ
2012-12-13	ディスパッチキューに関する説明の誤りを修正しました。
2012-07-17	自動解放プールの使い方に関する古い情報を削除しました。
2011-01-19	手動で実行した操作でキャンセル処理が正しく行われるためのコードを更新しました。 ディスパッチキューとの関連におけるObjective-Cオブジェクトの使用に関する情報を追加しました。
2010-04-13	iOSのサポートを反映するように更新しました。
2009-08-07	非並列操作オブジェクトのサンプルで開始メソッドを修正しました。
2009-05-22	複数のコードパスを並列処理するためのテクノロジーについて解説した新規ドキュメント。

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
U.S.A.

Apple Japan
〒106-6140 東京都港区六本木 6
丁目10番1号 六本木ヒルズ
http://www.apple.com/jp