

# Java による TCP プログラミング

2010 年 8 月

2010 年 9 月 (一部訂正)

株式会社 クレス

ネットワーキング・チュートリアルを株式会社クレスのサイトにアップロードして以来 9 年間に経過した。その間多くの Java によるネットワーキング・プログラミングのチュートリアルもインターネット上で多数出現している。

しかしながら今回新しくネットワーキング・プログラミングのチュートリアルを作成することにしたのは、単なるネットワーキング API の基本と使い方を習得するためだけのチュートリアルではなく、もっとその基本動作をきちんと説明し、開発者が自分が開発したアプリケーションは実際はどのようにそのもとになっているプラットフォームやネットワーク上で動作しているかを理解できる教材の必要性を感じているからである。

開発者たちは優れた開発環境やフレームワークをもとにアプリケーションを開発しているが、そのテスト段階や実際のサービス段階での問題や障害に面したときには、そのような知識が威力を発揮しよう。

ここではプログラム開発環境として、広く普及している Eclipse を使用した。

なおこの資料に含まれるプログラムはチュートリアル用のもので、商用化及びその結果に対しては当社は責任を持たない。

# 目次

目次.....	2
第1章 Eclipse.....	6
1.1 節 Eclipseとは.....	6
1.1.1 歴史.....	6
1.1.2 リリース.....	7
1.1.3 ライセンシング.....	7
1.2 節 Eclipseの構成.....	8
1.2.1 構成.....	8
1.2.2 ワークベンチ.....	10
1.3 節 Eclipseのインストール.....	11
1.3.1 Java JREのみのインストール.....	12
1.3.2 Java EE 6 SDKのインストール.....	12
1.3.3 Eclipseのインストール.....	13
1.4 節 基本的な使用法.....	15
1.4.1 Java開発の為のEclipseの基本要素.....	15
1.4.1.1 Javaのパーstekティブ.....	15
1.4.1.2 プロジェクト.....	16
1.4.1.3 Javaビルダ.....	16
1.4.1.4 ビルドのクラスパス.....	17
1.4.1.5 クラスパス変数(Classpath variables).....	17
1.4.2 新しいJavaプロジェクトの生成.....	17
1.4.3 トップ・レベルのクラスの生成.....	18
1.4.4 Javaプログラムの実行.....	19
1.4.5 添付ファイルのインポート.....	19
第2章 TCP/IP.....	21
2.1 節 通信プロトコル.....	21
2.1.1 OSIのプロトコル・スタック.....	21
2.1.2 インターネットのプロトコル・スタック.....	21
2.2 節 IP.....	24
2.2.1 IPデータグラム.....	24
2.2.2 IPv6データグラム.....	26
2.2.3 IPのアドレス.....	27
2.2.3.1 IPv4のアドレス.....	27
2.2.3.2 IPv6のアドレス.....	29
2.2.3.3 ドメイン名とIPアドレスのマッピング.....	29
2.2.4 IPルーティングの基礎.....	31
2.2.4.1 ローカル・ホストでの経路設定.....	31
2.2.4.2 ルータでの経路設定.....	31
2.2.5 ルータの基本アルゴリズムと構成.....	33

2.2.5.1	ベクター・ディスタンス.....	35
2.2.5.2	リンク・ステート、最短パス最初 (Link-State, Shortest Path First).....	36
2.2.5.3	BGP (Border Gateway Protocol).....	38
2.3 節	TCP.....	41
2.3.1	TCP のコンセプト.....	41
2.3.1.1	信頼性あるメッセージの到着の保証.....	41
2.3.1.2	誤りの検査.....	42
2.3.1.3	フロー制御.....	42
2.3.1.4	コネクション・モードとポートの概念.....	42
2.3.2	コネクションの状態遷移.....	43
2.3.2.1	コネクションの開設.....	45
2.3.2.2	コネクションの開放.....	45
2.3.2.3	コネクションのリセット.....	46
2.3.3	データ転送.....	46
2.3.4	緊急データ送信.....	47
2.3.5	ウィンドウ(フロー)制御.....	48
2.3.6	UDP.....	49
第3章	ソケット・インターフェイス.....	50
3.1 節	Java ソケット・インターフェイス.....	50
3.2 節	Java.net での TCP 通信の概念.....	51
3.3 節	Java.net での UDP 通信の概念.....	53
3.4 節	Java.net の主なクラスたち.....	53
3.4.1	Socket クラス.....	53
3.4.2	ServerSocket クラス.....	62
3.4.3	DatagramSocket クラス.....	67
3.4.4	DatagramPacket クラスのコンストラクタとメソッド.....	74
第4章	Java NET による TCP 通信のプログラム.....	77
4.1 節	エコーバック・サーバの動作記述.....	77
4.2 節	エコーバック・サーバのプログラム.....	79
4.2.1.1	このプログラムの動作.....	82
4.2.1.2	ストリームと文字エンコーディング.....	82
4.3 節	クライアントのプログラム.....	83
4.3.1.1	テキスト行ベースの通信における行終端.....	85
4.3.1.2	ローカル・ホスト(Localhost).....	85
4.4 節	Eclipse による確認.....	86
4.4.1	プログラムの作成.....	86
4.4.2	サーバとクライアントをデバッガ上で走らせる.....	86
4.4.3	複数のクライアントを立ち上げる.....	88
4.5 節	Tera Term による確認.....	89
4.6 節	Java.net における TCP の接続と開放.....	91
4.6.1	TCP の接続.....	91
4.6.2	TCP の開放.....	92
4.6.3	setReuseAddress メソッドについて.....	93

4.7 節 例外処理の確認.....	94
4.7.1 ネットワーク障害.....	94
4.7.2 キープ・アライブ.....	96
第5章 Java NIO.....	97
5.1 節 ByteBuffer.....	97
5.1.1 ByteBuffer のイメージ.....	98
5.1.2 Buffer と ByteBuffer クラスのメソッド.....	101
5.1.3 インスタンスの生成.....	116
5.1.4 ByteBuffer への読み書き.....	116
5.2 節 文字セット変換.....	117
5.2.1 一般的なクラスを使った変換.....	118
5.2.2 NIO のクラスを使った変換.....	118
5.2.3 PrintStream での変換.....	120
5.3 節 NIO のチャンネル.....	120
5.3.1 ファイル I/O での読み書き.....	121
5.3.2 ネットワーク I/O でのチャンネル.....	121
5.4 節 非ブロッキング・ソケット I/O.....	122
5.4.1 非ブロッキング・サーバと Apache MINA.....	122
5.4.2 非ブロッキング・ソケット I/O のセレクタとチャンネル.....	123
5.4.3 一般的な処理のながれ.....	123
5.4.4 NonBlockingServer のプログラム.....	126
5.4.5 実用に耐えるサーバ.....	129
第6章 Apache MINA.....	131
6.1 節 Apache MINA のコンセプト.....	131
6.2 節 基本的なクラスとインターフェイス.....	133
6.2.1 IoHandlerAdapter.....	134
6.2.2 IoSession.....	136
6.2.3 IoService.....	141
6.2.4 IoAcceptor.....	144
6.2.5 SocketAcceptor.....	146
6.2.6 NioSocketAcceptor.....	147
6.2.7 IoBuffer.....	150
6.3 節 Apache MINA が使っているロギング・システム.....	152
6.3.1 基本的な使い方.....	152
6.3.2 ロギング・システムのバインド.....	153
6.4 節 エコーバック・サーバのプログラム例.....	154
6.4.1 SLF4J のパッケージのダウンロード.....	154
6.4.2 MINA のパッケージのダウンロード.....	155
6.4.3 簡単なエコー・サーバのプログラム.....	155
6.4.4 Eclipse での動作確認.....	158
6.5 節 Apache MINA での文字エンコーディング.....	159
6.5.1 バイト・バッファでの変換.....	159
6.5.2 エンコーダ・フィルタによる方法.....	160
6.6 節 Apache MINA のフィルタ.....	163

6.6.1 圧縮のためのフィルタ(CompressionFilter).....	164
6.6.2 ブラック・リストのフィルタ(BlacklistFilter).....	164
6.6.3 送信データをバッファするフィルタ(BufferedWriteFilter).....	165
6.6.4 接続帯域制限フィルタ(ConnectionThrottleFilter).....	165
6.6.5 スレッド処理のフィルタ(ExecutorFilter).....	165
6.6.5.1 ライフ・サイクル管理.....	165
6.6.5.2 イベントの順序付け(Event Ordering).....	165
6.6.5.3 選択的フィルタリング(Selective Filtering).....	166
6.6.5.4 OutOfMemoryError エラーの防止.....	166
6.6.6 キープ・アライブのフィルタ(KeepAliveFilter).....	166
6.6.6.1 IoSessionConfig.setIdleTime(IdleStatus, int)とのインターフェイス.....	166
6.6.6.2 KeepAliveMessageFactory の実装.....	167
6.6.6.3 タイムアウト処理.....	168
6.6.6.4 このフィルタの追加法.....	168
6.6.7 ロギング・フィルタ(LoggingFilter).....	168
6.6.8 処理時間計測のフィルタ(ProfilerTimerFilter).....	168
6.6.8.1 テスト・プログラム.....	169
6.6.8.2 テスト例.....	171
6.6.9 メッセージのエンコードとデコードのフィルタ(ProtocolCodecFilter).....	171
6.6.10 プロキシのためのフィルタ(ProxyFilter).....	172
6.6.11 SSL フィルタ(SslFilter).....	172
6.7 節 Apache MINA のスレッド.....	173
6.7.1 Apache MINA のデフォルトのスレッド・モデル.....	173
6.7.2 MINA のスレッド・モデルの設定.....	174
6.7.2.1 IO ハンドラ・スレッド(NioProcessor)数の設定.....	174
6.7.2.2 ExecutorFilter スレッドの設定.....	175
6.7.3 簡単なテスト・プログラムによる確認.....	175
6.7.3.1 テスト用クライアント(MultiClientSimulator).....	175
6.7.3.2 MINA のデフォルト設定のサーバのテスト.....	177
6.7.3.3 スレッド枯渇時の返信の遅れ.....	178
6.7.3.4 クライアント接続継続との関係.....	179
6.7.3.5 NIO プロセッサ・スレッド・プールのサイズ増加による処理能力向上.....	180
6.7.3.6 スレッド処理のためのフィルタ追加による処理能力向上.....	180
6.7.4 スレッド・モデルの選択.....	184
6.8 節 複数のサーバの実装.....	185
6.9 節 Apache MINA での UDP アプリケーション.....	187
6.9.1 サーバのコード.....	188
6.9.2 クライアントのコード.....	191

# 第1章 Eclipse

今回のチュートリアルは開発環境は広く普及している Eclipse を使用する。Eclipse はいろんなプログラミング言語に対応する統合開発環境 (IDE : Integrated Development Environment) である。Eclipse に関する書籍も数多く出版されているので、ここではそのポイントと、実際のダウンロードと使用方法を手短に説明する。

## 1.1節 Eclipse とは

### 1.1.1 歴史

Eclipse は 1990 年代中頃出現した Java ベースの IDE たちのひとつである IBM の VisualAge for Java を初期コードベースとしている。そのころ IT でのオープンなアプローチが顧客の長期的な成功の為の最も良い手段だと確信した IBM は、Java 開発ツールがオープンなコミュニティを発展させる為には鍵と判断した。従ってその時点での同社の目標は開発者たちにとって Java ベースのミドルウェアをより手近なものとする事だった。IBM はまた総ての同社の開発製品のための共通のプラットフォームを確立して、IBM の各部門が作った複数のツールを使っている顧客がツールを切替るなかで一貫してより統合化された経験を持つようにしようとした。そこで同社は IBM からのツール、顧客向けのカスタムのツールボックス、及びサード・パーティのツールの混在した組み合わせで構成される顧客たちの為の開発環境を想定した。この混在してはいるが互換性を持ったツール環境がソフトウェア・ツールのエコシステム、即ち多くの関与するパーティが協働するソフトウェア・ツールのビジネス・モデルの最初だった。

1998 年 11 月に IBM のソフトウェア・グループ部門が開発ツールのプラットフォームの作成を開始し、これが最終的に Eclipse として知られるものとなった。当初は 1996 年に買収した Visual Age for Java を含む IDE を作っていたカナダの Object Technology International (OTI) という会社の専門技術者たちを中心にしてこのプラットフォームを開発し、また更なるチームを追加してこのプラットフォーム上の新製品を開発した。

当時は IBM は Eclipse の広範な普及にはサード・パーティたちの活発なエコシステムが不可欠だと考えていたが、同社のビジネス・パートナーたちはあまり積極的でなかったため 2001 年 11 月にオープン・ソースのライセンスと運用モデルを採用することを決定している。その結果 IBM と他の 8 組織からなる Eclipse のコンソーシアムを設立している。参加者たちには社内で Eclipse を使い、それを促進し、またそれをベースとした製品を出荷するという誠実なコミットメント (bona fide commitment) が求められた。

2003 年までに Eclipse の最初のリリースが開発者たちの好評と導入を獲得したが、IBM が支配している取り組みではなかつたの批判を受け、同社はより IBM から切り離された組織にすることを決め、その結果独立した Eclipse Foundation という非営利組織が 2004 年 1 月に設立された。

Eclipse Foundation は Eclipse 3.0 を、またその後 Eclipse 3.1 を出荷出荷し、双方とも大きな関心を集め爆発的な普及をした。現在は毎年 6 月末に新しいバージョンがリリースされている。

表 1-1 : Eclipse のリリース

リリース	リリース日	プラットフォームのバージョン	プロジェクト
Eclipse 3.0	28 June 2004	3.0	
Eclipse 3.1	28 June 2005	3.1	
Callisto	30 June 2006	3.2	<a href="#">Callisto projects</a>
Europa	29 June 2007	3.3	<a href="#">Europa projects</a>
Ganymede	25 June 2008	3.4	<a href="#">Ganymede projects</a>
Galileo	24 June 2009	3.5	<a href="#">Galileo projects</a>
Helios	23 June 2010	3.6	<a href="#">Helios projects</a>

3.2 以降のリリース名は当初は Ganymede まで木星の衛星からとられていたが、現在はアルファベット順で企画評議会(Planning Council)が選択している。

### 1.1.2 リリース

[Eclipse のダウンロードのページ](#)でわかるように、Eclipse の各リリースには幾つかのビルド(各節目での実行可能なファイル)が存在する。

- リリース(Release)  
リリースというのは商用に供する等で安定でテストされたものが必要なユーザ向けのもので、その代り最新の機能や改善は含まれていない。リリースのビルドの名前は例えば R1.0 というように常に'R'で始まる。
- 安定ビルド(Stable Builds)  
安定ビルドというのは殆どの人たちにとって十分使えるほど安定なビルドを言う。これはアーキテクチャのチームが次の統合ビルドを数日間使ってみて安定とみなせると判断したものである。これは最新の機能や改善を把握したいユーザに適している。これは開発チームが貴重でタイムリなフィードバックをユーザから得るためのビルドである。
- 統合ビルド(Integration Builds)  
これは定期的に各部品の開発チームたちが安定であるとみなしたものをバージョン・アップしたものである。統合ビルドは数日間のテストを経て安定ビルドとなる。
- 夜間ビルド(Nightly Builds)  
これは CVS レポジトリの HEAD ストリームにリリースされたものから毎晩作られるもので、これは開発チームのメンバーようである。
- 保守ビルド(Maintenance Builds)  
これは既存のリリースに他する問題修正を組み入れるために定期的になされている。

なお英語圏以外向けの言語パックが存在する。多言語パックはこれまでのバージョンでは IBM が無償提供していたが、コストがかかるため IBM はバージョン 3.3 からこの貢献をやめており、現在は [Babel というプロジェクト](#)が地域ベースで開発している。言語パックの最新は 2010 年 5 月時点で 0.7.1 で、これは Galileo リリース用である。

### 1.1.3 ライセンシング

Eclipse は Eclipse パブリック・ライセンス(Eclipse Public License)の契約条件のもとでリリースされており、無料かつオープンなソフトウェアである。これは IBM の共通パブリック・ライセンス(Common Public License)に沿ったものである。これは[日本語訳もネット上に存在する](#)ので参照されたい。

貢献者、すなわち Eclipse のそのバージョンに貢献したものは総て非独占的で世界規模で使用料無料の著作権ライセンス及び特許権(もしあれば)ライセンスを利用者に付与する。ただしその分、明示黙示を問わず、タイトル、非侵害性、商業的な使用可能性、および特定の目的に対する適合性に関する保証を含め、いかなる保証もされない。

このようなライセンスは現在はオープンなソフトウェアでは一般化している。例えば Apache の Tomcat や HTTP サーバなどでは [Apache License, Version 2.0](#) が適用されており、Google のスマートフォン用の OS の Android は GNU General Public License (GPL3)が採用されている。

## 1.2節 Eclipse の構成

Eclipse は総てが Java で書かれており、Windows、Linux、Solaris などいろんな OS で使用できるのが特徴である。またプラグインによりいろんなプログラミング言語での開発が可能になっている。このプラットフォームの最終的な機能を決めているのはプラグインたちである。Java 言語での開発には Java 開発ツール (JDT:Java development tools) というプラグインを使用する。

### 1.2.1 構成

Eclipse のプラットフォームはそれ自身はひとつ以上のプラグインに実装されるサブシステムたちで構造化されている。これらのサブシステムは小さなランタイムのエンジン上で構成されている。下図はその単純化された構成を示す。プラグインたちはこのシステムに機能を提供するコード及び / あるいはデータのバンドルで構成される。機能はコード・ライブラリ(パブリックな API を持った Java クラスたち)、プラットフォーム拡張物、あるいはドキュメンテーションの形で貢献される。プラグインたちは他のプラグインが機能を追加できる場所となる拡張点(extension points)を定義できる。

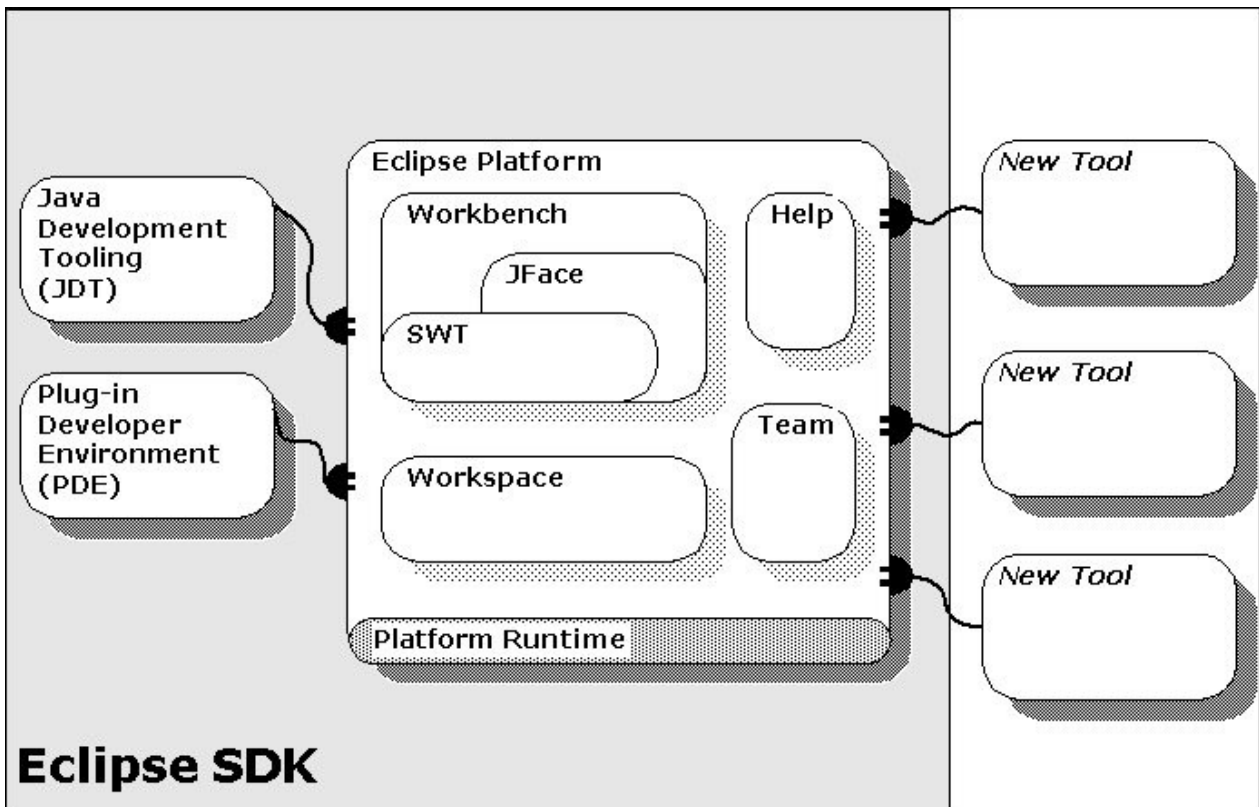
このプラットフォームの中の各サブシステムはそれ自身は幾つかの主要機能を実装したプラグインたちのセットで構成される。一部のプラグインはこの拡張モデルを使ってこのプラットフォームに可視な機能を追加している。他のプラグインではシステム拡張物を実装するのに使えるクラス・ライブラリを供給している。

Eclipse SDK はベーシックなプラットフォームに加えてプラグイン開発に有用な二つのツールが加えられている。Java 開発ツール (JDT:Java development tools) は完全な機能を持った Java 開発環境を有している。Java の開発者はこのツールを使用することになる。プラグイン開発者環境 (PDE:Plug-in Developer Environment) はプラグインと拡張の開発を簡素化することに特化したツールを付加するものである。

これらのツールは有用性を持っているだけでなく、このシステムを拡張するプラグインを構築することでこのプラットフォームにどのように新しいツールが追加され得るかの良い事例となっている。

図 1-1 :Eclipse の構成とプラグインの概念





出典: [http://help.eclipse.org/galileo/topic/org.eclipse.platform.doc.isv/guide/int\\_eclipse.htm](http://help.eclipse.org/galileo/topic/org.eclipse.platform.doc.isv/guide/int_eclipse.htm)

サブシステムを構成するプラグインたちはこのプラットフォームに振る舞いを追加するための拡張点を指定する。下表はひとつあるいはそれ以上のプラグインを実装するこのプラットフォームの主たる部品を示している:

プラットフォーム・ランタイム (Platform runtime)	拡張点とプラグイン・モデル。プラグインをダイナミックに発見し、そのプラグインとその拡張点に関する情報をプラットフォームのリポジトリに維持。プラグインはユーザが要求したときに開始する。ランタイムは <a href="#">OSGi フレームワーク</a> を使って実装されている。
リソース管理 (作業領域: ワークスペース) (Resource management (workspace))	ツールツールたちが作りだしファイル・システムに保持されるリソースたち (プロジェクト、ファイル、及びフォルダ) の作成と管理。
ワークベンチ・ユーザ・インターフェイス (Workbench UI)	このプラットフォームをナビゲートするためのユーザのコックピットとなるもの。ビューまたはメニューのアクションのようなユーザ・インターフェイス(UI)を追加するための拡張点である。これは UI 構築のためのツールキットたち (JF あせと SWT) を含んでいる。これらの UI サービスは構造化されていて、リソース管理とワークスペース・モデルとは独立して、リッチなクライアント・アプリケーションを構築するのにこれらの UI プラグインたちのサブセットが使えるようになっている。IDE に特化したプラグインではリソースのナビゲーションと操作の機能が追加される。
ヘルプ・システム	ブラウザ可能なヘルプまたは他のドキュメンテーションを提供するためのプラ

(Help system)	グインの拡張点。
チーム・サポート (Team support)	リソースたちの管理とバージョンングのためのチーム・プログラミングのモデル。
デバッグ・サポート (Debug support)	言語依存性のないデバッグ・モデル、及びデバッガとローンチャ構築のための UI のクラスたち。
その他のユーティリティ (Other utilities)	リソースのサーチと比較、XML 設定ファイルを使ったビルド、及びサーバからのこのプラットフォームのダイナミックなアップデートなどのような機能を提供するためのユーティリティのプラグインたち。

### 1.2.2 ワークベンチ

ワークベンチの中心になっているのが前述の SWT (標準ウィジェット・ツールキット: Standard Widget Toolkit) と JFace である。

SWT は Java の AWT や Swing に代わる GUI ツールキットで、OS によって異なる UI を独自の可搬性ある API セットで統一できるようにしている。使われているネイティブ OS の GUI プラットホームとの密な統合が特徴であり、ユーザは自分が使っている OS の UI と同様な操作で Eclipse を使うことができる。SWT は総てのサポートされるプラットフォーム上で提供される共通の可搬の API であり、出来る限りネイティブなウィジェット使って各プラットフォームごとに用意される。

一方ワークベンチはこのプラットフォームの UI のプラグインの拡張点ではあるが、その多くは(特にウィザードの拡張の場合は) org.eclipse.jface.\* のパッケージのクラスを使っている。JFace は UI 機能の拡張に便利なヘルパー的クラスが用意されている。これらのクラスには以下の共通の UI プログラミングのタスク処理のためのクラスが含まれている:

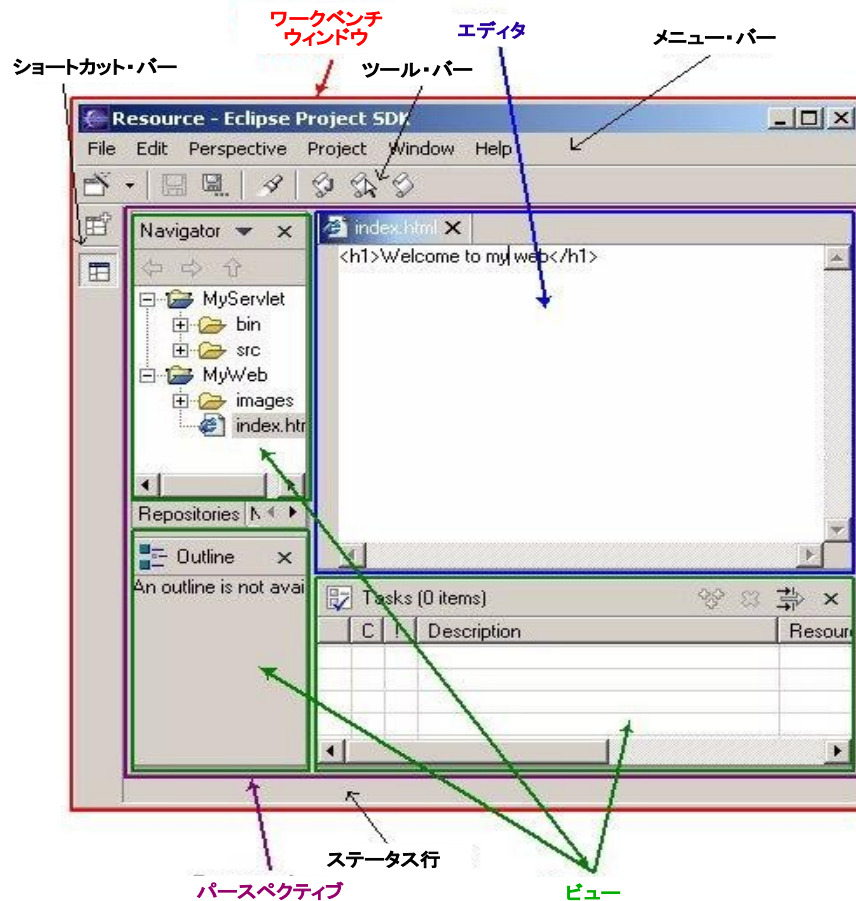
- ビュワー (Viewers) : ウィジェットたちを集める、ソートする、フィルタリングする、更新するといった面倒な処理を行う
- アクションと貢献 (Actions and contributions) : ユーザのアクションを決め、それらのアクションがどこで使えるかを定める
- イメージとフォントのレジストリ (Image and font registries) : UI のリソースを処理するための共通のボタンを提供
- ダイアログとウィザード (Dialogs and wizards) : ユーザとの複雑な関わり合いを構築するための枠組み
- フィールド支援 (Field assist) : ダイアログ、ウィザード、あるいはフォームのなかのフィールドの為に適切なコンテンツをユーザが選択するのを支援するガイドのクラスたち

一方ユーザからするとビュー (views) とエディタ (editors) がワークベンチのなかで見えるメインの要素である。パースペクティブ (perspective) はそれらが組み合わせられて画面上に配置された状態をいうが、どのパースペクティブも唯ひとつのエディタ・エリア (複数のエディタを含み得る) と、幾つかのそれにかかわるビューを持っている。ワークベンチはビューとエディタで作業する為の一連の操作が用意されている。各パースペクティブは特定のタイプのリソースでの特定の作業またはタスクを達成することを意図した機能のセットを提供している。例えば Java のパースペクティブは Java のソース・ファイルを編集する際に使うであろうビューたちを組み合わせしており、一方デバッグ (Debug) のパースペクティブは Java のプログラムをデバッグするときを使うビューたちを含んでいる。

パースペクティブはあるメニューとツールバーに何を出すかをコントロールしている。パースペクティブは可視のアクションのセットを決めており、ユーザはこれを変更してパースペクティブをカスタマイズできる。カスタマイズされたパースペクティブは保管し再度使用できる。

下図はパースペクティブ、ビュー、エディタなどの概念を示す具体的なスクリーンである。

図 1-2: Eclipse のスクリーン例



### 1.3節 Eclipse のインストール

Eclipse をダウンロードしてインストールするには、あらかじめ Java の実行環境 (JRE: Java Run-time Environment) が最低限必要である。Java の開発用に Eclipse を使うときは JDK も必須である。自分のコンピュータに JDK が存在していることをエクスプローラ等で確認する必要がある。無い場合は次の節に従って JRE または Java SDK をインストールする。

### 1.3.1 Java JRE のみのインストール

JRE はブラウザ等によって既にインストールされている場合が多いが、ない場合は以下のようにインストールすればよい。

1. [java.com のダウンロードのページ](#)から最新の Java の JRE をダウンロードする。これは jre-6u20-windows-i586-s (21 ビットの OS のユーザの場合) といった 16MB の実行ファイルであり、これを実行する。

### 1.3.2 Java EE 6 SDK のインストール

JRE だけでなく、SDK を含めてインストールする場合の方法を示す。ここではサーブレット開発のために現時点で最新の Java EE 6 SDK をインストールすることにする。これは 2009 年 12 月 10 日にリリースされたもので、サーブレット 3.0 に対応している。特にウェブ・アプリケーション向けの軽量なプロファイルの「ウェブ・プロファイル (Web Profile)」が新たに用意されている。これはウェブ・アプリケーションに必要な API セットを選択し軽量化したものである。しかしこれは、単純な Servlet コンテナだけではなく、トランザクション管理、セキュリティー、永続化も重要な要素として含まれており、EJB Lite や JPA、JTA といった API も含まれているのが特徴である。**Java EE 6 SDK をインストールするにはあらかじめ Java SE SDK のインストールが必要**である。

1. 最初に [Java SE のダウンロードのページ](#)から最新の JDK 6 のアップデートをダウンロードする (Download JDK の赤いボタンをクリックする)
2. ログオンのダイアログが出たら skip をクリックしてとばす。ダウンロードするファイルは jdk-6u20-windows-i586.exe という 78.5MB の実行ファイルで、これを適当なホルダにダウンロードする。
3. エクスプローラでこれを開いて実行させる。インストーラが開始する。インストール先は C:\Program Files\Java\jdk1.6.0\_20 などになる。
4. JDK がインストールされたら、DOS コマンドでのアクセスを容易にするために環境変数とパスを設定する。これはイ) マイコンピュータを右クリック→プロパティ→詳細設定→環境変数をクリックして変数名: JAVA\_HOME、変数値: C:\Program Files\Java\jdk1.6.0\_20\ を入力。ロ) システム環境変数→Path を選択→編集をクリックしてその文字列の最後に”;C:\Program Files\Java\jdk1.6.0\_20\bin”を追加する。ハ) システム環境変数→新規で、変数名 CLASSPATH、変数値 C:\Program Files\Java を登録する。これにより DOS のレベルで任意のディレクトリからこの SDK のコマンドやクラスが完全パスを指定しなくても用意にアクセスできるようになる。たとえばアクセサリのなかのコマンド・プロンプトのウィンドウで以下のように java コマンドを実行することでこれが確認できよう:

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\****>java -version
java version "1.6.0_20"
Java(TM) SE Runtime Environment (build 1.6.0_20-b02)
Java HotSpot(TM) Client VM (build 16.3-b01, mixed mode, sharing)

C:\Documents and Settings\****>
```

JDK をインストールすればウェブ・プロファイル (GlassFish) がインストールできる。ちなみに GlassFish という名前は "transparent development" あるいは "see-through" を意味し、オープン・ソースの透明性を強調して名付けられたと言われる。

5. [Java EE のダウンロードのページ](#) から Java EE 6 SDK (Web Profile) のダウンロードを選択する。これは `java_ee_sdk-6-web-windows` という 49MB の実行ファイルであり、これを実行する。
6. インストール・ディレクトリは `C:\glassfishv3` とする。Oracle ではこのディレクトリのことを `as-install-parent` と呼んでいる。また GlassFish がインストールされる `C:\glassfishv3\glassfish` ディレクトリを `as-install` ディレクトリと呼んでいる。
7. インストール中にサーバの管理設定を要求されるので、自分のユーザ名とパスワードを入力する。なお管理ポート番号は 4848、HTTP ポート番号は 8080 のままで良い。また更新 (インストーラが更新ツールをダウンロードし設定ができるようにする) の設定では、ファイアウォールを使っていない場合はプロキシ・ホストとポートを入力するが、通常はブランクのままで良い。更新ツールのインストールと更新ツールを有効にするの 2 つのチェックボックスをチェックのままとする。
8. JDK の選択はインストーラが選択したもの `C:\Program Files\Java\jdk1.6.0_20` とする。
9. 製品の登録をスキップすればインストールが終了する。
10. Oracle ではエンタープライズ・サーバのインストールが終了したら **`as-install-parent\bin` および `as-install\bin` の 2 つを PATH を追加することを推奨**している。

### 1.3.3 Eclipse のインストール

いよいよ Eclipse のインストールに入る。これに関しては多くのサイトがネット上で存在するのでそちらも参照して頂きたい。例えば [九州大学の記事](#) は分かりやすく推奨される。

1. [Eclipse のダウンロードのページ](#) から最新の Java EE 開発者向けのバージョンをダウンロードする。この資料作成時では Galileo のパッケージ (Eclipse 3.5 SR2) で、Windows 向けは 32 ビットの OS 用のものだけなので注意されたい。これは `eclipse-jee-galileo-SR2-win32.zip` という 190MB もの圧縮ファイルであり、適当な自分のホルダ、例えばデスクトップのダウンロードのホルダにダウンロードする。
2. このファイルをエクスプローラで選択右クリックして「すべて展開」を選択し、`C:\eclipse-jee-galileo-SR2-win32` というホルダに展開させる。そしてこのホルダ内に展開されている `eclipse` というホルダを `C:` に移動させる。即ち `C:\eclipse` が Eclipse のホルダになり、このホルダには `eclipse.exe` という実行ファイルが含まれている。
3. 次に日本語の表示が必要な人は Galileo の言語パックをダウンロードすることになる。このプロジェクトの [ダウンロードのサイト](#) から Galileo を選択すると言語ごとのパック zip ファイルのリストが出てくる。しかし日本語のところにあるこれらのファイルをひとつずつダウンロード・展開するのは面倒である。従ってそのベースとなっている Eclipse Galileo RC3 (3.5.0) 日本語化言語パック (サードパーティ版) を [Sourceforge のサイト](#) からダウンロードすることにする。このページに記されているようにまず自分の適当なフォルダに `NLpackja-eclipse-SDK-3.5RC3-blancofw20090531.zip` をダウンロードし、次にこのファイルをエクスプローラで選択右クリックして「すべて展開」を選択し、`C:\eclipse\dropins` ディレクトリ以下に展開する。なお Galileo の日本語パックのもう一つのベースである [Pleiades \(プレアデス\)のもの](#) もあるので自分でどちらが良いか判断すればよい。
4. Eclipse を起動させる: これはエクスプローラで `C:\eclipse\eclipse.exe` というアプリケーションを選択ダブル・クリックすればよい。なおデスクトップ上にショートカットを作って配置するほうが便利である。

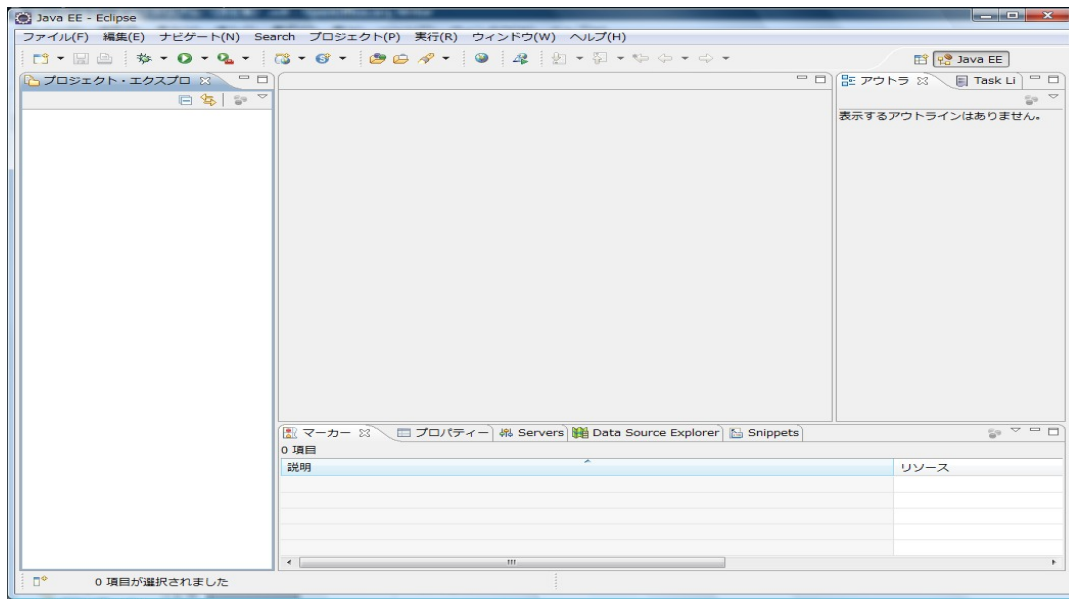
- ワークスペース・ラウンチャが開きワークスペースの場所を聞いてくるので、C:\eclipse\workspaceなどと指定する。なお「この選択をデフォルトとして、今後この質問をしない」というところはチェックしないほうが一般には良いとされている。
- 下図のようなウェルカム画面が出てくるので、右上の **workbench** を選択して開く。

図 1-3: Eclipse のウェルカム画面



- 最初のワークベンチ・ウィンドウは下図のようである。

図 1-4:最初のワークベンチのウィンドウ



8. JRE の確認と指定:これはメニュー・バーの「ウィンドウ(Window)」→「設定(Preferences)」→「Java」→「インストール済みの JRE(Installed JRE)」で「検索(S)」で C:\Program Files\java を選択し、見つかったものから選択する。通常は Eclipse が選択したもののままで良いが、**JRE ではなくて Java SDK を使うことが強く推奨されている**。なぜなら JDK には Java ライブラリのソース・コードが含まれており、デバッグがやりやすいからである。これまでの手順に従えば、C:\Program Files\Java\jdk1.6.0\_20 が見つかるはずであり、これをチェックする。
9. クラスパス変数の確認と指定:メニュー・バーの「ウィンドウ(Window)」→「設定(Preferences)」→「一般」→「ワークスペース」を開き、「自動的にビルド(B)」が選択されていることを確認する。次にこの設定のペインの「Java」→「ビルド・パス(Build Path)」を選択しソース及び出力ホルダを「プロジェクト(P)」とする。次に「Java」→「エディタ(Editor)」を選択し、「入出力中に問題をレポート(P)」をチェックする。そして「OK」をクリックして設定を終了する。

## 1.4節 基本的な使用法

### 1.4.1 Java 開発の為の Eclipse の基本要素

#### 1.4.1.1 Java のパースペクティブ

JDT では以下のパースペクティブがワークベンチに用意されている (Java EE のパースペクティブでは少し異なる)。これらは「ウィンドウ(W)」→「パースペクティブを開く(O)」→「その他(O)」で選択できる:

- **Java**:これは Java プロジェクトで作業するために設計されたパースペクティブで、これは以下のようなビューがある:
  - **パッケージ・エクスプローラ (Package Explorer)**:これはワークベンチ内でその Java プロジェクトの Java 要素の階層を示すためのビューである。この要素階層はそのプロジェクトのビルド・クラス・パ

スから引き出されている。各プロジェクトごとにそのソース・フォルダたちと参照されるライブラリたちがツリーで示される。内部および外部双方の JAR ファイルのコンテンツを開いてブラウズできる。

- 階層(Hierarchy)
- アウトライン(Outline)
- サーチ(Search)
- コンソール(Console)
- **Java ブラウジング**(Java 参照とも表示されている) :これは Java プロジェクトの構造をブラウズするために設計されたパースペクティブで、エディア・エリアと以下のビューを持っている:
  - プロジェクト(Projects)
  - パッケージ(Packages)
  - 型(Types)
  - メンバ(Members)
- **Java の型階層**:型階層を調べるために設計されたパースペクティブ。これは型、コンパイル単に、パッケージ、あるいはソース・フォルダで開くことができ、階層のビューとエディタを持っている。
- **デバッグ**:これは開発者が作ったプログラムのデバッグのために設計されたパースペクティブで、エディア・エリアのほかに次のようなビューを持つ:
  - デバッグ(Debug)
  - ブレークポイント(Breakpoints)
  - 例外(Expressions)
  - 変数(Variables)
  - 表示(Display)
  - アウトライン(Outline)
  - コンソール(Console)

#### 1.4.1.2 プロジェクト

Eclipse での **Java プロジェクト**というのは Java プログラムを作るためのソース・コードや関連ファイルたちを含んだものである。プロジェクトはそれに関連付けられた **Java ビルダ**を持っていて、Java のソース・ファイルたちに変更が出れば増分的にコンパイル出来る。Java プロジェクトはまたそのコンテンツの**モデル**を維持している。このモデルは Java 要素たちの型階層、参照、及び宣言に関する情報を含んでいる。この情報はユーザがその Java ソース・コードを変更するたびに定常的に更新される。内部 Java プロジェクト・モデルの更新は Java のビルダとは独立しており、特にコードの修正をする際は、オート・ビルドがオフになっていれば、このモデルは現在のプロジェクトのコンテンツを反映する。

Java プロジェクトの編集は次の 2 つの手段が使える:

- そのプロジェクトをソースのコンテナとして使う。これはシンプルなプロジェクトでは推奨される。
- そのプロジェクト内のソース・フォルダたちをソースのコンテナとして使う。これはより複雑なプロジェクトを編成する際は推奨される。このときはパッケージを幾つかのグループに分割できる。

#### 1.4.1.3 Java ビルダ

Java ビルダは Java 言語仕様に対応したコンパイラを使って Java プログラムたちを構築する。この Java ビルダは個々の Java ファイルが保管されるたびに増分的にプログラムを構築できる。コンパイラが検出したエラーは「警告」または「エラー」に区分される。警告が出てもプログラム実行には影響しないであたかも正しく書かれて



いるように実行する。Java 言語仕様で規定されているように、コンパイル時のエラーには Java コンパイラは常にエラーを報告する。他の何らかの問題に対しては Java コンパイラが警告を出すように設定できる。デフォルトの設定を変更するときは「ウィンドウ(W)」→「設定(P : Preferences)」→「Java(J)」→「コンパイラ(C)」→「エラー/警告の設定」のページで指定する。Java コンパイラはコンパイラのエラーがあっても Class ファイルを生成できる。しかし深刻なエラーが含まれているときは、Java ビルダは Class ファイルを生成しない。

#### 1.4.1.4 ビルドのクラスパス

ビルド・クラスパスは開発者の作ったソース・コードが参照しているクラスたちを見つけるのに使われるパスである。DOS の PATH などのファイル・システムのパスとは混同しないこと。コンパイル中はこのパスはプロジェクトの外部にあるクラスを探すのに使われる。ビルド・クラスパスは各プロジェクトに固有となる。プロジェクトのプロパティのなかではこれは「Java ビルドパス(Java Build Path)」として表現されている。

#### 1.4.1.5 クラスパス変数(Classpath variables)

ある Java プロジェクトの為のビルド・パスはソース・コードファイル、他の Java プロジェクト、及び JAR ファイルたちを含み得る。JAR ファイルはファイル・システムのパスを使って、あるいはネットワーク上の場所を参照する変数を使って指定できる。クラスパス変数により、JRE\_LIB といった変数名だけを使うことで JAR ファイルやライブラリを指定できるようになる。この変数を使えばチーム開発の場合は容易にビルド・パスを共有できる。

### 1.4.2 新しい Java プロジェクトの生成

Java プロジェクトの構成には 2 つの方法がある：

- そのプロジェクトをパッケージたちのコンテナとして使う。この構成では総ての Java パッケージはこのプロジェクトの内側に直接作られる。これはデフォルトの構成になっている。生成された CLASS ファイルたちは Java のソース・ファイルたちとともに保管される。
- そのプロジェクト内のソース・フォルダたちをパッケージたちのコンテナとして使う。このプロジェクト構成はより複雑なプロジェクトを編成する際は推奨される。パッケージたちはこのプロジェクトの内側に直接作られるのではなく、ソース・フォルダの内側に作られる。このときはパッケージを幾つかのグループに分割できる。ソース・フォルダたちはプロジェクトの子供として作られる。

新しいプロジェクトのデフォルトの構成は「ウィンドウ」→「設定」→「Java」→「ビルド・パス」の画面で変更ができる。

最初の構成での Java プロジェクトの生成は次のような手順となる：

1. メインのワークベンチのウィンドウから、「ファイル」→「新規」→「プロジェクト」をクリックすると (Java Ee のパースペクティブだと「Java プロジェクト」) 新規 Java プロジェクトのウィザードが開く
2. プロジェクト名のフィールドの自分の新しい Java プロジェクト名 (例えば MyProject など) を入力する
3. プロジェクト・レイアウトの個所で「プロジェクト・フォルダをソース及びクラス・ファイルのルートとして使用」が選択されていることを確認する
4. 「次へ(N)」をクリックすると Java 設定のページが開く
5. 「ソース」のタブではそのプロジェクトが唯一のソース・フォルダでデフォルトの出力フォルダであることを確認する。もし自分のプロジェクトが他のプロジェクトと依存性を持っているときは、「プロジェクト」のタブでそのプロジェクトを追加する。

6. 「順序及びエクスポート」のタブでは、「上部(Up)」「下部(Down)」ボタンで選択された JAR ファイルまたは CLASS フォルダたちのビルド・パス順序を変更できる
7. 「完了」ボタンをクリックする

### 1.4.3 トップ・レベルのクラスの生成

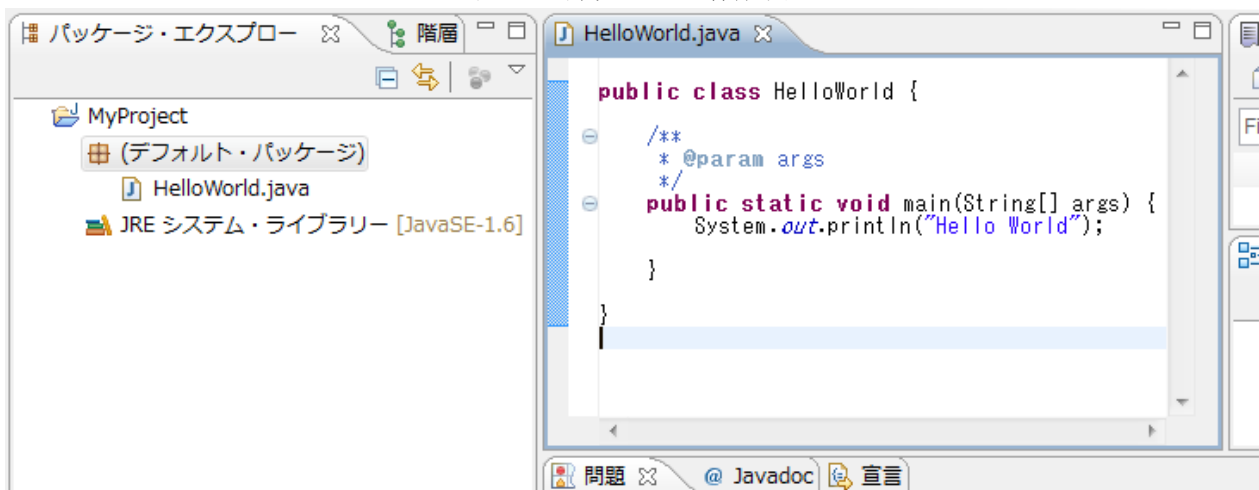
他の型に含まれていないクラスの生成は以下の手順となる:

1. パッケージ・エクスプローラ上でプロジェクトを選択右クリックして、「新規(W)」→「クラス」をクリックして新規 Java クラスのウィザードを開く
2. 必要ならばソース・フォルダ(D)を変更する
3. パッケージのフィールドでは新しいクラスを含めるためのパッケージを選択する。デフォルトのパッケージで良いときはそのままにする。デフォルト・パッケージではパッケージは使われない
4. 名前のフィールドでは新しいクラスの名前を入力する(例えば HelloWorld)
5. ボタンを使って修飾子を選択する
6. メソッド・スタブを選択する(例えば `public static void main(String[] args)`)
7. 完了ボタンを押すと次のようなソース・コードがエディタに表示される:

```
public class HelloWorld {  
  
    /**  
     * @param args  
     */  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
    }  
}
```

8. 編集画面で下図のように HelloWorld を編集し、エディタ上で右クリックし「保管(S)」をクリックすれば編集した内容が保管される:

図 1-5: 新規クラスの作成例



なおエディタに行番号表示を追加するには、

1. 「ウィンドウ(W)」→「設定」→「一般」→「エディター」→「テキスト・エディター」を選択
2. 「行番号の表示(B)」をチェック

すればよい。

#### 1.4.4 Java プログラムの実行

プログラムの実行は以下の手順となる：

1. パッケージ・エクスプローラ上で自分がたち上げたいと思う Java コンパイル単位またはクラス・ファイルを選択する。
2. ポップアップ・メニューから「実行(R)」→「Java アプリケーション」を選択する。あるいはワークベンチのメニュー・バーから「実行(R)」→「実行(R)」を選択、あるいはツールバー・メニューから「実行(R)」→「Java アプリケーション」を選択する。
3. これでプログラムが開始され、テキスト出力がコンソールに表示される。

Java コンパイル単位またはクラス・ファイルではなくて、プロジェクトから Java プログラムを開始することもできる。この場合はそのプロジェクトの main メソッドを持ったクラスが複数あればその中から選択する。

#### 1.4.5 添付ファイルのインポート

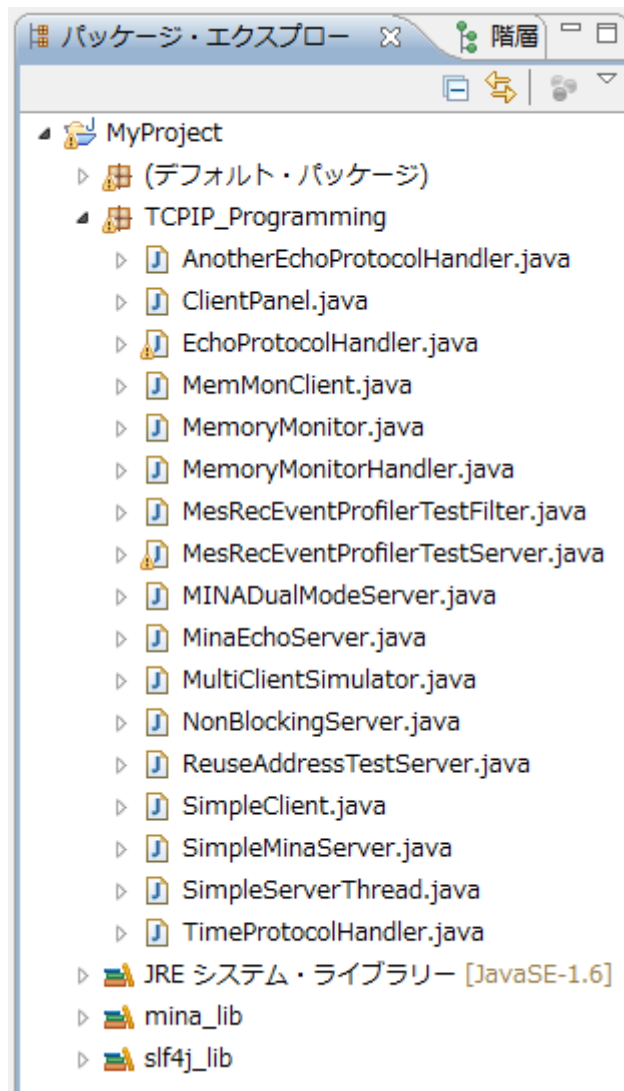
この資料には TCPIP\_Programming.zip というファイルが添付されている。これはこの資料で使われている java ファイルのパッケージである。これを自分のパッケージにインストールするには、次のようにする：

1. TCPIP\_Programming.zip を自分の適当なフォルダ(たとえば「ダウンロード」)にダウンロードする
2. パッケージ・エクスプローラ上の自分のプロジェクトを選択して右クリック→「新規(W)」→「パッケージ」を選択して TCPIP\_Programming という名前を入力して「完了(F)」
3. このパッケージを選択して、「インポート(I)」→インポート・ソースの選択で「アーカイブ・ファイル」を選択して「次へ(N)」をクリック
4. 「参照」で自分のコンピュータ上の「ダウンロード」の TCPIP\_Programming.zip というファイルを選択
5. 「あて先フォルダ」は MyProject\TCPIP\_Programming など、自分のプロジェクトのあとにパッケージ名を追加
6. 「完了」をクリック
7. そうすると自分のプロジェクトに TCPIP\_Programming というサンプル・プログラムが入ったパッケージがパッケージ・エクスプローラ上に表示される筈である

なお添付のプログラムには SLF4J と MINA のライブラリが必要であるので、「エコーバック・サーバのプログラム例」の「[SLF4J パッケージのダウンロード](#)」と「[MINA パッケージのダウンロード](#)」で示した手順でこれらのライブラリをインストールする必要がある。

下図はインポートした結果のパッケージ・エクスプローラ表示例である：

図 1-6: インポート例



## 第2章 TCP/IP

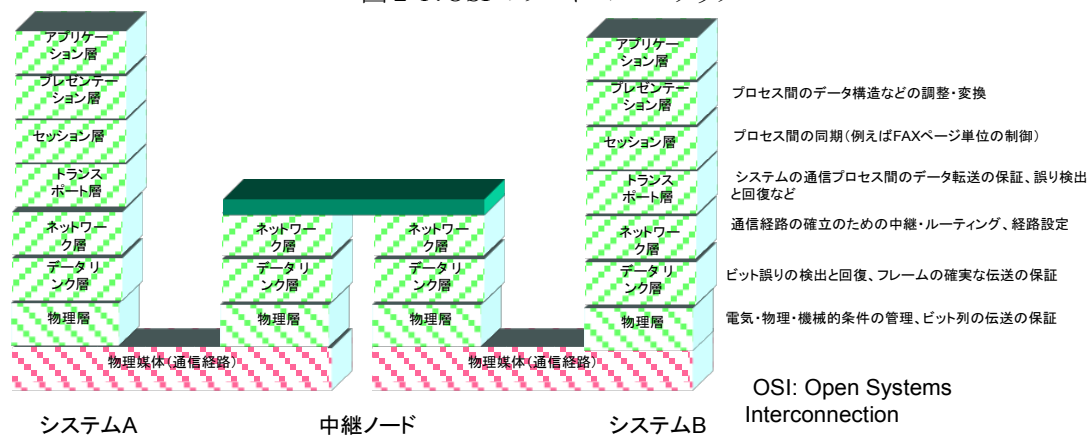
ウェブ・アプリケーションは TCP/IP 網をサーバクライアント間の通信のベースにしている。障害の多くは通信のレイヤで起きているので、TCP/IP を正しく理解することが重要である。

### 2.1節 通信プロトコル

#### 2.1.1 OSI のプロトコル・スタック

通信プロトコルの開設で良く引用されるのが OSI の参照モデルである。OSI そのものはあまり普及することなく TCP/IP が現在世界の主流となっている。しかし OSI は当時の国際組織の CCITT (現在の ITU-T) で開発された優れたもので、1977 年ごろから開発が始まっていた。TCP/IP はどちらかといえば学術研究機関の為のネットワーク (インターネット) のためのものであり、TCP/IP に比べると不十分な部分が多い。例えばネットワーク層では IP アドレスは当初は 32 ビットのみで、非常にシンプルなものである。これに比べて OSI のネットワーク層では送受信のアドレス長は可変であり、いろんなアドレスに対応でき、また接続型と非接続型があり、更にフロー制御が導入されている。

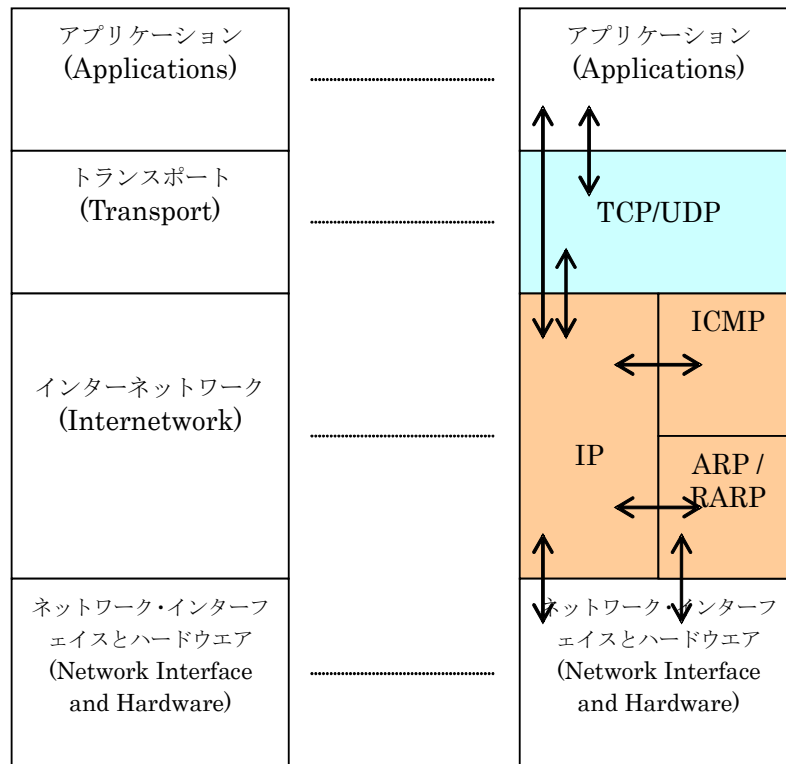
図 2-1: OSI のプロトコル・スタック



#### 2.1.2 インターネットのプロトコル・スタック

インターネットの世界では下図のように 4 層で、より簡単な構成になっている。ただ核となるインターネットワーク層にはサブレイヤが存在し、またトランスポート層は TCP と UDP の 2 つが用意されている。下位の層 (データ・リンクと物理層) はネットワーク・インターフェイスとハードウェアとして、プロトコル・スイートには含めていない。

図 2-2: インターネットのプロトコル・スタック



読者が自分の PC で DOS コマンドを使ってアクセスできるプロトコルは下図の色付けした部分である。一般のアプリケーションは TCP または UDP の上に乗っており、ソフトウェアの開発者は TCP 及び UDP とアプリケーション間のインターフェイスであるソケット層を介してアクセスすることになる。

#### アプリケーション(Applications)層

同じあるいは別のホスト上にある別のプロセスと協調するユーザのプロセスをいう。例えば SMTP(Simple Mail Transfer Protocol)、FTP(File Transfer Protocol)、Telnet(リモート・ターミナルのプロトコル)、HTTP(Webドキュメントの転送で使われるプロトコル)などがある。

#### トランスポート(Transport)層

エンドとエンドとのデータの転送をつかさどる。プロトコルとしては TCP (Transmission Control Protocol: コネクション・オリエンテッド)と UDP (User Datagram Protocol: コネクション・レス) がある。これらは別途説明する。

#### インターネットワーク(Internetwork)層

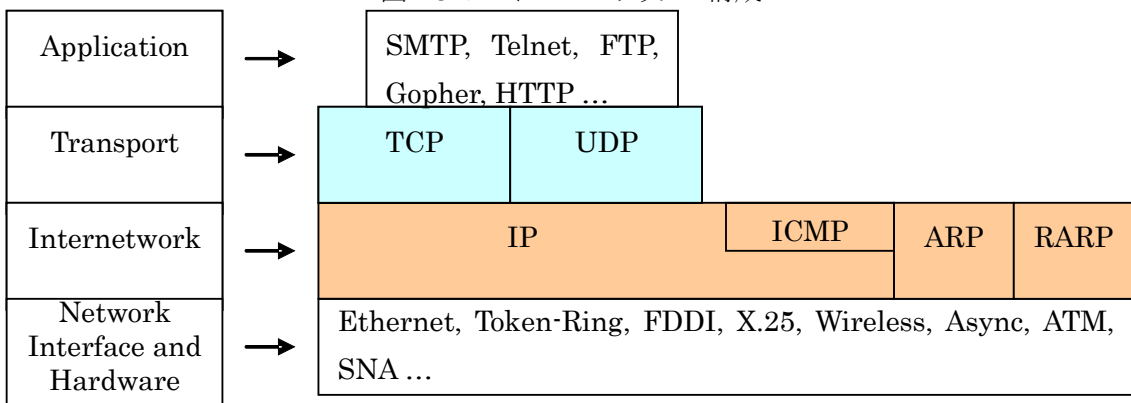
インターネット層あるいはネットワーク層とも呼ばれる。このインターネット層がインターネットの「バーチャル網」のイメージを提供するものである。すなわち上位層から見ると、この層はその下にあるネットワーク・アーキテクチャを遮蔽してくれ、1つのネットワークとして取扱うことができる。IP(Internet Protocol)はこの層で一番重要なプロトコルである。このプロトコルはコネクション・レス(データの転送に先立ち接続手順を踏まない。すなわちパケットはそこにふられた宛先をもとに転送される)であり、下位層の信頼性を仮定してはいない。IPは信頼性(いわゆる QoS)やフロー制御、誤り訂正などの機能は持たない。これらの機能は上位層で、即ちトランスポート層で TCP が使われる場合はトランスポート層で、UDP が使われる場合はアプリケーション層で用意しな

なければならない。IP 網でのメッセージのことを IP データグラム(IP Datagram)と呼ぶ。「IP パケット」というよりも、インターネットの世界では IP に関しては「IP データグラム」という言葉を使う。

ネットワーク・インターフェイス(Network Interface)層

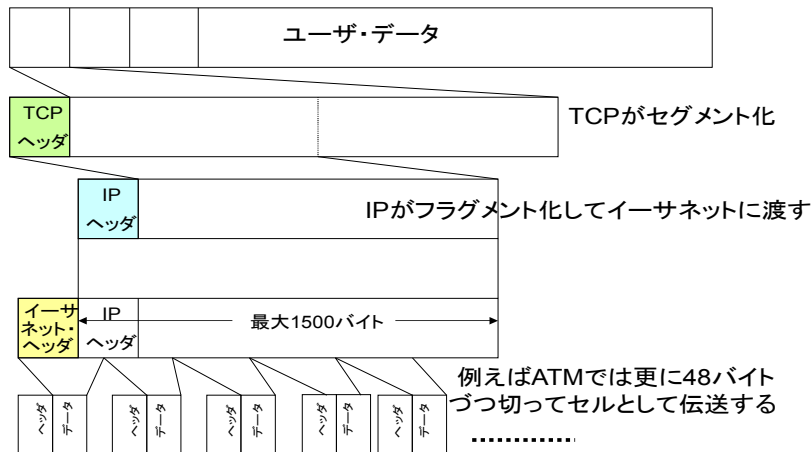
これはまたリンク層(Link Layer)とかデータ・リンク層(Data-link Layer)とも呼ばれる。この層は実際のネットワークのハードウェアとのインターフェイスを構成する。この層は信頼度を持たせる場合も持っていない場合もある。またパケット・ベースの場合もあればストリーム・ベースの場合もある。TCP/IP はここでの如何なるプロトコルも規定はしていないが、殆ど総てのネットワーク・インターフェイスが使える。IEEE 802.2 (いわゆるイーサネット)、X.25 (これは高信頼性のネットワーク・インターフェイスである)、ATM、FDDI、パケット無線網や IBM の SNA などがその例である。実際の層間のインタラクションは前図で矢印で示してある。

図 2-3: プロトコル・スタックの構成



各層ではそのプロトコルに従ってデータを送受信するが、実際のもとのデータは下図のように分割される。従ってあるコンピュータがメッセージを送信したとしても、それは各層で分割されて送信されている。

図 2-4: データの分割送信



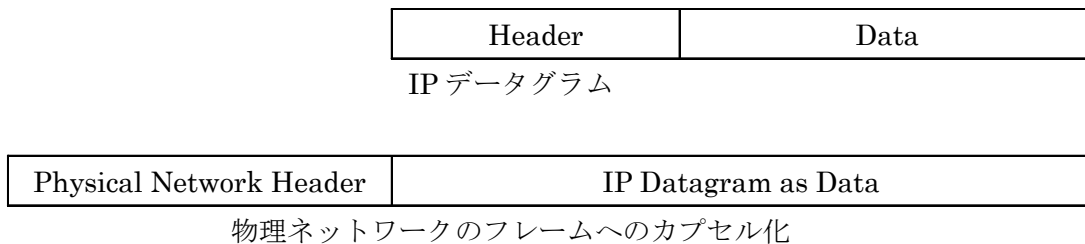
## 2.2節 IP

### 2.2.1 IP データグラム

IP データグラム(インターネット・データグラムともいう)は、インターネットのプロトコル・スイートの基本的な転送パケットであり、IP のための情報を有するヘッダと、上位層だけが関与するデータ部から構成されている。この IP データグラムは下位層ではさらにカプセル化されて転送される。たとえばイーサネットの場合は最大 1500 バイト長のデータ部分にカプセル化される。だからといって IP データグラムの長さを 1500 に規定しているわけではなく、IP 側で分割(Fragmentation)と再組立(Re-assembly)の機能が用意されている。IP 仕様書は最大長を規定しない代わりに「総てのサブネットワークは少なくとも 576 バイト長のデータグラムが処理できること」としている。

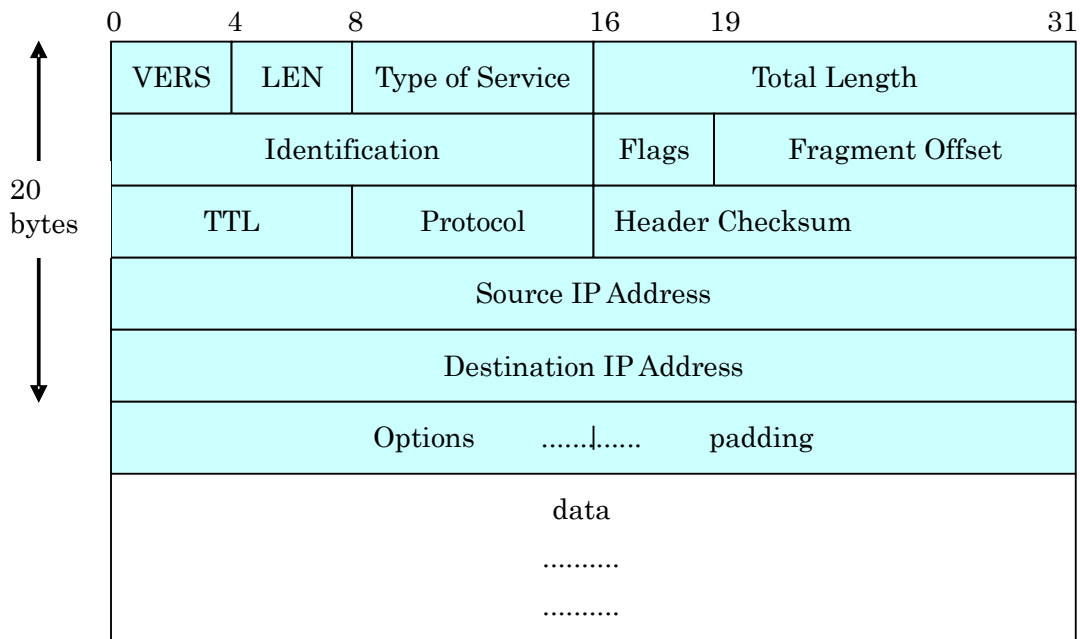
分割された各フラグメント(断片)には各々もとのデータグラムからコピーされたヘッダが付されており、通常の IP データグラムとして宛先に送られる。つまり下位層はこれを意識する必要がない。ただしフラグメントがひとつでも欠落した場合は、データグラム全体が欠落したとみなされる。これは IP は再送手順を持っていないためである。相手のホストは残りのフラグメントを単純に廃棄する。

図 2-5: IPデータグラムと物理ネットワークのフレーム



IPv4 データグラムのフォーマットは以下のようにになっている。ヘッダ部分は最小 20 バイト長である。





- **VERS:** IP プロトコルのバージョンを示す。現在のバージョンは 4 である。普及し始めている IPv6 (以前は IPng と呼ばれた) はバージョン 6 のことである。バージョン 5 も実験的に存在している。このバージョン値よりも若い番号のデータグラムを受信したときは破棄する。
- **LEN:** 32 ビットを単位としたヘッダ部の長さを示す。最小値は従って 5 である。
- **Type of Service:** この IP データグラムに要求されるサービス品質を示す。このフィールドは更に細分化されている。
- **Total Length:** このデータグラムの全体の長さ (ヘッダ + データ) をバイト単位で示す。
- **Identification:** 送信元が付すこのデータグラム固有の識別子である。これはフラグメント化されたデータグラムを再組立するときに使う。つまり受信側は同じ識別子のものをつなぎ合わせる。
- **Flags:** フラグメント処理のためのフラグである。
- **Fragment Offset:** フラグメント化されたデータグラムに付ける。つまりもとの (つまり一番最初の) データグラムのデータ部分の何番目のところからのフラグメントであるかを 64 ビット単位で示し、最初のフラグメントはこの値が 0 である。受信側は同じ識別子のフラグメントをこの数字の若いフラグメントから順にフラグの中のもの MF がゼロのものまで並べて再組み立てする。途中で欠落があれば全体を廃棄する。
- **Time to Live:** このデータグラムが相手に到達するまでに許容された最大時間を秒で示す。各ルータは自分のところでもかかった処理時間を引いていく。実際のところ 1 秒もかかるルータは存在しないので、この場合は 1 を引く。従ってこの値は通過時間の計測よりも何ホップかかったかの判断に使える。この値がゼロになったらルータはこのデータグラムを廃棄し、不要なデータグラムがいつまでもネットワーク上に浮遊するのを防止する。最初の値はこのデータグラムを作った上位プロトコルが設定する。
- **Protocol:** IP がこのデータグラムを渡すべき上位層プロトコルを示す。IP 層の上には TCP や UDP などが乗っているので、どのプロトコルに渡すかの情報により受信データグラムを所定のプロトコルに渡すことができる。
  - 0 Reserved
  - 1 Internet Control Message Protocol (ICMP)
  - 2 Internet Group Management Protocol (IGMP)

- 3 Gateway-to-Gateway Protocol (GGP)
  - 4 IP (IP encapsulation)
  - 5 Stream
  - 6 Transmission Control (TCP)
  - 8 Exterior Gateway Protocol (EGP)
  - 9 Private Internet Routing Protocol
  - 17 User Datagram (UDP)
  - 89 Open Shortest Path First (OSPF)
- **Header Checksum:** ヘッダ部分だけのチェックサムである。16ビット・ワードを1の補数で和をとりそれを1の補数とする。ルータはヘッダ部分が有効かをまず確認し処理できるので高速化が図れる。不正なデータグラムは直ちに廃棄する。
  - **Source IP Address:** このデータグラムを送信しているホストのIPアドレス。
  - **Destination IP Address:** このデータグラムの宛先ホストのIPアドレス。
  - **Options:** これは可変長である。すなわち1バイトのtypeだけのものとtype, length, option dataの組み合わせの最大255バイト長のものまで存在し得る。
  - **padding:** オプションが使われたときに、次の32ビットの境界に合わせるためにオールゼロで詰める。
  - **data:** これは上位プロトコルに渡すべきメッセージである。

### 2.2.2 IPv6 データグラム

下図はIPv6ヘッダがIPv4に比べてどのように変更されているかを示している。大きな相違はアドレス長が128ビットと大きく拡張されていること、またフラグメントが無くなっていることである。フラグメント化は上位のレイヤで行い、もはやルータは関与しなくなっている。

図 2-7: IPv4 と IPv6 のヘッダ部分の相違

## IPv4

Ver.	header	TOS	Total length	
identification			flag	fragment offset
TTL	Protocol		Checksum	
32 bits Source Address				
32 bits Destination Address				

## IPv6

Ver	Traffic Class	Flow Label		
Payload Length		Next Header	Hop Limit	
128 bits Source Address				
128 bits Destination Address				



変更された箇所



削除された箇所

- Ver.:これはIPv6では6である。
- Traffic Class:これはIPv4のTOS相当で、異なったクラスまたは優先度を指定する。
- Flow Label:これは20ビットの長さを持ち、ソースのノードがパケットのシーケンスのラベリングに使用する。
- Payload Length:IPv4のTotal Lengthに相当。
- Next Header:IPv4のProtocolに相当、カプセル化されたプロトコルを指定するのに使われる。またこれを使ってヘッダの拡張が出来る。拡張されたヘッダたちの説明は省略する。
- Hop Limit:IPv4のTTL相当。

## 2.2.3 IPのアドレス

IPのアドレスはIPv4では32ビット、IPv6では128ビットの長さを持つ。これらは一般的には2進数で表現するよりも8ビットずつ区切って10または16進数で表現されている。

### 2.2.3.1 IPv4のアドレス

インターネット上のホスト（ノードとも言い、コンピュータと考えても良い）を識別するために、各ホストにはIPアドレス（あるいはインターネット・アドレスともいう）が付与される。ルータの

ようにそのホストが複数のネットワークと接続している場合は、そのホストはマルチ・ホーム (multi-homed)だといひ、各ネットワーク・インターフェイスごとにひとつのIPアドレスが付される。32ビット長のIPアドレスは、通常ドットつき10進数表記(Dotted Decimal)法で表現される。たとえば128.1.10.1というIPアドレスは後述のとおり128.1がネットワーク番号で10.1がホスト番号となる。このIPアドレスをバイナリで表現すると、

10000000 00000001 00001010 00000001

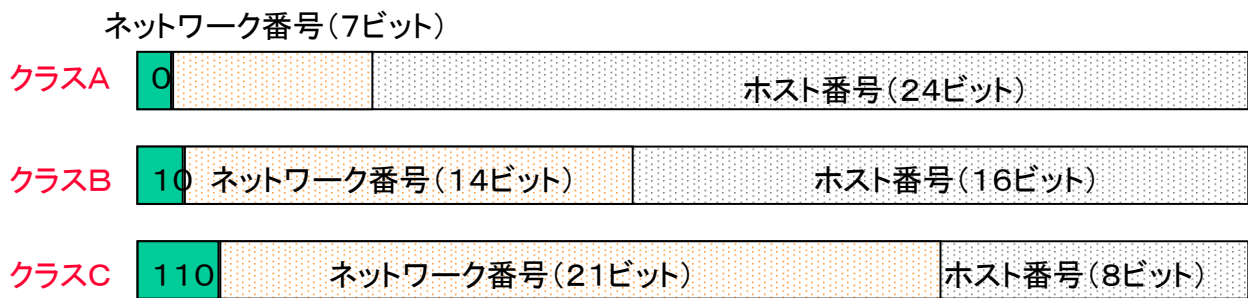
となる。

IPアドレスは前に説明したとおりネットワーク番号とホスト番号との組み合わせである。すなわち：

IPアドレス = <ネットワーク番号><ホスト番号>

ネットワーク番号はIPアドレスの一部であり、インターネット・ネットワーク・情報センタ（日本の場合はJPNIC: Japan Network Information Center）が一元的に管理し、インターネット上において唯一で重複が生じないように付与されている。ネットワーク番号とホスト番号のビット長の割当ては、当初は下図のようにクラスAからCまでの3つのみだった：

図 2-8: IPv4 のアドレス・クラス



しかしながら現在では通常の企業あるいは組織のネットワークはファイヤウォールが置かれており、そのファイヤウォールの外部ネットワーク・インターフェイスはひとつである。それ以外のウェブ・サーバのような外部に公開するものを含めても多くのIPアドレスを使わなくとも良くなっている。加えて割当て可能なIPv4アドレスの枯渇の問題もあり、現在はクラスレスのアドレス方式、即ちCIDR (Classless Inter-Domain Routing: サイダーと発音する)が広く導入されている。例えば

11001010 00001100 00011110 10000000

というIPアドレスが、202.12.30.128/26と表記されていたとする。この/26はプレフィックス(ネットワーク・アドレスのビット長)であり、この場合は6ビット分(64台)が内部のホストに割り振れることになる。但しオールゼロは自分自身、オール1はブロードキャストである。CIDRはアドレス枯渇の問題だけでなく、バックボーンのルータの持つルーティング・テーブルの増大という問題にも有効な手段である。

なお次のような特別なIPアドレスが定められている：

- ループバック・アドレス(127.0.0.0/8):これは"localhost"という名前が一般的である。これはネットワーク・インターフェイスの出口のところで折り返すことで、そこまでの機能を確認する為に使われる。
- プライベート・アドレス:これは社内LANのように、グローバルなインターネットと分離して割り当てることが可能なアドレスであり、ルータはこのアドレスへのIPデータグラムをグローバルなインターネットに通さない。特にクラスCのプライベート・アドレスは、住宅内用として広く利用されている。

表 2-1: プライベート・アドレス

クラス	開始	終了
A	10.0.0.0	10.255.255.255
B	172.16.0.0	172.31.255.255
C	192.168.0.0	192.168.255.255

- マルチキャスト・アドレス:これは OSPF、マルチキャスト、あるいは実験用として使用される。

表 2-2: マルチキャスト・アドレス

クラス	開始	終了
D	224.0.0.0	239.255.255.255

### 2.2.3.2 IPv6 のアドレス

IPv6 ではアドレス長が 128 ビットと長いので、16 ビットずつのフィールドに区切り 16 進数表記し各々をコロン(:)でつなぐ表記法がとられている。例えば:

2001:0000:1234:0000:0000:C1C0:ABCD:0876

は、

2001:0000:1234:0000:0000:c1c0:abcd:0876 (大文字と小文字は区別しない)

2001:0:1234:0:0:c1c0:abcd:0876 (先頭のゼロは省略可)

2001:0:1234::c1c0:abcd:0876 (ゼロのフィールドが続いているときは::で表現できるが、アドレスあたり 1 回のみ。例えば FF02:0:0:0:0:0:1 は FF02::1 と、0:0:0:0:0:0:1 は::1 と表現できる)

またブラウザなどでの URL では、これはブラケット([])で囲む。例えば:

http://[2001:1:4f3a::206:ae14]:8080/index.html

のように表現する。ただしこれは面倒であり、一般には完全修飾ドメイン名 (FQDN)が使われる。

128 ビットのアドレスは IPv4 と同じく前半部と後半部に分けられ、前半はネットワーク・プレフィックス、また後半はインタフェース識別と呼ばれており、左から次のような構成になっている:

- 3 ビット: 001 (IANA から各レジストリに割り当てられるブロックは全アドレス空間の 1/8 ということになる)
- 12 ビットの TLA (Top Level Aggregators: ティア 1 プロバイダ識別)
- 8 ビット: 予約
- 24 ビットの NLA (Next Level Aggregators: 次の階層のプロバイダ識別)
- 12 ビットのサイト・サブネット
- 64 ビットのインターフェイス識別

IPv6 の特別なアドレスとしては以下のものがある:

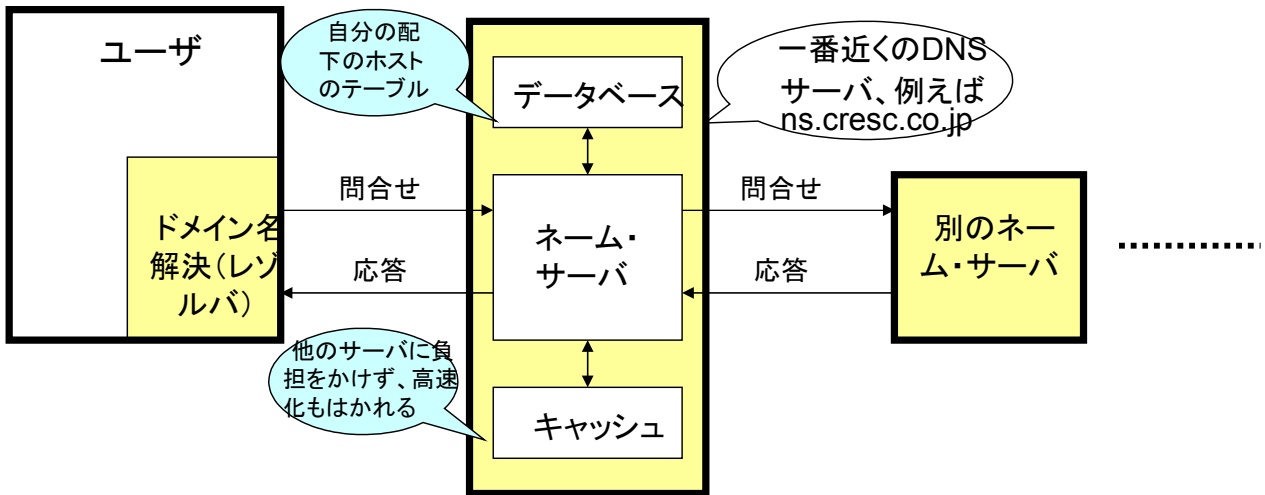
- ループバック・アドレス (0:0:0:0:0:0:1 (“::1” 圧縮表現))
- マルチキャスト・アドレス: 複数のノードに割り当てられるアドレス。このアドレスあてに送信されたパケットは、複製されてこのアドレスに参加しているノードに配送される。ffxx:: で始まるアドレス。

### 2.2.3.3 ドメイン名と IP アドレスのマッピング

インターネットを利用する場合、IP アドレスで相手を直接指定することは殆どなく、もっぱらドメイン名が利用される。ドメイン名システムは IP アドレスとの対応をデータベースとして蓄積している。つまりどこかのネーム・

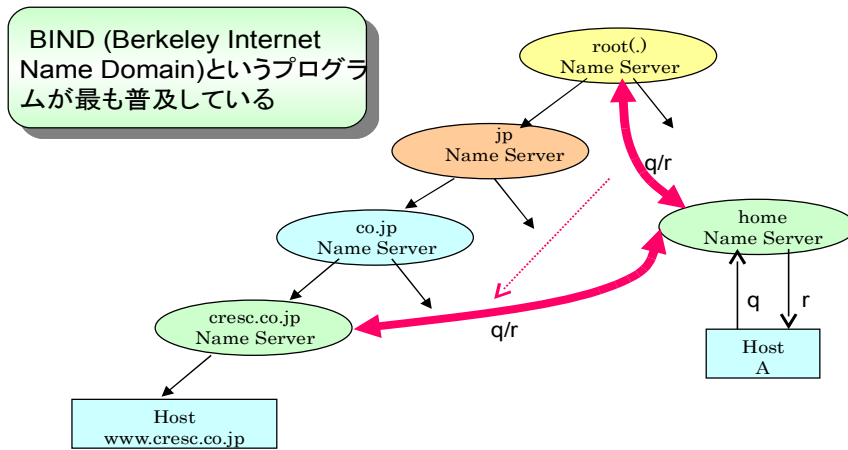
サーバがこれを知っている。ユーザのプログラムがあるドメイン名の IP アドレスを知りたいときは、その変換を自分が登録している(ホームの)ネーム・サーバにこれを問い合わせる。そのネーム・サーバは自分のデータベースにそのドメイン名をもっていなければ他のネーム・サーバに問い合わせる。ドメイン名システムは階層化されており、ルート・サーバからトリーをたどりながら最終的にターゲットのドメイン名が登録されているネーム・サーバにたどりつくことになる。階層化のゆえにドメイン名の検索は比較的簡単なアルゴリズムとなる。

図 2-9:ドメイン名解決システム



下図のように目的とするドメイン名にたどり着くまでに多くのサーバを介することとなり、UDP を使って高速化を図ってはいるが検索時間がかかることになる。ネーム・サーバはその為にキャッシュを持っており、一度検索したドメイン名はある期間これを保持することで効率と速度をあげる。

図 2-10:アドレス解決のツリー



問合せを受けた最初のサーバは、まずルートネームサーバ(世界に 13 個存在する)に照会する。すると、「そのドメイン名ならこのサーバに権限を委譲(delegate)している」というデータが返ってくる。そこで、指示されたサーバに再び照会し直すと、さらに「そのドメイン名ならこのサーバに権限を委譲している」という回答が返ってくる。こうして権限を委譲されたサーバをたどっていくことで、最終的に「そのドメイン名の IP アドレスは xxx.xxx.xxx.xxx」と答えてくれるサーバに行き着くことになる。但し通常はキャッシュのおかげで途中を省略できる。

## 2.2.4 IP ルーティングの基礎

### 2.2.4.1 ローカル・ホストでの経路設定

各ローカル・ホストは到来したデータグラムの宛先が自分宛かどうかを調べる。

- Yes そのデータグラムを上位層プロトコルに渡す
- No このデータグラムは別のホスト宛である。対応は自分に設定された IP 転送フラグ (ipforwarding flag) の値によって異なる。
  - true このデータグラムは外に発信されるデータグラムゆえ以下に示すアルゴリズムで次のホップ (next hop) に回送される。
  - false このデータグラムは廃棄される (他のホストが受理する)。

外に発信される IP データグラムは、宛先 IP アドレスをもとに何処に送れば良いかを決定する IP 経路設定アルゴリズムを通る。

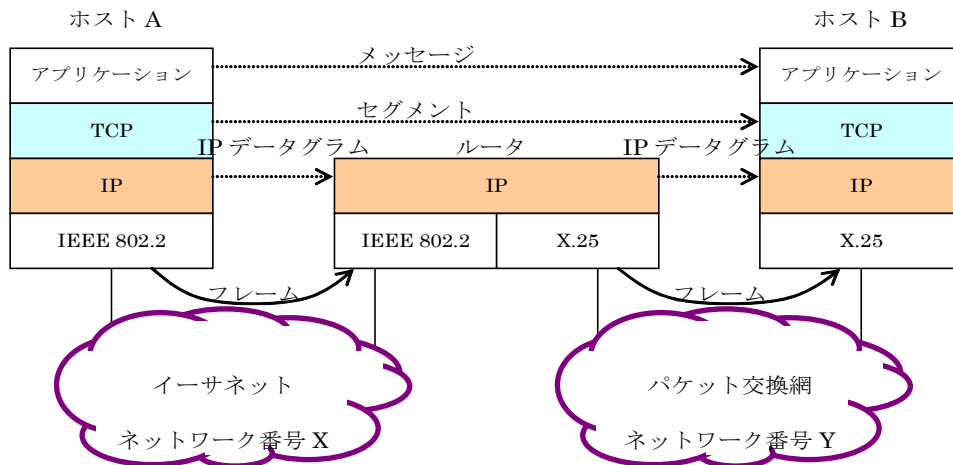
- そのホストが持つ IP 経路設定テーブル (IP routing table) に宛先 IP アドレスと合致するアドレスが登録されているなら、そのデータグラムはその登録されているアドレスに送信される。
- 宛先 IP アドレスのネットワーク番号が、そのホストが持っているネットワーク・アダプタのネットワーク番号と同じであれば (宛先のホストがそのネットワーク・アダプタでつながっているネットワーク上にある)、そのデータグラムは宛先 IP アドレスと合致するホストの物理アドレスに向けて送信される。
- それ以外の場合は、そのデータグラムはデフォルトのルータ (default router) にむけて送られる。

これが基本のアルゴリズムで、一般の PC を含む総ての IP 実装に必要とされるものであり、基本経路設定機能を実行するには充分である。このアルゴリズムは 2 つのネットワークが 1 つのルータで接続されているような簡単なインターネットワークでは事足りるが、複雑な構成の場合は高度なプロトコルが必要となり、それらは後ほど紹介する。

### 2.2.4.2 ルータでの経路設定

ルータはプロトコル・スタックで表現すると下図のようになる。この図ではイーサネット上にあるホスト A とパケット交換網上にあるホスト B をルータが中継している。

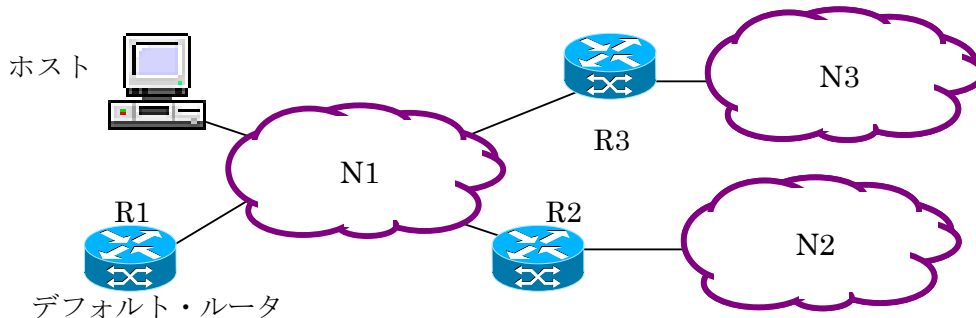
図 2-11: ルータの IP 中継



この図では、ネットワーク番号 X 上にあるホスト A からネットワーク番号 Y 上にあるホスト B にむけてルータを介してメッセージが転送される。アプリケーション層でのメッセージは TCP 層ではセグメントとして、IP 層では IP データグラムとして、そして最下位層ではフレームとして転送される。中間にあるルータはネットワーク・インターフェイス層と IP 層までを実装していることに注意されたい。

ここでデフォルト・ルータという言葉が出てきたので、簡単に図示する。ネットワーク N1 上のあるホストがネットワーク N3 上のあるホストに送信する場合、自分が含まれているネットワーク N1 宛ではないので、その IP データグラムをどちらかのルータに送らねばならない。一番簡単な方式はデフォルト方式で、自分の属しているネットワーク以外のネットワーク宛のデータグラムは R1 に送るといものである。R1 はデフォルト・ルータでそれには N3 宛のデータグラムは R3 経由と登録されていなければならない。この場合データグラムは R1 と R3 経由で相手のホストに届けられる。インターネットワークの構成の変更があった場合は R1 のテーブルを変更しさえすれば良い。他のネットワーク宛は総てデフォルト・ルータ経由とする方式をデフォルト経路設定方式あるいは固定方式と言う。

図 2-12: デフォルト・ルータ



デフォルト経路設定方式に対比されるものとして、ダイナミック経路設定方式がある。ホストは経路設定テーブル(ルート・キャッシュとも言う)を持つ。自分が持っている経路設定テーブルに N3 宛のデータグラムはルータ R3 経由だと登録されていれば、R3 宛に送れば良い。そうでなければデフォルトの R1 に送れば良い。R1 はそのうち N3 宛のデータグラムは R3 へ直接送ったほうが良いと判断して、その情報を後ほど説明する ICMP メッセージで教えてくれる。そのときはホストは自分の経路設定テーブルを変更すれば良い。ICMP (Internet



Control Message Protocol) というのは IP パケットではあるが、ルータやホストの間で IP 動作の補佐として使われる制御メッセージである。これは IP と組で実装すべきプロトコルであり、別途説明する。

## 2.2.5 ルータの基本アルゴリズムと構成

インターネットは自律分散のシステムである。インターネットは毎日その規模を増やし、また変化している中で、どうしてはるか遠くの相手に IP データグラムが届くのであろうか。その核となるのがルータである。これは電話網で言えば交換機に相当するノードである。そのアルゴリズムについて簡単に説明する。

ルータは IP (ネットワーク) 層のデバイスである。その役割はネットワークあるいはサブネットを相互接続し、その間での IP データグラムを正しく中継することである。「正しく」というのは不要なデータグラムを通さないことにも通じる。中継にあたってはいくつかの選択肢があることもある。ルータはそれらの経路のうち最適なものを選択し、そのルートへ IP データグラムを転送する。また宛先のネットワークによっては再フラグメント化が必要な場合もあり、そのときはそのネットワークに対応したフラグメント化を行う。ルーティングの基本機能は次のようである:

到来 IP データグラムの宛先 IP アドレスが、自分のローカルのホストの IP アドレス以外のものであれば、それは通常の転送すべき IP データグラムである。

この転送(forwarding)IP データグラムを次にどこに転送するか(Next hop: 次ホップ)を選択するのが IP ルーティングアルゴリズムである。

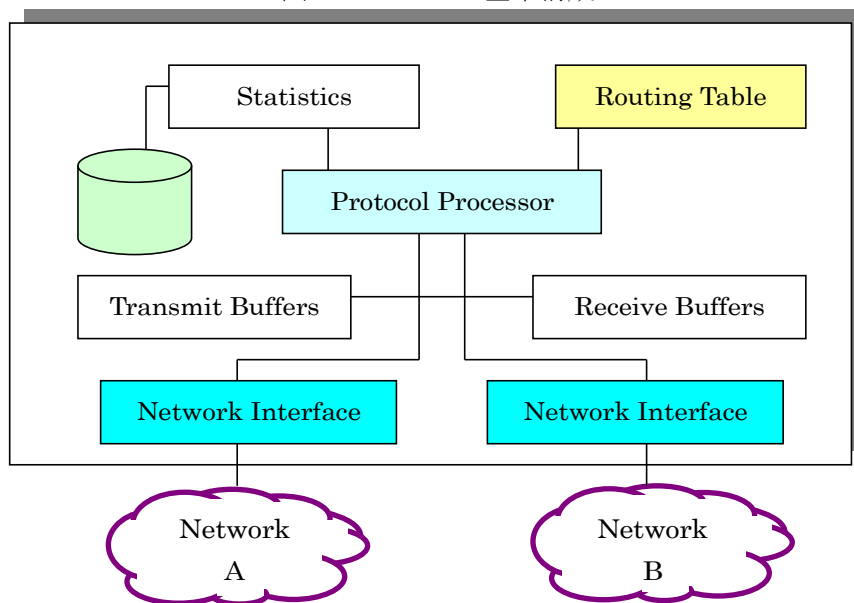
その他のルータの重要な役割として以下のものがある。

- **ブロードキャスト・IP データグラムのフィルタリング:**  
ブロードキャストやマルチキャストは有用なコンセプトではあるが、これが幾つものネットワークに伝播したり、極端な場合にはネットワークの中を回ってついにはネットワークの機能を失わせてしまう危険性を伴う。例えば  $\Delta$  型に 3 つのネットワークが接続されたインターネットワークの場合、ルータがブロードキャストを許してしまうとこのような事態が発生する。また大きなインターネットワークにおいてルータがこれを通してしまふとそのなかでブロードキャストのパケットだらけになってしまい、ネットワークのスループットを極端に低下させてしまうことになりかねない。これをブロードキャストの嵐(Broadcast Storm)と呼ぶ。ルータは従ってブロードキャストの IP データグラムを通してはならない。
- **通過 IP データグラムの制限:**  
不必要なパケットの通過を制限して、ローカルのセキュリティを図ることができる。例えば所定の IP アドレスへの(からの)IP データグラムしか通さない、所定のプロトコル番号のデータグラムしか通さないなどである。上位の層での制限(例えば TCP におけるポート番号による制限)もあるが、これはルータの機能ではなく、しばしばファイアウォールと呼ばれるものの役割であるが、高度なルータにはそのような機能を持たせたものも存在する。
- **速度調整機能:**  
ルータが相互接続しているネットワークは、通常帯域が等しいことはない。従って広帯域のネットワークから狭帯域のネットワークに IP データグラムを転送する場合、パケットの欠落を防ぐために通常バッファを置き、その差を吸収する。バッファがいっぱいになりそうときは ICMP の発信制限メッセージを送信元に通知する。
- **統計情報の取得:**  
通過するパケットに関する種類の統計量を集積することができる。これはネットワークの管理、課金、障害時の原因調査などに有用である。
- **ネットワーク管理:**

ルータの管理や設定、ルータが取得した統計量の取得などを遠隔(リモート)で行えることが必須である。一般には SNMP(Simple Network Management Protocol)が使われる。

ルータはこれらの機能を有するために、下図のような構成となっている。基本的にはネットワーク・インターフェイスを持ったコンピュータでソフトウェア処理で実現できるが、専用のハードウェアで高速化と低価格化を図ったものが主流である。

図 2-13:ルータの基本構成



さて話を経路設定に戻そう。ルーティングは各ルータが自分が持っているこの図のルーティング・テーブルをもとに行う。しかしながら IP の基本思想は自立分散のシステムである。発祥が米軍のシステムであることから、ネットワークの一部に変更や障害が発生しても、正しく相手に IP データグラムが転送できるようにすることが目的であったことは、周知のことである。そのような場合に各ルータのルーティング・テーブルをどう協調しながら更新するかがルーティング・プロトコルの原点である。

UNIX ベースのシステムではルーティングのデーモンが通常 `routed` と `gated` の二つが実装されている。「ルート D」と呼ばれるデーモンはオートノマス・システム(ある管理者の管理下にあるネットワークの集まり)内で使われるもので、RIP(Routing Information Protocol)なるプロトコルが使われている。「ゲート D」と呼ばれるデーモンはオートノマス・システム内および外に使われ、OSPF(Open shortest Path First Protocol)と BGP(Border Gateway Protocol)などにも対応している。ここではこれらの 3 種類のプロトコルをごく簡単に紹介する。現在は OSPF は企業のネットワークで標準的に採用され、BGP はキャリア(WAN)のネットワークに使われる。しかしながら MPLS による VPN が普及しているため、BGP は VPN を利用する企業にも採用されている。

表 2-3:ルーティング・プロトコルの比較

	RIP Routing Information Protocol	OSPF Open shortest Path First Protocol	BGP Border Gateway Protocol
経路制御プロトコル・アルゴリズム	ベクター・ディスタンス (Bellman-Ford)	リンク・ステート(Dijkstra)	パス・ベクトル(Path Vector)

適用されるシステム	オートノマス・システム内(Interior Gateway)		オートノマス・システム間 (Exterior Gateway)
経路情報の送信	隣接するルータ同士が経路情報を交換し合う。他のルータからもらった情報に自分の情報を加えてこれを別のインターフェイスに渡す。定期的(全経路)	リンクの状態情報をマルチキャスト。30分ごと、またはトポロジに変更があったとき配布(差分のみ)	明示的に定義された隣接ルータとのみ経路情報を交換。変更時(差分のみ)、キープ・アライブで継続性確認
経路決定	最適経路をホップ数で決定する。大規模ネットワークでは収束に時間がかかる。	メトリック・コスト。各ルータは共通のトポロジカルなデータベースを持つことになる。これをもとに各ルータは最短パスツリーを作成してルーティング・テーブルにする。	パス属性をもとに決定
収束速度	低	高	高
注記	歴史的な存在だがまだ使われている		CIDR 対応、またトラフィック管理可能。比較的シンプルなプロトコルだが、設定は複雑

### 2.2.5.1 ベクター・ディスタンス

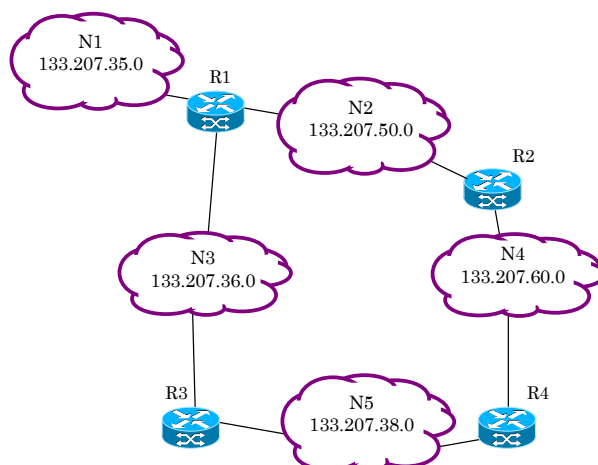
ベクター・ディスタンス(人によってはディスタンス・ベクタと呼ぶ人もいる)というのは、ルータがルーティング情報を更新するアルゴリズムのひとつのクラスを示す。各ルータは、自分に直接繋がっているネットワークやサブネットワークへのルートセット、そして必要とするルートを正しく決定できない場合には付加的な他のホストやネットワークへのルートセットからスタートする。このリストはルーティング・テーブルに記録されており、各リストはあて先ネットワークやホスト、それにそのネットワークへのディスタンス(距離)が示されている。このディスタンスのことを「メトリック(尺度)」と呼び、「ホップ数(hops)」で通常表示する。

各ルータは直接送信できる(他のルータを介さないで)ルータの全てに宛てて、自分が保有しているルーティング・テーブルのコピーを定期的(デフォルトは30秒)に送信する。ルータBがルータAからの報告が届いたら、Bは宛先と各宛先へのディスタンスを調べる。そして、

- Aがその宛先へのもっと短いルートを知っている
- AがBのテーブルに記録していない宛先のリストを持っている
- BからAを介するある宛先のディスタンスが変わった

のいずれかが発見されたときは、自分のテーブルを更新する。

図 2-14: ネットワーク例



上図のように 5 つのクラス B のネットワークが 4 つのルータで接続されていたとしよう。このときの R1 のルーティング・テーブルは次のようになる。

表 2-4: ルーティング・テーブル例

宛先ネットワーク番号 Destination NW	次ルータ First Hop	ホップ数 Metric
133.207.35.0		1
133.207.36.0		1
133.207.50.0		1
133.207.60.0	R2	2
133.207.38.0	R3	2

次ルータ、たとえば R2 からの定期的な報告がこなくなる (RIP は 180 秒と規定) と R1 は R2 に障害が生じたと判断して R2 経由のネットワークへのホップ数を無限大とする。(そのうち R3 からネットワーク 133.207.60.0 への経路を含む報告が来るので、このネットワーク宛ての経路を R3 経由、ホップ数を 3 と書き換える。自らの送出を 1 メトリックとするのは、日本人には馴染めないかもしれない。

この種のアプローチは実装が簡単ではあるものの、欠点も多い。まず定期的なルータ間の通知では、情報の共有に時間がかかり、早くネットワークが変化する (新たな接続が生じたり消滅したりする) 場合対応できない。したがって間違っただルータリングをしてしまう可能性が出る。もうひとつの欠点は、各ルータが隣接する全てのルータに自分のルーティング・テーブル全部のコピーを送らねばならないことである。これは、ネットワークに負担をかけることにもなる。このルーティング・アルゴリズムはメトリックだけを基に最適ルートを求めるので、パスやネットワークの帯域が配慮されない。ホップ数が多くても広帯域のルートを選択したほうが良いことが多い。

RIP には RIP-1 と RIP-2 の二つのバージョンが存在する。RIP はもともと Xerox 社の PUP 及び XNS というプロトコルであり、これが Berkeley BSD UNIX に実装された為、多くの UNIX 機器に標準的に実装されている。RIP-2 は RIP-1 との互換性を維持しつつ CIDR にも対応している。

#### 2.2.5.2 リンク・ステート、最短パス最初 (Link-State, Shortest Path First)

ネットワークがどんどん拡大するにつれ、ベクター・ディスタンスのアルゴリズムでは限界に達し、対応しきれなくなった。その代替として導入されたのがこのアルゴリズムである。このアルゴリズムの特徴は次のようである。

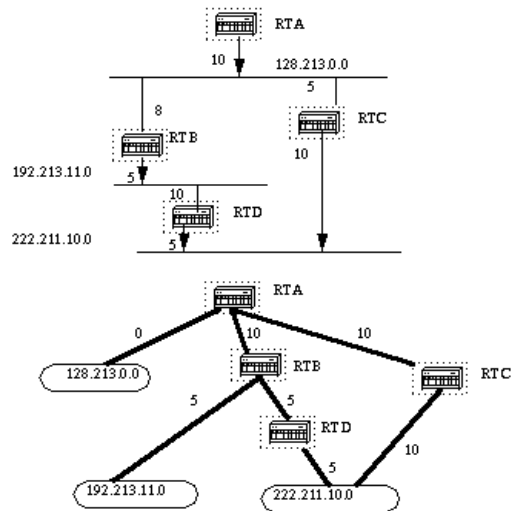
- 物理ネットワークのセットがいくつかの領域(エリア)に分割される
- あるエリアに属する全てのルータは同一のデータベースを所有する(これが仕様書の複雑さの要因となっている)
- 各ルータのデータベースにはどのルータがどのネットワークに接続しているかのルーティング・ドメインの完全なトポロジーが記述されている。あるエリアのトポロジーは、そのエリアの各ルータが持つ全てのリンクを記述したリンク・ステート・データベース(LSDB : Link State Database)で表現されている。従ってこのアルゴリズムは「リンク・ステート・アルゴリズム」と呼ばれる
- 各ルータは、全ての宛先への最適なパスの組をこのデータベースを使って導き出し、これを基にルーティング・テーブルを構築する。最適パスを決定するアルゴリズムのことを最短パス最初(SPF: Shortest Path First)と呼ぶ

一般には、リンク・ステートプロトコルは以下の手順に従う。各ルータは隣接する(ともに同じネットワークに繋がった)ルータに定期的(30秒ごと)にその接続に関する(それが持つリンクの状態(ステート))記述を送信する。この記述のことを「リンク状態広告(LSA: Link State Advertisement)」という。このLSAはこのルータのドメイン全体に行き渡る。同一ドメイン内のルータはこのリンク状態の情報を基に組み立てたデータベースの同一で且つ同期の取れたコピーを保有することになる。このデータベースには、このルータのドメインのトポロジーとこのドメインの外部のネットワークへのルートが記述されている。各ルータはそのトポロジーのデータベースに対してあるアルゴリズムを適用して、最短パスのツリーを作る。このツリーにはこのルータが到達可能な全てのネットワークとルータへの最短のパスが含まれる。この最短パスのツリー、宛先までのコスト、次ホップをもとにルーティング・テーブルが作られる。

ベクター・ディスタンス方式と比較すると、リンク・ステート方式は変更が生じたときだけ、および隣接ルータとの接続が正常であることを確認するために定期的に、LSAを送信する。またルーティング・テーブルの全部の内容を送信するのではなく、一度リンクステート情報が交換されると、この情報に更新がない場合は、基本的にはHelloパケットによる生存確認のみを行う。そして、更新があった場合には、その差分情報だけが交換されるので、ネットワークにかかる負担が少なく、ネットワークの大規模化に対応できる。

広く使われているOSPF(Open Shortest Path First: RFC 1247)はRIPで使われているメトリックではなく、「メトリック・コスト」という基準で経路を選択するアルゴリズムを使っている。コストとは接続されているネットワークの帯域幅、信頼性、通信費などを総合した尺度である。コストはサービス(TOS: Type of Service)によって算出が異なる。一般的には $10^8$  / 回線速度(bps)をコストとしている。

図 2-15: 最短パス・ツリー



上図の上半分はインターフェイスのコスト付きで示したネットワークの例を示している。ルータ A(RTA)にとって最短パスのツリーを作るために、この RTA をツリーのルートとして置き、各宛先への最小コストを算出する。上図の下半分は RTA から見たこのネットワークである。コスト算出のための矢印の方向に注意されたい。例えば、RTB の 128.213.0.0 ネットワークへのインターフェイスのコスト(8)は、192.213.11.0 ネットワークへのコスト算出には関連しない。コスト算出では各ネットワークへの出力インターフェイスのコストが加算されることになる。RTA は RTB を介して 192.213.11.0 に 15 (10+5) のコストで届けることができる。RTA は 222.211.10.0 のネットワークに対し RTC 経由では 20 (10+10) のコストで、あるいは RTB 経由では 20 (10+5+5) のコストで届けることができる。同じ宛先に同じコストのパスが存在するときには、どれを選択するかはメーカーの実装による(例えば負荷バランスや現用予備等)。そのルータが最短パスのツリーを作ったら、それに基づいてルータはルーティング・テーブルを作成する。直結しているネットワークへはコスト 0 で届けられ、他のネットワークへはこのツリーのコスト計算に基づくことになる。

### 2.2.5.3 BGP (Border Gateway Protocol)

BGP(Border Gateway Protocol)はオートノマスな系間(例えばバックボーンやプロバイダ間)の経路設定のプロトコルとして標準的に使用されている。現在は BGP-4 となっており、CIDR 対応になっている。これはバックボーンのルータ内ルーティング・テーブルの膨張を抑制するのに効果があった。

BGP は以下のようなものとして要約されよう:

- AS 間の経路設定情報交信に使用されるプロトコルである
- パス・ベクターのアルゴリズムを採用している
- インクリメンタル(増分)な更新を実施
- コネクション型の TCP(port 番号 179)を採用している
- AS パスのトポロジーに関する情報を交信する

標準的な動作は次のようである:

- 内部または外部の BGP スピーカ(話し手)から複合パスを知る
- 最適なパスを選択し、これを IP 転送テーブルに設定する
- ポリシーは最適パスの選択に影響を与えることで適用される

この動作をもう少し詳細に記すと以下のようである：

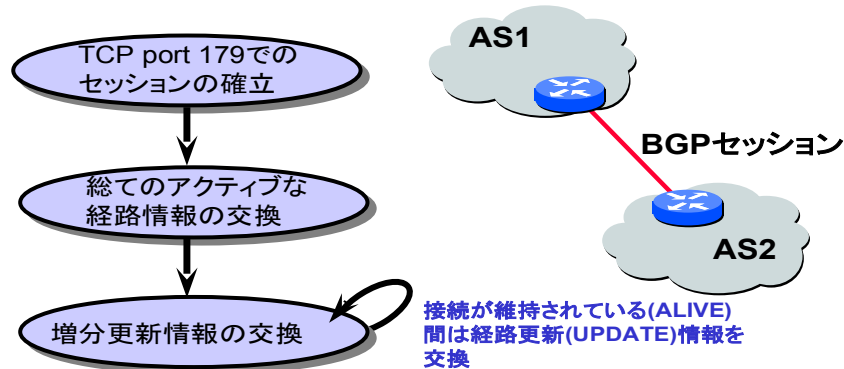
BGP は「パス・ベクターのプロトコル」と考えることができる。BGP バックボーンを跨ぐ最初のデータの交信で完全な BGP ルーティング・テーブルを形成する。ルーティング・テーブルに変更が生じたときは各ルータは更新の増分のみを送信する。BGP はルーティング・テーブル全部を周期的に更新する必要はない。従って BGP 話手(speaker)は自分に対向している総ての相手(次ホップ)との接続が取れている間は、その相手あての総ての現在のバージョンのルーティング・テーブルを保持する。その接続が確実に維持されていることを知るためにキープアライブのメッセージを定期的にその相手に送信する。

BGP のノード同士は信頼度が高い TCP 経由で交信する。Cisco IOS®の BGP 実装ではこれに加えてルーティングの更新に際しては MD5 の認証アルゴリズムも採用している。これにより認定されていない情報源からの誤ったルーティング情報の取り込みを防止している。

外部 AS との情報交信を開始する前に、BGP はその AS 内の各ネットワークが到達可能(reachable)かどうかの確認を行う。このステップはその AS 内のルータとの対向(peering)する内部 BGP との組み合わせによって、及びその AS 内で走っている内部ゲートウェイ・プロトコルにむけて BGP ルーティング情報を再配布することでなされている。

下図は 2 つのルータ間の経路情報の交換プロセスを示している。

図 2-16: BGP における経路情報交換プロセス



交換メッセージは次の 4 種類である：

1. Open(開設) : ピア間のセッションの確立
2. Keep Alive (キープアライブ) : 一定の時間間隔でのハンドシェイク
3. Notification (通報) : ピア間のセッションの終了
4. Update(更新) : 新規経路の通知(Announcing) または以前通知した経路の撤回(withdrawing)。なお通知(announcement) = プレフィックス(prefix) + 属性値(attributes values)

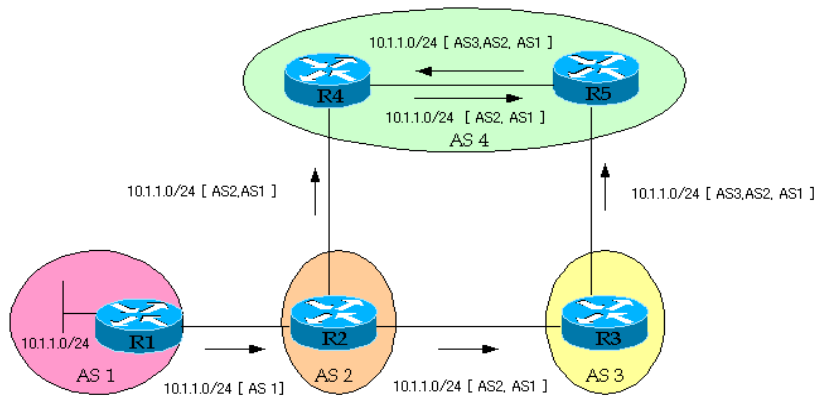
この属性はパス属性とも呼ばれ、以下のものがある：

表 2-5: パス属性

値	コード	意味	参照
1	ORIGIN	経路情報の生成元	[RFC1771]
2	AS_PATH	AS 番号のリストを示す。ルータは、UPDATE	[RFC1771]

		メッセージをアナウンスする際に AS 番号を追加する	
3	NEXT_HOP	トラフィックを転送すべきルータのアドレス	[RFC1771]
4	MULTI_EXIT_DISC	同一の AS に複数の接続を持つようなユーザーが、入力トラフィックを区別するために、優先度を隣接ルータにアナウンス	[RFC1771]
5	LOCAL_PREF	AS からの出力トラフィックの優先度。AS 内にて複数経路を持つような場合、どの経路を優先するかを示す	[RFC1771]
6	ATOMIC_AGGREGATE	経路を集約した際の情報の欠落を示す	[RFC1771]
7	AGGREGATOR	集約経路を生成した AS や BGP ルータ	[RFC1771]
8	COMMUNITY	共通のポリシーを共有する経路のグループ	[RFC1997]
9	ORIGINATOR_ID	これらの属性は通常は知らなくても良い	[RFC2796]
10	CLUSTER_LIST		[RFC2796]
11	DPA		[Chen]
12	ADVERTISER		[RFC1863]
13	RCID_PATH / CLUSTER_ID		[RFC1863]
14	MP_REACH_NLRI		[RFC2283]
15	MP_UNREACH_NLRI		[RFC2283]
16	EXTENDED COMMUNITIES		[Rosen]
...255	reserved for development		

図 2-17:パス属性の伝搬



本資料はネットワークの技術者向けでは無いので、これらの属性と設定法等の詳細は省略する。

経路決定は上記の属性を基になされる。例えば Cisco の場合では以下のような優先順で決定される：

1. 最高の WEIGHT を持ったパス (WEIGHT は Cisco 固有のパラメタ)
2. 最高の LOCAL\_PREF を持ったパス (LOCAL\_PREF はデフォルトでは 100)



3. 最短の AS\_PATH を持ったパス
4. 最小の ORIGIN タイプ値を持ったパス (IGP < EGP < INCOMPLETE)
5. 最低の MULTI\_EXIT\_DISC を持ったパス
6. iBGP パスよりも eBGP パスを優先
7. BGP の次のホップに対し最低の IGP 尺度を持ったパス
8. 最小のルータ ID を持ったパス

## 2.3節 TCP

TCP(トランスミッション制御プロトコル)は IP と組になって高い信頼性を持ったデータ転送をおこなうプロトコルである。TCP/IP では IP がかなりシンプルである分、TCP は重いプロトコルになっている。IP はインターネットのなかを中継されて相手に届けるまでの経路選択と転送で使われるネットワーク層の非接続型(コネクション・レス)のプロトコルで、雑音等で転送中に誤りが生じたデータグラムは廃棄されてしまいまたデータグラム間の順序も保証されないが、TCP は通信しあうピア間で交わすトランスポート層のプロトコルで、これらの問題を解決し実用に耐えるようにしている。

### 2.3.1 TCP のコンセプト

IP はピア間でのデータ・パケットの交信の手段を与えてくれる。しかしながら、ネットワークの状況によってはそのパケットは途中で紛失し未着となる可能性もある。また途中の経路は固定されている訳でもないし、輻輳が生じたりするとパケットごとに到達時間がばらつくこともある。そうすると、送り側でのパケットの順番と受けた時のパケットの順番が異なってしまうこともあり得る。更に送り側は受け側の状況を無視して(というか相手の状況を知るメカニズムを持っていない)パケットを送信するので、受信側のピアが処理できない可能性も出てくる。このような特性は一般のアプリケーションで満足できない。もっと信頼性と順序性が確立され、更にはピア間で複数の種類のデータ交換を同時に行えることが好ましい。例えば、同じピア間で情報検索と IP ベースの音声交話を同時に出来ればもっと便利である。これはピア間で複数の通信のチャンネルを持つイメージである。TCP はこのように IP では一般のアプリケーションでは不十分なところを補うものと考えてよい。

OSI はトランスポート層を次のように定義している:

「トランスポート層は、信頼性あるメッセージの到着を保証し、誤りの検査とフロー制御のメカニズムを持つ。トランスポート層はコネクション・モードとコネクションレス・モードの二つの伝送のモードのサービスを提供する。コネクション・モードの伝送では、他端において完全なメッセージに再組み立てするに必要な形式でのパケットの送受信がなされることになる。」TCP/IP の TCP 層(RFC 793)(と UDP)はまさしく OSI のトランスポート層に対応付けられるものである。この定義で要求されている事項はどのように実現されているのかをまず説明する。

#### 2.3.1.1 信頼性あるメッセージの到着の保証

TCP が上位のアプリケーションと受け渡すデータは、数バイトのデータから、例えば画像のように大変大きなものまで存在し得る。しかしながら下位層の IP では、データグラムあたりの長さに制限があるので、TCP はこれを幾つかのパケットに分割して下位層に渡す。これをセグメントと呼ぶ。受け側で正しい順番でセグメントを並び替える直すことが出来るようシーケンス番号(Sequence Number)が導入された。この番号は送信される

データのオクテット(バイト)位置を示すもの、あるいはデータの総てのオクテットにはシーケンス番号が付与されている、と考えても良い。この番号はある値からスタートし、受かったあるいは送ったデータ分結果として加算される。このシーケンス番号を導入することで、パケットの欠落や順序の乱れに対応できる。詳細は後ほど説明する。

### 2.3.1.2 誤りの検査

誤りの検査は幾つかの手段がある。最初は TCP がパケットとして送り出すセグメント全体のチェックサムでビット誤りなどが検出される。IP 層でも CRC による厳密な誤り検出がすでになされているので、チェックサムで十分という判断であろう。

誤りが発生すれば結果として順序誤りが生じるので、これはシーケンス番号により検出される。受け側は最後に受かったシーケンス番号に 1 を加え(即ち次に期待するシーケンス番号)肯定応答番号(Acknowledge Number)として送信元に送り返す。所定時間内にこの肯定応答が帰ってこなかったら、欠落か誤りが生じたものとして前回のセグメントを再送する。再送のメカニズムも後ほど詳細に説明する。

### 2.3.1.3 フロー制御

フロー制御とは、受け側の処理能力や事情に合わせて送りもとでのパケットの送信を制御するメカニズムである。TCP ではいわゆるウィンドウ制御と呼ばれる方式が採用されている。一般に送信にも受信にもバッファが用意される。受け側は受信バッファに余裕が出たときに次にどれだけ送ってよいかをウィンドウ(Window)というパラメタで送り元に通知する。ネットワークの効率を上げるためにも、この値が小さくならないよう配慮しなければならない。送り元は受けて最大余裕までのサイズのセグメントを送るべきだし、受け側はなるべく大きなウィンドウ値を返すよう配慮すべきである。これものちほど詳細に説明する。

### 2.3.1.4 コネクション・モードとポートの概念

IP は郵便と同じように、パケット単位で相手先にとどけるプロトコルである。しかしながら、電話と同じように相手とある接続を行って、その接続の間相手とそのアプリケーションに関するデータ交換を行うことができれば、相手と複数のアプリケーションを同時に利用できて便利である。電話の場合は実際に回線交換網が相手と回線を接続するが、TCP の場合は IP 網の上で接続を実現しなければならないので、仮想的な接続を行うことになる。そのためにポート(Port)という概念が導入されている。

ポートというのは、相手のホストと複数の相互通信のパイプを作るための接続口のようなものと捉えることが出来る。TCP を実装した各ホストは皆 16 ビットの番号をもった接続口を用意できる。そうすると、二つのホスト間で互いに相手口同士をポート番号を使って接続できることになる。例えばホスト A の 10 番とホスト B の 15 番が、同時にホスト A の 20 番とホスト B の 5 番と、計 2 つのチャンネルが開設される、といった具合である。このときホスト C がホスト B の 15 番に同時に接続してきても問題は生じない。ポート番号が同じであっても相手が違う(IP アドレスが違う)ので切り分けが可能だからである。現に TCP は(自分のポート番号、相手のポート番号、相手の IP アドレス)からなるポート割付管理テーブルをもとにポートの割付を行っている。電話と同じように TCP の接続が確立したら双方は同時に話し合うことが可能である。これを全 2 重方式と呼ぶ。

TCP の一般的なアプリケーションはクライアント/サーバの形式である。読者になじみの WWW サーバは、クライアントのブラウザから最初にアクセスされる。サーバが自発的にクライアントにアクセスをかける事は無い。サーバは受動的、クライアントは能動的に接続を開始する。標準的なアプリケーションではサーバ側のポート

番号が決まっている。これを「よく知られたポート」(Well-known Port)と呼ばれている。これらは国際管理組織の IANA が[定めているので参照されたい](#)。下表はその代表的なものである。

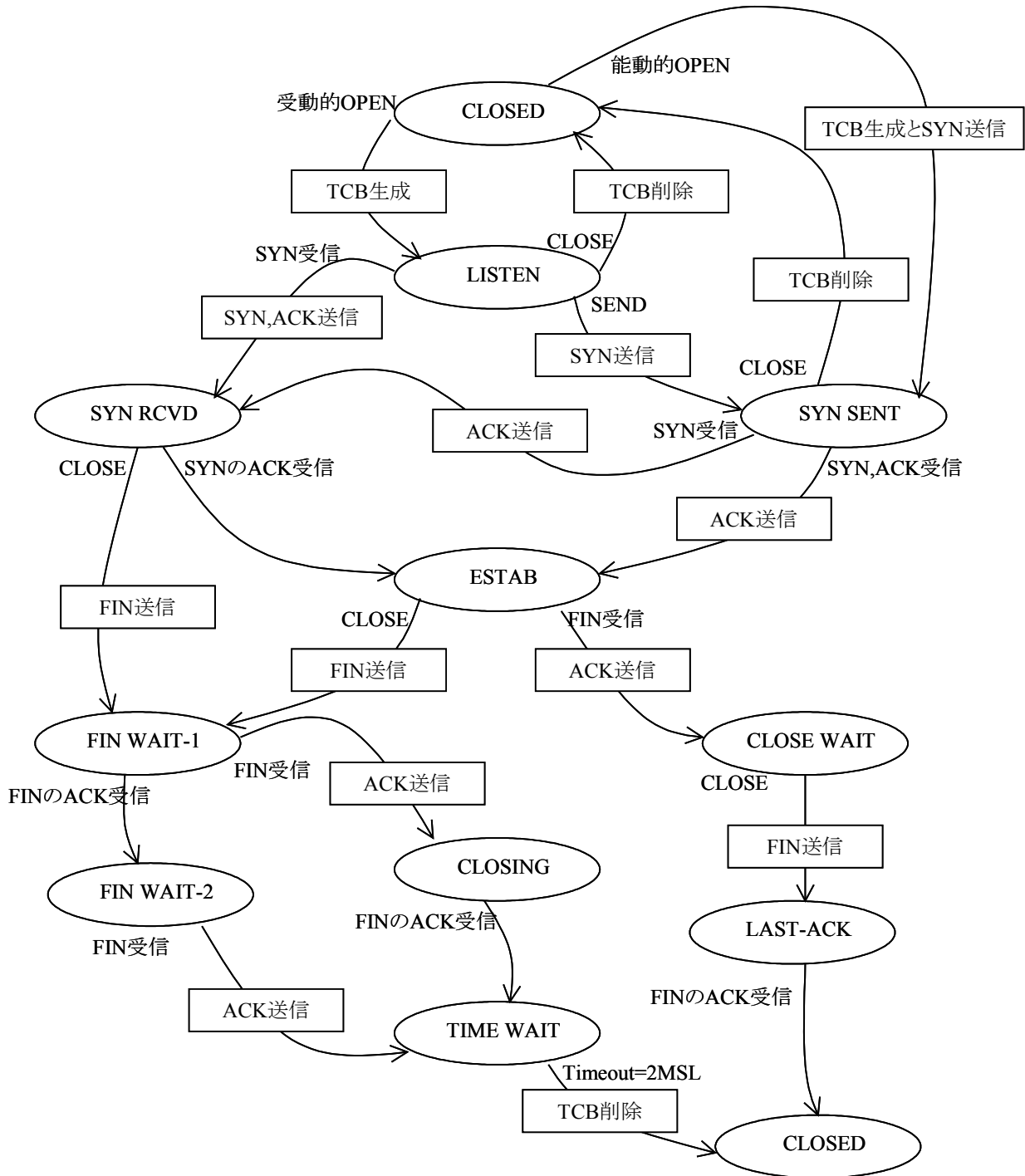
表 2-6: Well-known Port

ポート番号	アプリケーション	ポート番号	アプリケーション
1	TCP Port Service Multiplexer (TCPMUX)	118	SQL Services
5	Remote Job Entry (RJE)	119	Newsgroup (NNTP)
7	ECHO	137	NetBIOS Name Service
18	Message Send Protocol (MSP)	139	NetBIOS Datagram Service
20	FTP -- Data	143	Interim Mail Access Protocol (IMAP)
21	FTP -- Control	150	NetBIOS Session Service
22	SSH Remote Login Protocol	156	SQL Server
23	Telnet	161	SNMP
25	Simple Mail Transfer Protocol (SMTP)	179	Border Gateway Protocol (BGP)
29	MSG ICP	190	Gateway Access Control Protocol (GACP)
37	Time	194	Internet Relay Chat (IRC)
42	Host Name Server (Nameserv)	197	Directory Location Service (DLS)
43	WhoIs	389	Lightweight Directory Access Protocol (LDAP)
49	Login Host Protocol (Login)	396	Novell Netware over IP
53	Domain Name System (DNS)	443	HTTPS
69	Trivial File Transfer Protocol (TFTP)	444	Simple Network Paging Protocol (SNPP)
70	Gopher Services	445	Microsoft-DS
79	Finger	458	Apple QuickTime
80	HTTP	546	DHCP Client
103	X.400 Standard	547	DHCP Server
108	SNA Gateway Access Server	563	SNEWS
119	POP2	569	MSN
110	POP3	1080	Socks
115	Simple File Transfer Protocol (SFTP)		

### 2.3.2 コネクションの状態遷移

TCP のコネクション (接続) は下図のような状態遷移を起こす。以下この図をもとにコネクションの説明をすることにする。

図 2-18: TCP 接続の状態遷移



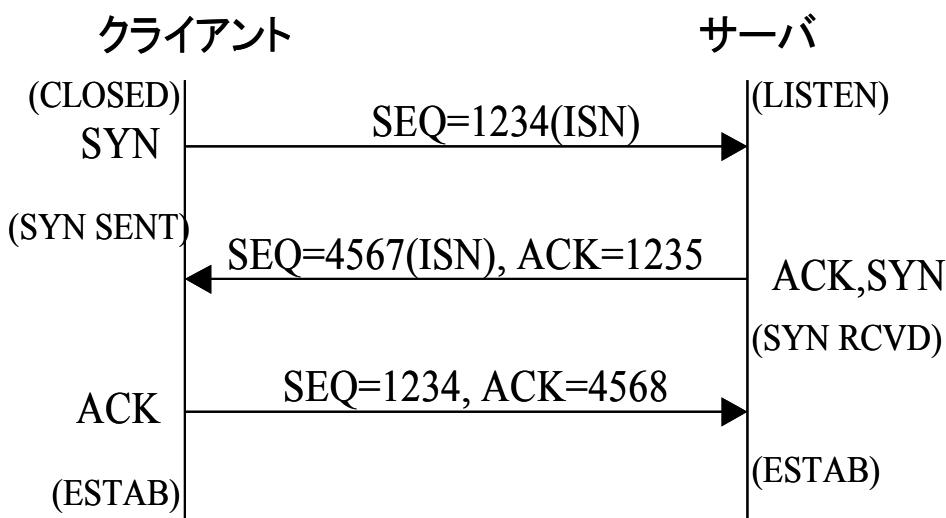
まずこの図の見方を示すと、楕円で示したのが各状態である。状態から状態への遷移を矢印の線で示してある。遷移の際に実行することを長方形で示してある。遷移の契機(イベント)は曲線の付け根の箇所に示してある。CLOSE、OPEN、SEND、RECEIVE などのイベントはユーザから渡されるコマンドと考えればよい。この

図ではまた TCB (Transmission Control Block)と呼ばれる管理テーブルが接続ごとに作られることを想定している。

### 2.3.2.1 コネクションの開設

コネクションの開設は下図のようになる。最初にクライアントから SYN (開設要求)をサーバに送る。サーバはその開設要求に対する肯定応答(ACK,SYN)を返す。クライアントはそれを受信したらその SYN に対する ACK を返す。これでコネクションが開設された (ESTAB) ことになる。状態遷移図でこれを追っていただくと理解が早い。SEQ 番号と ACK 番号には自分がこれから送信するデータの初期番号と、相手に送信を期待するデータの最初のシーケンス番号を指定することができる。つまり双方でデータの送受が可能 (全 2 重通信) ないようにこのような手順を踏む。状態遷移図には受動的 OPEN を始めたのにユーザが SEND を指示してきた時などの状態遷移も示されていて、総ての状態と契機でおかしなことになるよう注意されている。

図 2-19:コネクションの開設

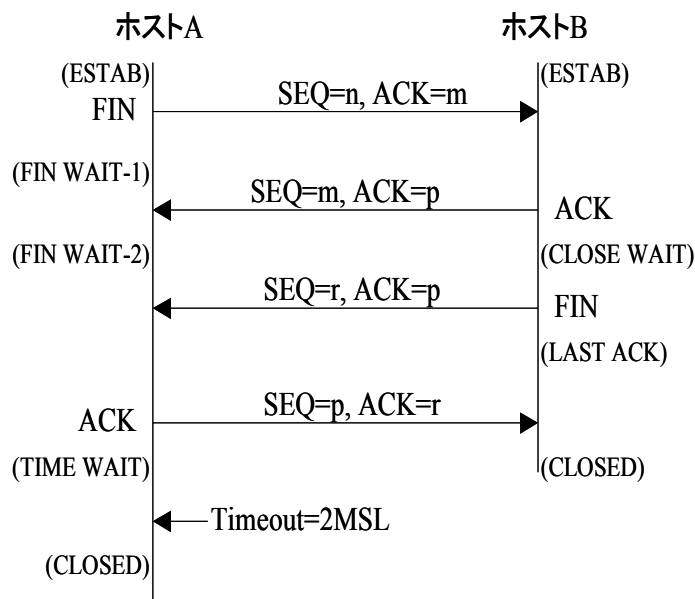


この図ではコネクションが確立された状態では、クライアントは 1235、サーバは 4568 からのデータの送信が可能になる。

### 2.3.2.2 コネクションの開放

コネクションの開放はどちら側から開始しても構わない。双方が同意のもとに開放が行われるように、つまり双方から FIN を送りそれに対して二つとも ACK が返されて開放が終了する。一方がユーザの指示で開放を要求しても他方がまだ送信したいデータがあれば、これを送信し終わってから FIN を送ればよい。FIN を送った後のデータの送信は許されない (ユーザからの送信データは受け付けない)。このような開放手続きは「緩やかな開放 (Graceful Close)」と呼ばれる。

図 2-20:コネクションの開放



この図のように双方が送りたいデータ送信終了してFINを相手に送り、かつそのFINのACK(確認)がとれたところでこのコネクションは開放される。ホストBはホストAからのFINを受理しCLOSE WAITの状態に移っても、まだ送信したいデータがあればこの状態でデータの送信を継続する。ホストBはこれ以上送るデータが無く、かつアプリケーションが指示することでFINの送信を開始する。状態遷移図は双方が同時に開放を開始した時なども対応できるようになっている。ホストAは相手のFINに対するACKを返したら直ちにCLOSEDに移行はしないで一定時間TCBを保持するが、これは返したACKが相手に万一届かないと、ホストBはタイムアウトでまたFINを送ってくることになり、これに対応できるようにしばらくこの接続を存続させるためである。MSLはインターネット上での最大セグメント生存時間(Maximum Segment Lifetime)で通常2分間に設定されている。MSLはインターネット上での最大セグメント生存時間(Maximum Segment Lifetime)で通常2分間に設定されている。ただしこれは現在のインターネットの環境ではかなり長い時間であり、実装しているOSによってもっと短時間に設定されていることもある。

### 2.3.2.3 コネクションのリセット

どちらかのホストでどうしてもプロトコルでは回復できない誤りが検出されたときは、RST(リセット)フラグ付のセグメントを相手に送る。相手はSYN-SENT及びSYN-RECEIVED状態以外ではそのコネクションを放棄しCLOSED状態に戻ることで強制終了を行う。

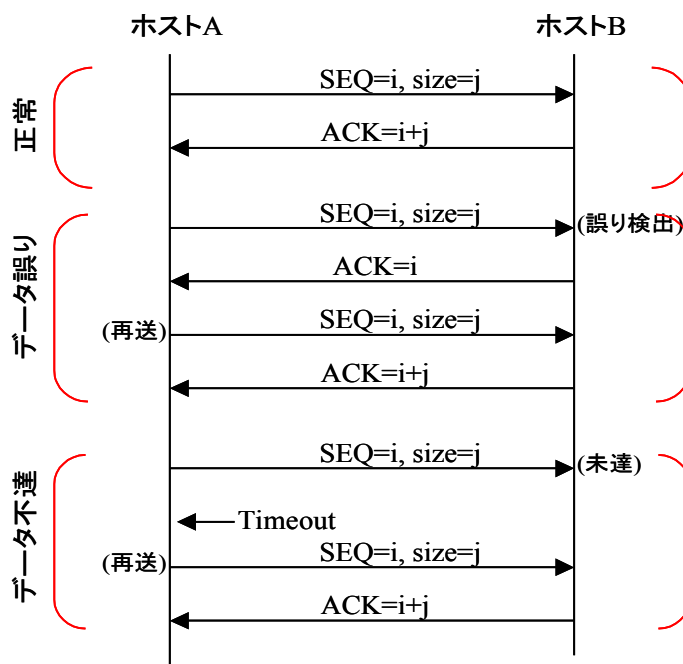
### 2.3.3 データ転送

TCPが信頼あるメッセージの到着を保証するために「再送」のメカニズムを備えていることを既に紹介した。送信元のホストは所定時間内に送ったセグメントに対する肯定応答が帰ってこなかったらそのセグメントを再送する。TCPでは否定応答(NAK)が使われていないことに注意されたい。その代り再送すべきACK番号でこれを送信元に通知する。IP層では誤ったデータグラムを廃棄してしまうので、データ誤りが検出されることは少なく、タイムアウトによる再送が主体となるであろう。これはTCP/IPのひとつの弱点でもある。従って、再送

のタイムアウトはネットワーク環境に対応して、アダプティブかつダイナミックに設定されるようになっていることが好ましい。

下図に正常なシーケンスとデータ誤りや欠落が生じたときのシーケンスを示す。この図ではあるひとつのセグメントのみに着目して示されているが、データを送信する側はここに送ったセグメントの応答を待って次のセグメントを送信する必要は無く、受けてのバッファに余裕がある限りどんどん続けてセグメントを送信しても構わない。

図 2-21: データ転送のシーケンス



再送タイマーはあまり短く設定すると、ネットワークの輻輳などの原因で重複したセグメントが相手に届く確率が大きくなる。重複したセグメントを受信したホストは単純にそのセグメントを破棄すればよいが、ネットワークのトラフィックを圧迫するので注意しなければならない。

再送タイマー値の決定法の例として、仕様書にはラウンドトリップ・タイム (RTT: Round Trip Time) が示されている。これはあるシーケンス番号のセグメントを送信してから、それに対する応答が受信されるまでの時間を計測するものである。これを RTT とすると、これをスムージングした SRTT を下式に基づいて算出する (左辺の SRTT は更新された SRTT であることに注意)。

$$SRTT = (\text{ALPHA} * SRTT) + ((1-\text{ALPHA}) * RTT)$$

しかる後に再送タイマー RTO を下式に基づいて算出する。

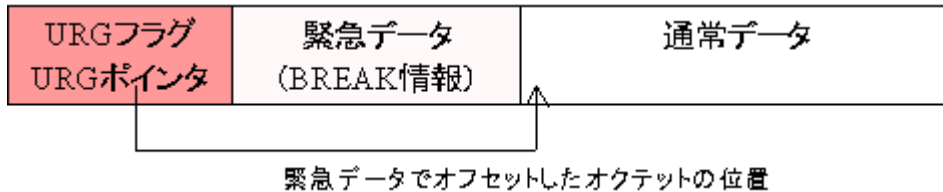
$$RTO = \min[\text{UBOUND}, \max[\text{LBOUND}, (\text{BETA} * SRTT)]]$$

UBOUND は上限 (例えば 1 分)、LBOUND は下限 (例えば 1 秒)、ALPHA はスムージング計数で例えば 0.8~0.9、BETA は遅延バリエーション変数 (例えば 1.3~2.0) である。但しこのようなアルゴリズムが総てのネットワーク環境でうまく機能するかどうかは議論があろう。

### 2.3.4 緊急データ送信

相手からのデータを受信中にもう止めて欲しい(BREAK)とか、それを切り上げて次の処理に移って欲しいと  
かを通知したい場合が存在しよう。URG (緊急ポイントフィールド識別) フラグとポインタはそのような事態に対  
応できるよう用意されている。URG フラグをたて、かつ URG ポインタをセットして相手に送信することで、その  
セグメントに緊急データが含まれていることを通知する。受けたほうはこれをユーザに受信緊急データとともに  
直ちに通知する。

図 2-22: 緊急通知



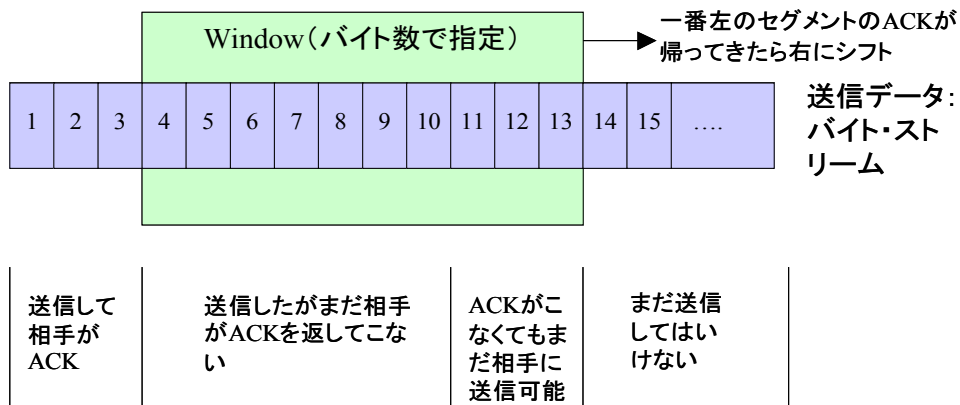
### 2.3.5 ウィンドウ(フロー)制御

これまで説明してきたメカニズムだけではデータ転送速度はネットワークの帯域(と誤り率や遅延)で制限され  
ており、受けての状況が念頭におかれていない。しかしながら受けての処理能力やバッファ用メモリ領域など  
の理由で、送信量を制限する必要もでてくる。ブロードバンドが普及するとますますこれが問題となる。セグメ  
ント・ヘッダにある Window というフィールドはそのため(これをフロー、つまり流量制御という)に用意されてい  
る。基本的に受けてはこれからどれだけまでのデータが受付可能かをこのフィールドにセットすると、送信側は  
それ以上のデータを相手が ACK を返す前に送信することを控える。

下図を見ていただきたい。

図 2-23: フロー制御

TCP のウィンドウによるフロー制御はオクテット(バイト)単位でなされる。一方転送はセグメント単位でなされる。  
従って TCP のフロー制御は一寸ややこしくなっている。TCP はセグメントを作るときは Window も配慮しなけ  
ればならない。



受け側のホストは ACK とともに Window をセットする。各 ACK メッセージにウィンドウ・サイズをセットして構わ  
ない。上図では、Window で表示された枠はこの受けてからの要求でこの接続に適用されているウイン  
ドウを示す。送り手はこのウィンドウ内のデータは ACK が帰ってくる前に相手に送信して構わない。このウイ



ンドウの左端の送信済みデータが相手から ACK されるとこのウィンドウを右にシフトさせる。なお、Window 値はなるべく大きな値のほうが効率が上がる。

受け側は Window 値を決めたら、それに対応したサイズの受信バッファを用意する。これを RWIN と呼ぶ。ブロードバンドの場合のインターネットのダウンロード速度改善のためには、この RWIN の値と PPP に影響を与える IP の MTU の値が重要となる。

### 2.3.6 UDP

UDP も IP 上のプロトコルではあるが、TCP と違ってコネクション・レスのシンプルなプロトコルである。これは TCP のような高信頼のサービスを提供するものではないが、非常に軽いのでオーバーヘッドが少なく、アプリケーションが直接 IP の機能にアクセスしたり、TCP には無い機能（例えばマルチキャストやブロードキャスト）を利用するのに使われる。またテレメトリのようなシステムにも有用であろう。またコネクションの接続・解放を要しないので、比較的短いメッセージ（即ち 1 メッセージ = 1 データグラム）を短時間で送信したいときには効率的である。UDP が使われているものとしては、マルチメディア通信で使われる SIP が良く知られる。

図 2-24: UDP データグラムの構造

0	...	3	4	...	7	8	9	10	...	15	16	...	17	18	...	20	21	...	23	24	...	27	28	...	31
送り元ポート番号(Source Port)												宛先ポート番号(Destination Port)													
長さ(Length)												チェック・サム(Checksum)													
データ(Data ...)																									

上図は UDP データグラムの構造であり、左から右、上から下への順で送信される。

- 送り元ポート番号: TCP と同じく 16 ビットであるが、このフィールドはオプションで、使わないときは 0 とする。UDP では送信側の情報は必要ならばデータ部分に含める。従って UDP は基本的には返信をしない一方方向性の通信だと言える。
- 宛先ポート番号: これも TCP と同じく 16 ビットで、「よく知られたポート」(Well-known Port)が使われることが多い。例えば TFTP(69)、Echo(7)、DNS(53)、DHCP(67,68)、SIP(5060)などである。
- 長さ: 16 ビット長で、ユーザ・データグラムの全長をオクテット単位で示し、ヘッダ部とデータ部を合わせた長さである。最小値はヘッダ部分のみの 8 となる。
- チェック・サム: TCP と同じく IP 層から渡される疑似ヘッダ（プロトコル番号フィールドは 17）と UDP ヘッダとデータ部を合わせて計算される。このフィールドはオプションで、使わないときは 0 がセットされる。またチェック・サムの結果が 0 になったときは 1 が入る。

## 第3章 ソケット・インターフェイス

ほとんどのネットワーク・アプリケーションは TCP / UDP の上に置かれたソケット(Socket)インターフェイスを介してネットワークにアクセスしている。Socket は、TCP と UDP のアプリケーションとのインターフェイスである。Socket の歴史は古く、1980 年代に ARPA (アメリカ国防省の国防高等研究計画局) がカリフォルニア大学の Berkeley 校に資金を出して、TCP/IP プロトコル・スイート(TCP/IP プロトコル・スタックともいう)を UNIX に実装したときに命名された。その意味合いは電気機器のソケットとおなじ概念を与えるため以外のものはない。現在の PC の殆どの OS はこれを実装している。

UDP 通信では接続という概念はなく、従ってユーザは毎回データを送信し、また毎回自分のソケット、及び相手のソケットとアドレスを指定することになる。TCP では最初に相手のソケットとの間の接続を行い、双方のソケット間の接続が確立されたら、互いの通信が可能になる。UDP はデータグラムサイズは 64kB までであるが、TCP ではそのような制限はない。TCP では接続が確立されたら 2 つのソケットはストリームのように振る舞う。TCP では受信したデータの信頼性と順序が保障される。

図 3-1:ソケット・インターフェイスの位置づけ



UDP のためのソケットはデータグラム・ソケット、TCP のためのソケットをストリーム・ソケットと呼ぶことがあるが、これは UDP ではデータグラムのパケット単位で通信されるのにたいし、TCP では一連の TCP パケットで通信の単位が構成されることを区別するために使われている。TCP あるいは UDP 通信をしあう端末のことを端点(end point)と呼ぶ。

### 3.1節 Java ソケット・インターフェイス

Java 言語においてもこのインターフェイスを `java.net.Socket` 及び `java.net.DatagramSocket` というクラスを中心にしてアクセスできるようにしている。Java では UDP を TCP と区別するのに、UDP の User Datagram Protocol という名称から引用した `Datagram` という言葉を使っている。また UDP ではサーバのための `ServerDatagramSocket` というクラスは存在しない。その代わりに `DatagramPacket` というクラスが用意されている。

表 3-1: Java のソケットの型

クラス	概要
Socket	TCP 通信
SocketImpl	TCP 通信実装
SocketImplFactory	TCP 通信実装ファクトリ(インターフェイス)
ServerSocket	サーバの TCP 通信
DatagramSocket	UDP 通信
MulticastSocket	UDP(マルチキャスト)通信
DatagramSocketImpl	UDP 通信実装
DatagramSocketImplFactory	UDP 通信実装ファクトリ(インターフェイス)
DatagramPacket	UDP 通信パケット

これらのクラス(及びインターフェイス)のうち、Socket、ServerSocket、DatagramSocket、及び DatagramPacket だけで通常のアプリケーションを開発できる。

SocketImpl というのは Socket の実装をカスタム化する為のインターフェイスを提供する抽象クラスである。SocketImpl のサブクラスは明示的にインスタンス化されることは意図されていないで、それらは SocketImplFactory によって生成される。SocketImpl と SocketImplFactory という組み合わせが用意されているのは、Socket.setSocketImplFactory() を呼ぶことであるアプリケーション全体で使われているソケットのデフォルトのタイプを変更し、ネットワーキングのコードのすべてを書き換えなくて良いようにしようという為である。カスタム化されたソケットは SSL やファイヤウォールのトンネルなどの機能で必要になる。しかしあるひとつのアプリケーション内での複数のタイプを使うことはできないので、setSocketImplFactory() というメソッドはやや限定的なものになっている。そのような場合は Socket をサブクラス化し、protected Socket(SocketImpl) コンストラクタを使用することになる。

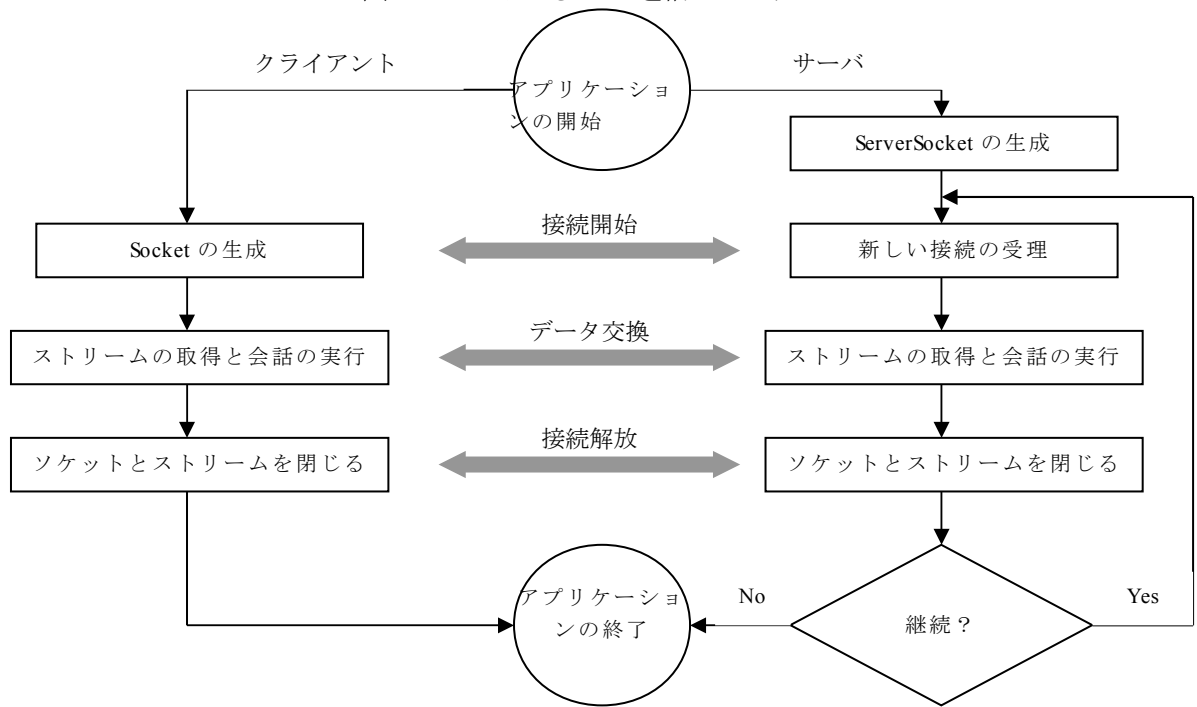
DatagramSocketImpl と DatagramSocketImplFactory も同様であるので説明は省略する。

### 3.2節 Java.net での TCP 通信の概念

TCP 通信では端点は、接続を開始するクライアントとクライアントからの接続を待つ処理を行うサーバの関係になる。Java.net ではその為のクラスとして ServerSocket がサーバ側で、Socket がクライアントとサーバ側用に用意されている。クライアント側では Socket を生成する(あるいは非接続のコンストラクタで生成した Socket のオブジェクトの connect メソッドを呼ぶ)ことで TCP 接続を開始し、ストリームの取得と会話の実行でデータ交換を行い、ソケットとストリームを閉じることで TCP 接続を開放する。サーバ側では ServerSocket を生成することでクライアントからの接続要求を可能にし、接続はクライアントによって開始される。クライアントからの要求待ちは ServerSocket.accept というメソッドであり、要求を受け付け接続が確立すればこのメソッドは Socket 型のオブジェクトを返す。その後でクライアント側と同じく Socket オブジェクト間でのデータの交換がなされる。接続

開放はクライアントと同じであるが、サーバ側では通常他の接続要求にも対応するよう、ループを作って待機することになる。

図 3-2:Java による TCP 通信のシーケンス



Java のソケット通信の API では基本的には通信は `Socket` のインスタンス間で 1 対 1 で実行するが、サーバの為に必要な機能、即ちあるポート番号で多くのクライアントからの接続要求を受け付けるという操作は `ServerSocket` で扱うようにしている。TCP 接続の開始はクライアントからの `Socket.connect` メソッドが行い、TCP 接続の開放は `Socket.close` メソッドが行う。このメソッドはクライアントとサーバの双方から行うことができる。一方の端点が `Socket.close` を行ったときには他の端点のスレッドでは `SocketException` をスローするので、これを知ることができる。

`Socket` オブジェクトはクライアント側では `Socket` のコンストラクタを使ってプログラマ的に生成するが、サーバ側では `ServerSocket.accept` メソッドが、クライアント側からの接続がなされた時点で生成している。つまり `ServerSocket.accept` メソッドはクライアントからの接続待ち受けの為のものである。 `ServerSocket` にも `close` というメソッドがあるが、これは `Socket.close` とは違って、不特定多数のクライアントからの窓口となっている `Socket` を閉じる、つまりそのサービスを終了する為のものであって、ある TCP 接続を終了させる為のものではない。しかし `ServerSocket.close` はそのソケット上で存在している総ての TCP 接続を終了させ、そのスレッドは総て `SocketException` をスローする。

Java では通信データ(バイト・データ)は `Socket.getInputStream` 及び `Socket.getOutputStream` で得られる入力及び出力ストリームを介することになる。これらのストリームをクローズしても TCP 接続が開放される。TCP 接続上へのデータの送信は出力ストリームの内容を吐き出す(`flush`)ことでなされる。TCP 接続上で受信したデータは入力ストリームに蓄積されるが、アプリケーションはこれをバイト単位で読出すことになる。データの終了は TCP プロトコルではないので、双方の取り決め(あるいはアプリケーションのプロトコル)によることになる。データの終了は相手が接続を切ったことでも `SocketException` を受けて知ることができるが、毎回 TCP 接続を切る

のは非常に効率が悪い(HTTP 1.0 ではそうなっている)ので、避けるべきである。TCP の接続と開放は 2.3.2 節で説明したように高度な手続きになっていて、時間と処理のオーバーヘッドが大きい。

### 3.3節 Java.net での UDP 通信の概念

Java での UDP 送信の基本的な手順は以下のようになる:

- DatagramSocket の生成
- InetAddress または InetSocketAddress の生成
- DatagramPacket の生成
- DatagramSocket の send メソッドで DatagramPacket を送信

一方 UDP 受信の基本的な手順は以下のようである:

- DatagramSocket の生成
- DatagramPacket の生成
- DatagramSocket の receive メソッドで DatagramPacket に受信データを詰める

ネットワーク上の端点(end point)間の UDP データグラムの送受信は DatagramSocket.send および DatagramSocket.receive 間でなされる。つまり DatagramSocket は UDP 通信を、DatagramPacket は UDP データグラムのデータを抽象化したものといえる。UDP 通信には接続という概念が存在しないものの、DatagramSocket には close というメソッドが存在する。これは Java が用意した DatagramSocket オブジェクトのクリーンアップのためのメソッドで、もはや通信の必要がなくなったときに使用する。

### 3.4節 Java.net の主なクラスたち

#### 3.4.1 Socket クラス

Java API の Socket クラスは、接続に関してはコンストラクタで TCP 接続をおこなうことと、非接続のオブジェクトを生成してそのオブジェクトの connect メソッドを呼んで接続することの双方が出来るよう用意されている。利便性を考えてのことだろうが、やや分かりづらい。connect メソッドにはタイムアウトを指定できるものがある。なお Socket (及び ServerSocket) には bind という分かりにくいメソッドが存在する。これは特にサーバなどで複数のネットワーク・インターフェイスを実装している場合に自分のホストのどのネットワーク・インターフェイス(ローカル・アドレスとポート)を使うかを指定する為のもので一般用途では使われない。

以下は Javadoc からの転記である:

このクラスは、クライアント・ソケット (単に「ソケット」とも呼ばれる) を実装する。ソケットとは、2 つのマシン間で通信を行う際の端点のことをいう。ソケットの実際の処理は、SocketImpl クラスのインスタンスによって実行される。アプリケーションは、ソケット実装を作成するソケット・ファクトリを変更することで、ローカル・ファイアウォールに適したソケットを作成するように自身を構成することができる。

表 3-2: Socket クラスのコンストラクタとメソッド

コンストラクタ	
<b>public Socket()</b>	システムでデフォルトになっているタイプの <b>SocketImpl</b> を使用して、接続されていないソケットを作成
<b>public Socket(Proxy proxy)</b>	<p>接続されていないソケットを作成。ほかの設定にかかわらず使用すべきプロキシタイプが存在する場合は、そのタイプを指定。セキュリティーマネージャーが存在する場合、その <b>checkAccess</b> メソッドが、プロキシのホストアドレスとポート番号を引数に指定して呼び出される。この結果、<b>SecurityException</b> がスローされることがある。</p> <p>例:</p> <p><b>Socket s = new Socket(Proxy.NO_PROXY);</b> は、ほかのプロキシ構成を無視してプレーンなソケットを作成する。</p> <p><b>Socket s = new Socket(new Proxy(Proxy.Type.SOCKS, new InetSocketAddress("socks.mydom.com", 1080)));</b> は、指定された SOCKS プロキシサーバー経由で接続するソケットを作成する。</p> <p>パラメタ:</p> <p><b>proxy</b> - どのような種類のプロキシ処理を使用すべきかを指定した <b>Proxy</b> オブジェクト</p> <p>例外:</p> <p><b>IllegalArgumentException</b> - プロキシの型が無効な場合、またはプロキシが <b>null</b> の場合</p> <p><b>SecurityException</b> - セキュリティーマネージャーが存在し、プロキシに接続する権限が拒否された場合</p>
<b>protected Socket(SocketImpl impl) throws SocketException</b>	<p>ユーザーが指定した <b>SocketImpl</b> を使用して、接続されていないソケットを作成。</p> <p>パラメタ:</p> <p><b>impl</b> - サブクラスが <b>Socket</b> 上で使用する <b>SocketImpl</b> のインスタンス</p> <p>例外:</p> <p><b>SocketException</b> - 使用しているプロトコルでエラー (TCP エラーなど) が発生した場合</p>
<b>public Socket(String host, int port) throws UnknownHostException, IOException</b>	<p>ストリームソケットを作成し、指定されたホスト上の指定されたポート番号に接続。指定されたホストが <b>null</b> の場合、それは、アドレスを <b>InetAddress.getByName(null)</b> と指定するのと等価になる。つまり、それは、ループバックインタフェースのアドレスを指定するのと等価になる。</p> <p>アプリケーションでサーバーソケットファクトリを指定している場合は、そのファクトリの <b>createSocketImpl</b> メソッドが呼び出され、実際のソケットが作成される。そうでない場合は「プレーンな」ソケットが作成される。</p> <p>セキュリティーマネージャーが存在する場合、その <b>checkAccess</b> メソッドが、ホストアドレスと <b>port</b> を引数に指定して呼び出される。この結果、<b>SecurityException</b> がスローされることがある。</p> <p>パラメタ:</p> <p><b>host</b> - ホスト名。ループバックアドレスの場合は <b>null</b></p> <p><b>port</b> - ポート番号</p> <p>例外:</p> <p><b>UnknownHostException</b> - ホストの IP アドレスを決定できなかった場合</p> <p><b>IOException</b> - ソケットの生成中に入出力エラーが発生した場合</p> <p><b>SecurityException</b> - セキュリティーマネージャーが存在し、その <b>checkConnect</b> メソッドがこの操作を許可しない場合</p>
<b>public Socket(InetAddress address, int port) throws IOException</b>	<p>ストリームソケットを作成し、指定された IP アドレスの指定されたポート番号に接続する。アプリケーションでソケットファクトリを指定している場合は、そのファクトリの <b>createSocketImpl</b> メソッドが呼び出され、実際のソケットが作成される。そうでない場合は「プレーンな」ソケットが作成される。</p> <p>セキュリティーマネージャーが存在する場合、その <b>checkAccess</b> メソッドが、ホストアドレスと <b>port</b> を引数に指定して呼び出される。この結果、<b>SecurityException</b> がスローされることがある。</p> <p>パラメタ:</p>

	<p>address - IP アドレス  port - ポート番号  例外:  IOException - ソケットの生成中に入出力エラーが発生した場合  SecurityException - セキュリティーマネージャーが存在し、その checkConnect メソッドがこの操作を許可しない場合</p>
<p>public Socket(String host,  int port,  InetAddress localAddr,  int localPort)  throws IOException</p>	<p>ソケットを作成し、指定されたリモートポート上の指定されたリモートホストに接続する。さらに、このソケットは、指定されたローカルアドレスとローカルポートにバインドされる。指定されたホストが null の場合、それは、アドレスを InetAddress.getByName(null) と指定するのと等価になる。つまり、それは、ループバックインタフェースのアドレスを指定するのと等価になる。  セキュリティマネージャーが存在する場合、その checkAccess メソッドが、ホストアドレスと port を引数に指定して呼び出される。この結果、SecurityException がスローされることがある。  パラメタ:  host - リモートホストの名前。ループバックアドレスの場合は null  port - リモートポート  localAddr - ソケットのバインド先のローカルアドレス  localPort - ソケットのバインド先のローカルポート  例外:  IOException - ソケットの生成中に入出力エラーが発生した場合  SecurityException - セキュリティーマネージャーが存在し、その checkConnect メソッドがこの操作を許可しない場合</p>
<p>public Socket(InetAddress address,  int port,  InetAddress localAddr,  int localPort)  throws IOException</p>	<p>ソケットを作成し、指定されたリモートポート上の指定されたリモートアドレスに接続する。さらに、このソケットは、指定されたローカルアドレスとローカルポートにバインドされる。セキュリティマネージャーが存在する場合、その checkAccess メソッドが、ホストアドレスと port を引数に指定して呼び出される。この結果、SecurityException がスローされることがある。  パラメタ:  address - リモートアドレス  port - リモートポート  localAddr - ソケットのバインド先のローカルアドレス  localPort - ソケットのバインド先のローカルポート  例外:  IOException - ソケットの生成中に入出力エラーが発生した場合  SecurityException - セキュリティーマネージャーが存在し、その checkConnect メソッドがこの操作を許可しない場合</p>
メソッド	
<p>public void connect(SocketAddress endpoint)  throws IOException</p>	<p>このソケットをサーバーに接続。  パラメタ:  endpoint - SocketAddress  例外:  IOException - 接続時にエラーが発生した場合  IllegalBlockingModeException - このソケットに関連するチャネルが存在し、そのチャネルが非ブロックモードである場合  IllegalArgumentException - 端点が null であるか、このソケットによってサポートされていない SocketAddress サブクラスである場合</p>
<p>public void connect(SocketAddress endpoint,  int timeout)  throws IOException</p>	<p>指定されたタイムアウト値を使って、このソケットをサーバーに接続する。タイムアウト 0 は無限のタイムアウトとして解釈される。その後、接続が確立されるかエラーが発生するまで、接続がブロックされる。  パラメタ:  endpoint - SocketAddress</p>

	<p>timeout - 使用するタイムアウト値 (ミリ秒)</p> <p>例外:</p> <p>IOException - 接続時にエラーが発生した場合</p> <p>SocketTimeoutException - 接続する前にタイムアウトが過ぎた場合</p> <p>IllegalBlockingModeException - このソケットに関連するチャンネルが存在し、そのチャンネルが非ブロックモードである場合</p> <p>IllegalArgumentException - 端点が null であるか、このソケットによってサポートされていない SocketAddress サブクラスである場合</p>
<p>public void <b>bind</b>(SocketAddress bindpoint)</p> <p>throws IOException</p>	<p>ソケットをローカルアドレスにバインドする。アドレスが null の場合は、システムにより一時的なポートと有効なローカルアドレスが選択されてソケットがバインドされる。</p> <p>パラメタ:</p> <p>bindpoint - バインド先の SocketAddress</p> <p>例外:</p> <p>IOException - バインド操作に失敗した場合、あるいはソケットがすでにバインドされている場合</p> <p>IllegalArgumentException - bindpoint が、このソケットによってサポートされていない SocketAddress サブクラスである場合</p>
<p>public InetAddress <b>getInetAddress</b>()</p>	<p>ソケットの接続先のアドレスを返す。</p> <p>戻り値:</p> <p>このソケットの接続先のリモート IP アドレス。ソケットが接続されていない場合は null</p>
<p>public InetAddress <b>getLocalAddress</b>()</p>	<p>ソケットのバインド先のローカルアドレスを取得する。</p> <p>戻り値:</p> <p>ソケットのバインド先のローカルアドレス。ソケットがまだバインドされていない場合は InetAddress.anyLocalAddress()</p>
<p>public int <b>getPort</b>()</p>	<p>このソケットの接続先のリモートポートを返す。</p> <p>戻り値:</p> <p>このソケットの接続先のリモートポート番号。ソケットがまだ接続されていない場合は 0</p>
<p>public int <b>getLocalPort</b>()</p>	<p>このソケットのバインド先のローカルポートを返す。</p> <p>戻り値:</p> <p>このソケットのバインド先のローカルポート番号。ソケットがまだバインドされていない場合は -1</p>
<p>public SocketAddress <b>getRemoteSocketAddress</b>()</p>	<p>このソケットが接続されている端点のアドレスを返す。ソケットが接続されていない場合は null を返す。</p> <p>戻り値:</p> <p>このソケットのリモート端点を表す SocketAddress。ソケットがまだ接続されていない場合は null</p>
<p>public SocketAddress <b>getLocalSocketAddress</b>()</p>	<p>このソケットがバインドされている端点のアドレスを返す。ソケットがバインドされていない場合は null を返す。</p> <p>戻り値:</p> <p>このソケットのローカル端点を表す SocketAddress。ソケットがまだバインドされていない場合は null</p>
<p>public SocketChannel <b>getChannel</b>()</p>	<p>このソケットに関連する固有の SocketChannel オブジェクトを返す (存在する場合)。</p> <p>チャンネル自体が SocketChannel.open または ServerSocketChannel.accept メソッドを使用して作成された場合にだけ、ソケットにチャンネルが存在する。</p> <p>戻り値:</p> <p>このソケットに関連付けられたソケットチャンネル。このソケットがチャンネル用に作成されたものでない場合は null</p>
<p>public InputStream <b>getInputStream</b>()</p> <p>throws IOException</p>	<p>このソケットの入力ストリームを返す。このソケットにチャンネルが関連付けられている場合、結果として得られる入力ストリームは、その操作のすべてをチャンネルに委譲する。そのチャンネルが非ブロックモードである場合、入力ストリームの read 操作が</p>



	<p><code>IllegalBlockingModeException</code> をスローする。</p> <p>異常な状況下では、リモートホストやネットワークソフトウェアによって使用している接続が解除される可能性がある (TCP 接続の場合であれば接続がリセットされるなど)。接続の解除がネットワークソフトウェアによって検出された場合、返された入力ストリームに対して次のことが当てはまる:</p> <ul style="list-style-type: none"> <li>ネットワークソフトウェアがソケットによってバッファリングされたバイトを破棄する可能性がある。ネットワークソフトウェアによって破棄されていないバイトは、<code>read</code> を使って読み取ることができる。</li> <li>ソケット上にバッファリングされたバイトが 1 つも存在しないか、バッファリングされたすべてのバイトが <code>read</code> によって消費されてしまった場合、後続の <code>read</code> 呼び出しはすべて、<code>IOException</code> をスローする。</li> <li>ソケット上にバッファリングされたバイトが 1 つも存在せず、<code>close</code> を使ってソケットがクローズされた場合、<code>available</code> が 0 を返す。</li> </ul> <p>返された <code>InputStream</code> をクローズすると、関連付けられたソケットがクローズする。</p> <p>戻り値: このソケットからバイトを読み込むための入力ストリーム</p> <p>例外: <code>IOException</code> - 入力ストリームの作成時に入出力エラーが発生した場合、ソケットがクローズされている場合、ソケットが接続されていない場合、または <code>shutdownInput()</code> を使ってソケットの入力がシャットダウンされた場合</p>
<p><code>public OutputStream getOutputStream() throws IOException</code></p>	<p>このソケットの出力ストリームを返す。このソケットにチャンネルが関連付けられている場合、結果として得られる出力ストリームは、その操作のすべてをチャンネルに委譲する。チャンネルが非ブロックモードである場合、出力ストリームの <code>write</code> 操作が <code>IllegalBlockingModeException</code> をスローする。</p> <p>返された <code>OutputStream</code> をクローズすると、関連付けられたソケットがクローズする。</p> <p>戻り値: このソケットにバイトを書き込むための出力ストリーム</p> <p>例外: <code>IOException</code> - 出力ストリームの作成中に入出力エラーが発生した場合、またはソケットが接続されていない場合</p>
<p><code>public void setTcpNoDelay(boolean on) throws SocketException</code></p>	<p><code>TCP_NODELAY</code> を有効または無効にする。</p> <p>パラメタ: <code>on</code> - <code>TCP_NODELAY</code> を有効にする場合は <code>true</code>、無効にする場合は <code>false</code></p> <p>例外: <code>SocketException</code> - 使用しているプロトコルでエラー (TCP エラーなど) が発生した場合</p>
<p><code>public boolean getTcpNoDelay() throws SocketException</code></p>	<p><code>TCP_NODELAY</code> が有効かどうかを調べる。</p> <p>戻り値: <code>TCP_NODELAY</code> が有効かどうかを示す <code>boolean</code> 値</p> <p>例外: <code>SocketException</code> - 使用しているプロトコルでエラー (TCP エラーなど) が発生した場合</p>
<p><code>public void setSoLinger(boolean on, int linger) throws SocketException</code></p>	<p>指定された遅延時間 (秒) を使って <code>SO_LINGER</code> を有効または無効にする。タイムアウトの最大値はプラットフォームに固有である。設定はソケットを閉じる場合にだけ影響する。</p> <p>パラメタ: <code>on</code> - 遅延時間を有効にするかどうかを指定 <code>linger</code> - <code>on</code> が <code>true</code> の場合は、遅延時間</p> <p>例外: <code>SocketException</code> - 使用しているプロトコルでエラー (TCP エラーなど) が発生した場合 <code>IllegalArgumentException</code> - 遅延時間の値が負の数値である場合</p>
<p><code>public void setSoLinger(boolean on, int linger)</code></p>	<p>指定された遅延時間 (秒) を使って <code>SO_LINGER</code> を有効または無効にする。タイムアウトの最大値はプラットフォームに固有である。設定はソケットを閉じる場合にだけ影響する。</p>

throws <code>SocketException</code>	<p>パラメタ:</p> <p><code>on</code> - 遅延時間を有効にするかどうかを指定</p> <p><code>linger</code> - <code>on</code> が <code>true</code> の場合は、遅延時間</p> <p>例外:</p> <p><code>SocketException</code> - 使用しているプロトコルでエラー (TCP エラーなど) が発生した場合</p> <p><code>IllegalArgumentException</code> - 遅延時間の値が負の数値である場合</p>
<p>public int <b>getSoLinger()</b></p> <p>throws <code>SocketException</code></p>	<p><code>SO_LINGER</code> の設定を返す。戻り値 -1 は、このオプションが無効になっていることを意味する。設定はソケットを閉じる場合にだけ影響する。</p> <p>戻り値:</p> <p><code>SO_LINGER</code> の設定</p> <p>例外:</p> <p><code>SocketException</code> - 使用しているプロトコルでエラー (TCP エラーなど) が発生した場合</p>
<p>public void <b>sendUrgentData</b>(int data)</p> <p>throws <code>IOException</code></p>	<p>このソケット上で 1 バイトの緊急データを送信する。送信されるバイトは、データパラメタの最下位の 8 ビットである。この緊急バイトは、ソケットの <code>OutputStream</code> への先行するすべての書き込みの後、<code>OutputStream</code> への後続のすべての書き込みの前に送信される。</p> <p>パラメタ:</p> <p><code>data</code> - 送信するデータのバイト</p> <p>例外:</p> <p><code>IOException</code> - データ送信時にエラーが発生した場合</p>
<p>public void <b>setOOBInline</b>(boolean on)</p> <p>throws <code>SocketException</code></p>	<p><code>OOBINLINE</code> (TCP 緊急データの受信) を有効または無効にする。デフォルトではこのオプションは無効になっており、ソケット上で受信された TCP 緊急データは何の通知もなく破棄される。ユーザーが緊急データの受信を望んでいる場合は、このオプションを有効にすること。有効にした場合、緊急データは通常データとともにインラインで受信される。</p> <p>受信緊急データの処理に関しては、限られたサポートしか提供されていないことに注意されたい。特に、高位レベルのプロトコルが提供されていない場合、受信する緊急データの通知は提供されず、通常データと緊急データを区別する機能はない。</p> <p>パラメタ:</p> <p><code>on</code> - <code>OOBINLINE</code> を有効にする場合は <code>true</code>、無効にする場合は <code>false</code></p> <p>例外:</p> <p><code>SocketException</code> - 使用しているプロトコルでエラー (TCP エラーなど) が発生した場合</p>
<p>public boolean <b>getOOBInline</b>()</p> <p>throws <code>SocketException</code></p>	<p><code>OOBINLINE</code> が有効かどうかを調べる。</p> <p>戻り値:</p> <p><code>OOBINLINE</code> が有効かどうかを示す <code>boolean</code> 値</p> <p>例外:</p> <p><code>SocketException</code> - 使用しているプロトコルでエラー (TCP エラーなど) が発生した場合</p>
<p>public void <b>setSoTimeout</b>(int timeout)</p> <p>throws <code>SocketException</code></p>	<p>指定されたタイムアウト (ミリ秒) を使って <code>SO_TIMEOUT</code> を有効または無効にする。このオプションを 0 以外のタイムアウトに設定すると、この <code>Socket</code> に関連付けられた <code>InputStream</code> の <code>read()</code> 呼び出しが、その時間の間だけブロックされる。タイムアウトの期限が切れると、<code>Socket</code> がまだ有効であっても <code>java.net.SocketTimeoutException</code> が発行される。このオプションは、ブロック処理に入る前に有効にしておく必要がある。タイムアウトは 0 より大きい値を指定する。タイムアウト 0 は無限のタイムアウトとして解釈される。</p> <p>パラメタ:</p> <p><code>timeout</code> - ミリ秒で表される、指定されたタイムアウト</p> <p>例外:</p> <p><code>SocketException</code> - 使用しているプロトコルでエラー (TCP エラーなど) が発生した場合</p>
<p>public int <b>getSoTimeout</b>()</p> <p>throws <code>SocketException</code></p>	<p><code>SO_TIMEOUT</code> の設定を返す。このオプションが無効 (タイムアウトが無限) の場合は 0 を返す。</p> <p>戻り値:</p> <p><code>SO_TIMEOUT</code> の設定</p>

	<p>例外:  <b>SocketException</b> - 使用しているプロトコルでエラー (TCP エラーなど) が発生した場合</p>
<p><b>public void setSendBufferSize(int size)</b>  throws <b>SocketException</b></p>	<p>この <b>Socket</b> の <b>SO_SNDBUF</b> オプションを指定された値に設定する。<b>SO_SNDBUF</b> オプションは、使用するネットワーク入出力バッファに設定するサイズのヒントとして、プラットフォームのネットワークコードが使う。<b>SO_SNDBUF</b> はヒントなので、アプリケーションでバッファのサイズ設定を調べる必要がある場合は、<b>getSendBufferSize()</b> を呼び出すこと。</p> <p>パラメタ:  <b>size</b> - 送信バッファサイズの設定サイズ。この値は 0 より大きくなければならない</p> <p>例外:  <b>SocketException</b> - 使用しているプロトコルでエラー (TCP エラーなど) が発生した場合  <b>IllegalArgumentException</b> - 値が 0 または負の値である場合</p>
<p><b>public int getSendBufferSize()</b>  throws <b>SocketException</b></p>	<p>この <b>Socket</b> で使われる <b>SO_SNDBUF</b> オプションの値を取得する。これは、この <b>Socket</b> で出力用としてプラットフォームが使うバッファのサイズである。</p> <p>戻り値:  この <b>Socket</b> の <b>SO_SNDBUF</b> オプションの値</p> <p>例外:  <b>SocketException</b> - 使用しているプロトコルでエラー (TCP エラーなど) が発生した場合</p>
<p><b>public void setReceiveBufferSize(int size)</b>  throws <b>SocketException</b></p>	<p>この <b>Socket</b> の <b>SO_RCVBUF</b> オプションを指定された値に設定する。<b>SO_RCVBUF</b> オプションは、使用するネットワーク入出力バッファに設定するサイズのヒントとして、プラットフォームのネットワークコードが使う。受信バッファのサイズを増やすと、大規模な接続でのネットワーク入出力のパフォーマンスを上げることができる。一方、サイズを減らすと、受信データのバックログを減らすことができる。<b>SO_RCVBUF</b> はヒントなので、アプリケーションでバッファのサイズ設定を調べる必要がある場合は、<b>getReceiveBufferSize()</b> を呼び出すこと。</p> <p><b>SO_RCVBUF</b> の値は、リモートピアに通知される TCP 受信ウィンドウの設定にも使用される。一般に、ソケットが接続されているかぎり、このウィンドウサイズはいつでも変更できる。ただし、64K を超える受信ウィンドウを要求する場合は、ソケットをリモートピアに接続する前に変更を要求する必要がある。次の 2 つの場合に注意されたい:</p> <ul style="list-style-type: none"> <li>• <b>ServerSocket</b> から受け入れたソケットの場合、<b>ServerSocket</b> をローカルアドレスにバインドする前に、<b>ServerSocket.setReceiveBufferSize(int)</b> を呼び出してこれを実行する必要がある。</li> <li>• クライアントソケットの場合、ソケットをそのリモートピアに接続する前に、<b>setReceiveBufferSize()</b> を呼び出す必要がある。</li> </ul> <p>パラメタ:  <b>size</b> - 受信バッファ・サイズの設定サイズ。この値は 0 より大きくなければならない</p> <p>例外:  <b>IllegalArgumentException</b> - 値が 0 または負の値である場合  <b>SocketException</b> - 使用しているプロトコルでエラー (TCP エラーなど) が発生した場合</p>
<p><b>public int getReceiveBufferSize()</b>  throws <b>SocketException</b></p>	<p>この <b>Socket</b> で使われる <b>SO_RCVBUF</b> オプションの値を取得する。これは、この <b>Socket</b> で入力用としてプラットフォームが使うバッファのサイズである。</p> <p>戻り値:  この <b>Socket</b> の <b>SO_RCVBUF</b> オプションの値</p> <p>例外:  <b>SocketException</b> - 使用しているプロトコルでエラー (TCP エラーなど) が発生した場合</p>
<p><b>public void setKeepAlive(boolean on)</b>  throws <b>SocketException</b></p>	<p><b>SO_KEEPALIVE</b> を有効または無効にする。</p> <p>パラメタ:  <b>on</b> - ソケットをオンのままにしておくかどうかを指定</p> <p>例外:  <b>SocketException</b> - 使用しているプロトコルでエラー (TCP エラーなど) が発生した場合</p>
<p><b>public boolean getKeepAlive()</b>  throws <b>SocketException</b></p>	<p><b>SO_KEEPALIVE</b> が有効かどうかを調べる。</p> <p>戻り値:</p>

	<p>SO_KEEPALIVE が有効かどうかを示す boolean 値 例外: SocketException - 使用しているプロトコルでエラー (TCP エラーなど) が発生した場合</p>
<p>public void <b>setTrafficClass</b>(int tc) throws SocketException</p>	<p>このソケットから送信されるパケットの IP ヘッダーのトラフィッククラスまたはサービスタイプのオクテットを設定する。使用するネットワーク実装がこの値を無視することがあるので、アプリケーションではこの値をヒントと考えられたい。tc の範囲は <math>0 \leq tc \leq 255</math> でなければならない。そうでない場合は、IllegalArgumentExceptio がスローされる。 注: IP (Internet Protocol) バージョン 4 の場合、RFC 1349 で説明されているように、この値は優先度の高い octet と TOS フィールドで構成される。TOS フィールドは、次のようにビット単位の論理和によって作成されるビットセットである。 IPTOS_LOWCOST (0x02) IPTOS_RELIABILITY (0x04) IPTOS_THROUGHPUT (0x08) IPTOS_LOWDELAY (0x10) 最下位ビットは、MBZ (0 でなければならない) ビットに対応するので、常に無視される。優先フィールドにビットを設定すると、操作が許可されないことを示す SocketException になることがある。 RFC 1122 のセクション 4.2.4.2 に示されているように、準拠した TCP 実装は、接続の寿命がある間はアプリケーションが TOS フィールドを変更できるようにすべきである (ただし、必ずしもそうする必要はない)。したがって、TCP 接続の確立後にサービスタイプフィールドを変更できるかどうかは、使用するプラットフォーム内の実装によって決まる。アプリケーションは、接続後に TOS フィールドを変更できると仮定すべきではない。 IP (Internet Protocol) バージョン 6 の場合、tc は IP ヘッダーの sin6_flowinfo フィールドに格納される値である。 パラメタ: tc - ビットセットの int 値 例外: SocketException - トラフィッククラスまたはサービスタイプの設定時にエラーが発生した場合</p>
<p>public int <b>getTrafficClass</b>() throws SocketException</p>	<p>このソケットから送信されるパケットの IP ヘッダーのトラフィッククラスまたはサービスタイプを取得する。使用するネットワーク実装が、setTrafficClass(int) を使用して設定されたトラフィッククラスまたはサービスタイプを無視することがあるので、この Socket で setTrafficClass(int) メソッドを使用して以前に設定された値とは異なる値がこのメソッドから返されることがある。 戻り値: すでに設定されているトラフィッククラスまたはサービス型 例外: SocketException - トラフィッククラスまたはサービスタイプの値を取得する際にエラーが発生した場合</p>
<p>public void <b>setReuseAddress</b>(boolean on) throws SocketException</p>	<p>SO_REUSEADDR ソケットオプションを有効または無効にする。TCP 接続をクローズする場合、接続クローズ後の一定期間、その接続がタイムアウト状態 (通常、TIME_WAIT 状態または 2MSL 待機状態と呼ばれる) にとどまる可能性がある。周知のソケットアドレスまたはポートを使用するアプリケーションの場合、ソケットアドレスまたはポートに関連する接続がタイムアウト状態にあると、ソケットを必要な SocketAddress にバインドできないことがある。 bind(SocketAddress) を使用してソケットをバインドする前に SO_REUSEADDR を有効にすると、以前の接続がタイムアウト状態でもソケットをバインドすることができる。 Socket が作成されると、SO_REUSEADDR の初期設定は無効になる。 ソケットがバインドされた (isBound() を参照) あとで SO_REUSEADDR を有効または無効にする場合の動作は定義されていない。 パラメタ: on - ソケットオプションを有効にするか無効にするかを指定 例外:</p>

	SocketException - SO_REUSEADDR ソケットオプションの有効化または無効化時にエラーが発生した場合、またはソケットがクローズされている場合
public boolean <b>getReuseAddress()</b> throws SocketException	SO_REUSEADDR が有効かどうかを調べる。 戻り値: SO_REUSEADDR が有効かどうかを示す boolean 値
public void <b>close()</b> throws IOException	このソケットを閉じる。現在このソケットの入出力操作でブロックされているすべてのスレッドが SocketException をスローする。 ソケットが閉じられると、その後のネットワークにそのソケットを使用することはできない (つまり、再接続または再バインドはできない)。新しいソケットを作成する必要がある。このソケットをクローズすると、このソケットの InputStream と OutputStream もクローズされる。このソケットに関連するチャンネルが存在する場合は、そのチャンネルも閉じられる。 例外: IOException - このソケットを閉じるときに入出力エラーが発生した場合
public void <b>shutdownInput()</b> throws IOException	このソケットの入力ストリームを「ストリームの終わり」に設定する。ソケットの入力ストリーム側に送信されたデータはすべて、確認されたあと何の通知もなく破棄される。ソケットで shutdownInput() を呼び出したあとにソケットの入力ストリームから読み込むと、ストリームは EOF を返す。 例外: IOException - このソケットを停止するときに入出力エラーが発生した場合
public void <b>shutdownOutput()</b> throws IOException	このソケットの出力ストリームを無効にする。TCP ソケットの場合、それまでに書き込まれたデータのすべてが、TCP の通常の接続終了シーケンスに従って送信される。ソケットで shutdownOutput() を呼び出したあとにソケットの出力ストリームに書き込むと、ストリームは IOException をスローする。 例外: IOException - このソケットを停止するときに入出力エラーが発生した場合
public String <b>toString()</b>	このソケットを String に変換。 オーバーライド: クラス Object 内の toString 戻り値: このソケットの文字列表現
public boolean <b>isConnected()</b>	ソケットの接続状態を返す。 戻り値: ソケットがサーバーに正常に接続されている場合は true
public boolean <b>isBound()</b>	ソケットのバインディング状態を返す。 戻り値: ソケットが正常にアドレスにバインドされている場合は true
public boolean <b>isClosed()</b>	ソケットの閉じた状態を返す。 戻り値: ソケットが閉じた場合は true
public boolean <b>isInputShutdown()</b>	ソケット接続の読み込み側の半分が閉じているかどうかを返す。 戻り値: ソケットの入力が停止した場合は true
public boolean <b>isOutputShutdown()</b>	ソケット接続の書き込み側の半分が閉じているかどうかを返す。 戻り値: ソケットの出力が停止した場合は true
public static void <b>setSocketImplFactory(SocketImplFactory fac)</b>	アプリケーションのクライアントソケット実装ファクトリを設定する。ファクトリを指定できるのは一度だけである。アプリケーションで新しいクライアントソケットを作成すると、ソケット実装ファクトリの createSocketImpl メソッドが呼び出され、実際のソケットが作成される。

throws IOException	<p>このメソッドに null を渡しても、ファクトリがすでに設定されていないかぎり、それは無操作になる。</p> <p>セキュリティーマネージャーが存在する場合、このメソッドは最初にセキュリティーマネージャーの checkSetFactory メソッドを呼び出すことにより、この操作が許可されていることを確認する。この結果、SecurityException がスローされることがある。</p> <p>パラメタ:</p> <p>fac - 目的のファクトリ</p> <p>例外:</p> <p>IOException - ソケットファクトリの設定中に入出力エラーが発生した場合</p> <p>SocketException - ファクトリがすでに定義されている場合</p> <p>SecurityException - セキュリティーマネージャーが存在し、その checkSetFactory メソッドがこの操作を許可しない場合</p>
public void setPerformancePreferences(int connectionTime, int latency, int bandwidth)	<p>このソケットのパフォーマンス設定を行う。ソケットはデフォルトで、TCP/IP プロトコルを使用する。実装によっては、TCP/IP とは異なるパフォーマンス特性を持つ代替プロトコルを提供することもある。アプリケーションは、このメソッドを使用することで、実装で利用可能なプロトコルの選択時に、これらのかね合いの取り方を示す独自の設定を表現できる。</p> <p>パフォーマンス設定は、接続時間の長さ、応答時間の速さ、および帯域幅の広さの相対的な重要度を示す 3 つの整数値によって記述される。これらの整数の絶対値は重要ではない。ある特定のプロトコルを選択するために、これらの値が単純に比較されるが、その際、値が大きければより強い設定を示す。負の値は正の値よりも低い優先順位を表す。たとえば、アプリケーションが応答時間の速さや帯域幅の広さよりも接続時間の短さを優先する場合には、値 (1, 0, 0) を指定してこのメソッドを呼び出すことができる。アプリケーションが応答時間の速さよりも帯域幅の広さを優先し、接続時間の短さよりも待ち時間の短さを優先する場合には、値 (0, 1, 2) を指定してこのメソッドを呼び出すことができる。このソケットの接続後にこのメソッドを呼び出しても、何の効果も生じない。</p> <p>パラメタ:</p> <p>connectionTime - 接続時間の長さの相対的な重要度を表す int</p> <p>latency - 応答時間の速さの相対的な重要度を表す int</p> <p>bandwidth - 帯域幅の広さの相対的な重要度を表す int</p>

### 3.4.2 ServerSocket クラス

以下は Javadoc からの転記である:

このクラスはサーバーソケットを実装している。サーバーソケットは、ネットワーク経由でクライアントからの要求が送られてくるのを待つ。このクラスは、その要求に基づいていくつかの操作を実行し、その後、場合によっては要求元に結果を返す。

サーバーソケットの実際の処理は、SocketImpl クラスのインスタンスによって実行されている。アプリケーションは、ソケット実装を作成するソケットファクトリを変更することで、ローカルファイアウォールに適したソケットを作成するようにアプリケーション自体を構成することができる。

表 3-3: ServerSocket クラスのコンストラクタとメソッド

コンストラクタ	
public ServerSocket() throws IOException	<p>バインドされていないサーバーソケットを作成。</p> <p>例外:</p> <p>IOException - ソケットを開くときの入出力エラー</p>
public ServerSocket(int port) throws IOException	<p>指定されたポートにバインドされたサーバーソケットを作成。ポート 0 を指定すると、空いているポート上でソケットが作成される。受信する接続 (接続要求) のキューの最大長</p>

	<p>は、50 に設定される。キューが埋まっているときに接続要求があると、接続は拒否される。</p> <p>アプリケーションでサーバーソケットファクトリを指定している場合は、そのファクトリの <code>createSocketImpl</code> メソッドが呼び出され、実際のソケットが作成される。そうでない場合は「プレーンな」ソケットが作成される。</p> <p>セキュリティーマネージャーが存在する場合、その <code>checkListen</code> メソッドが <code>port</code> 引数をその引数として指定して呼び出され、この操作が許可されるかどうかを確認される。この結果、<code>SecurityException</code> がスローされることがある。</p> <p>パラメタ:</p> <p><code>port</code> - ポート番号。空いているポートを使用する場合は 0</p> <p>例外:</p> <p><code>IOException</code> - ソケットを開いているときに入出力エラーが発生した場合</p> <p><code>SecurityException</code> - セキュリティーマネージャーが存在し、その <code>checkListen</code> メソッドがこの操作を許可しない場合</p>
<p><code>public ServerSocket(int port, int backlog)</code> throws <code>IOException</code></p>	<p>サーバーソケットを作成し、指定されたローカルポート番号にバインドし、指定されたバックログを設定する。ポート番号 0 を指定すると、空いているポート上でソケットが作成される。</p> <p>受信する接続 (接続要求) のキューの最大長は、<code>backlog</code> パラメタに設定される。キューが埋まっているときに接続要求があると、接続は拒否される。</p> <p>アプリケーションでサーバーソケットファクトリを指定している場合は、そのファクトリの <code>createSocketImpl</code> メソッドが呼び出され、実際のソケットが作成される。そうでない場合は「プレーンな」ソケットが作成される。</p> <p>セキュリティーマネージャーが存在する場合、その <code>checkListen</code> メソッドが <code>port</code> 引数をその引数として指定して呼び出され、この操作が許可されるかどうかを確認される。この結果、<code>SecurityException</code> がスローされることがある。</p> <p><code>backlog</code> 引数は、0 より大きい正の値である必要がある。渡された値が 0 以下の場合は、デフォルトの値が使用される。</p> <p>パラメタ:</p> <p><code>port</code> - 使用するポート。空いているポートを使用する場合は 0</p> <p><code>backlog</code> - キューの最大長</p> <p>例外:</p> <p><code>IOException</code> - ソケットを開いているときに入出力エラーが発生した場合</p> <p><code>SecurityException</code> - セキュリティーマネージャーが存在し、その <code>checkListen</code> メソッドがこの操作を許可しない場合</p>
<p><code>public ServerSocket(int port, int backlog, InetAddress bindAddr)</code> throws <code>IOException</code></p>	<p>指定されたポート、待機バックログ、およびバインド先のローカル IP アドレスを使ってサーバーを作成。複数ホームのホストの場合は、<code>bindAddr</code> 引数を使用すれば、特定のアドレスに対する接続要求だけを受信する <code>ServerSocket</code> を作成できる。<code>bindAddr</code> が <code>null</code> の場合、これはデフォルトで、任意の (すべての) ローカルアドレス上の接続を受け入る。ポートは 0 から 65535 まででなければならない。</p> <p>セキュリティーマネージャーが存在する場合、このメソッドによってその <code>checkListen</code> メソッドが <code>port</code> 引数をその引数として指定して呼び出され、この操作が許可されるかどうかを確認される。この結果、<code>SecurityException</code> がスローされることがある。</p> <p><code>backlog</code> 引数は、0 より大きい正の値である必要がある。渡された値が 0 以下の場合は、デフォルトの値が使用される。</p> <p>パラメタ:</p> <p><code>port</code> - ローカル TCP ポート</p> <p><code>backlog</code> - 待機するバックログ</p> <p><code>bindAddr</code> - サーバーをバインドするローカル <code>InetAddress</code></p> <p>例外:</p> <p><code>SecurityException</code> - セキュリティーマネージャーが存在し、その <code>checkListen</code> メソッドがこの操作を許可しない場合</p> <p><code>IOException</code> - ソケットを開いているときに入出力エラーが発生した場合</p>
メソッド	

<p><b>public void bind(SocketAddress endpoint)</b> throws IOException</p>	<p>ServerSocket を特定のアドレス (IP アドレスおよびポート番号) にバインドする。アドレスが null の場合は、システムにより一時的なポートと有効なローカルアドレスが選択されてソケットがバインドされる。</p> <p>パラメタ: endpoint - バインド先の IP アドレスおよびポート番号</p> <p>例外: IOException - バインド操作に失敗した場合、あるいはソケットがすでにバインドされている場合 SecurityException - SecurityManager が存在し、その checkListen メソッドがこの操作を許可しない場合 IllegalArgumentException - 端点が、このソケットによってサポートされていない SocketAddress サブクラスである場合</p>
<p><b>public void bind(SocketAddress endpoint, int backlog)</b> throws IOException</p>	<p>ServerSocket を特定のアドレス (IP アドレスおよびポート番号) にバインドする。アドレスが null の場合は、システムにより一時的なポートと有効なローカルアドレスが選択されてソケットがバインドされる。</p> <p>backlog 引数は、0 より大きい正の値である必要がある。渡された値が 0 以下の場合は、デフォルトの値が使用される。</p> <p>パラメタ: endpoint - バインド先の IP アドレスおよびポート番号 backlog - 待機するバックログの長さ</p> <p>例外: IOException - バインド操作に失敗した場合、あるいはソケットがすでにバインドされている場合 SecurityException - SecurityManager が存在し、その checkListen メソッドがこの操作を許可しない場合 IllegalArgumentException - 端点が、このソケットによってサポートされていない SocketAddress サブクラスである場合</p>
<p><b>public InetAddress getInetAddress()</b></p>	<p>このサーバーソケットのローカルアドレスを返す。</p> <p>戻り値: このソケットのバインド先アドレス。ソケットがバインドされていない場合は null</p>
<p><b>public int getLocalPort()</b></p>	<p>このソケットが接続を待機中のポートを返す。</p> <p>戻り値: このソケットが待機するポート番号。ソケットがまだバインドされていない場合は -1</p>
<p><b>public SocketAddress getLocalSocketAddress()</b></p>	<p>このソケットがバインドされている端点のアドレスを返す。ソケットがバインドされていない場合は null を返す。</p> <p>戻り値: このソケットのローカル端点を表す SocketAddress。ソケットがまだバインドされていない場合は null</p>
<p><b>public Socket accept()</b> throws IOException</p>	<p>このソケットに対する接続要求を待機し、それを受け取る。このメソッドは接続が行われるまでブロックされる。</p> <p>新しいソケット s が作成され、セキュリティーマネージャーが存在する場合には、その checkAccept メソッドが引数として s.getInetAddress().getHostAddress() および s.getPort() を指定して呼び出され、この操作の実行が許可されていることが確認される。この結果、SecurityException がスローされることがある。</p> <p>戻り値: 新しいソケット</p> <p>例外: IOException - 接続の待機中に入出力エラーが発生した場合 SecurityException - セキュリティーマネージャーが存在し、その checkAccept メソッドがこの操作を許可しない場合 SocketTimeoutException - 以前に setSoTimeout を使ってタイムアウトが設定されていて、そのタイムアウトに達した場合</p>



	<p><b>IllegalBlockingModeException</b> - このソケットに関連するチャンネルが存在し、そのチャンネルが非ブロッキングモードであり、受け入れ準備の整った接続が存在しない場合</p>
<p>protected final void <b>implAccept(Socket s)</b> throws IOException</p>	<p>ServerSocket のサブクラスは、このメソッドを使って <b>accept()</b> をオーバーライドすることで、独自のサブクラスのソケットが返されるようにする。したがって、FooServerSocket は通常、このメソッドに「空」の FooSocket を渡す。implAccept から戻ると、その FooSocket がクライアントに接続される。</p> <p>パラメタ: s - ソケット</p> <p>例外: IllegalBlockingModeException - このソケットに関連するチャンネルが存在し、そのチャンネルが非ブロックモードである場合 IOException - 接続の待機中に入出力エラーが発生した場合</p>
<p>public void <b>close()</b> throws IOException</p>	<p>このソケットを閉じる。accept() で現在ブロックされているスレッドはすべて、SocketException をスローする。このソケットに関連するチャンネルが存在する場合は、そのチャンネルも閉じられる。</p> <p>例外: IOException - ソケットを閉じるときに入出力エラーが発生した場合</p>
<p>public ServerSocketChannel <b>getChannel()</b></p>	<p>このソケットに関連する固有の ServerSocketChannel オブジェクトを返す (存在する場合)。チャンネル自体が ServerSocketChannel.open メソッドを使用して作成された場合にだけ、サーバーソケットにチャンネルが存在する。</p> <p>戻り値: このソケットに関連付けられたサーバーソケットチャンネル。このソケットがチャンネル用に作成されたものでない場合は null</p>
<p>public boolean <b>isBound()</b></p>	<p>ServerSocket のバインディング状態を返す。</p> <p>戻り値: ServerSocket が正常にアドレスにバインドされている場合は true</p>
<p>public boolean <b>isClosed()</b></p>	<p>ServerSocket の閉じた状態を返す。</p> <p>戻り値: ソケットが閉じた場合は true</p>
<p>public void <b>setSoTimeout(int timeout)</b> throws SocketException</p>	<p>指定されたタイムアウト (ミリ秒) を使って SO_TIMEOUT を有効または無効にする。このオプションを 0 以外のタイムアウトに設定すると、この ServerSocket の accept() 呼び出しが、その時間の間だけにブロックされる。タイムアウトの期限が切れると、ServerSocket がまだ有効であっても java.net.SocketTimeoutException が発行される。このオプションは、ブロック処理に入る前に有効にしておく必要がある。タイムアウトは 0 より大きい値を指定する。タイムアウト 0 は無限のタイムアウトとして解釈される。</p> <p>パラメタ: timeout - ミリ秒で表される、指定されたタイムアウト</p> <p>例外: SocketException - 使用しているプロトコルでエラー (TCP エラーなど) が発生した場合</p>
<p>public int <b>getSoTimeout()</b> throws IOException</p>	<p>SO_TIMEOUT の設定を取得。このオプションが無効 (タイムアウトが無限) の場合は 0 を返す。</p> <p>戻り値: SO_TIMEOUT 値</p> <p>例外: IOException - 入出力エラーが発生した場合</p>
<p>public void <b>setReuseAddress(boolean on)</b> throws SocketException</p>	<p>SO_REUSEADDR ソケットオプションを有効または無効にする。</p> <p>TCP 接続をクローズする場合、接続クローズ後の一定期間、その接続がタイムアウト状態 (通常、TIME_WAIT 状態または 2MSL 待機状態と呼ばれる) にとどまる可能性がある。周知のソケットアドレスまたはポートを使用するアプリケーションの場合、ソケットア</p>

	<p>ドレスまたはポートに関連する接続がタイムアウト状態にあると、ソケットに必要な <code>SocketAddress</code> にバインドできないことがある。</p> <p><code>bind(SocketAddress)</code> を使用してソケットをバインドする前に <code>SO_REUSEADDR</code> を有効にすると、以前の接続がタイムアウト状態でもソケットをバインドすることができる。</p> <p><code>ServerSocket</code> が作成されると、<code>SO_REUSEADDR</code> の初期設定は定義されていない。アプリケーションは、<code>getReuseAddress()</code> を使って <code>SO_REUSEADDR</code> の初期設定を確認できる。ソケットがバインドされた (<code>isBound()</code> を参照) あとで <code>SO_REUSEADDR</code> を有効または無効にする場合の動作は定義されていない。</p> <p>パラメタ:</p> <p>on - ソケットオプションを有効にするか無効にするかを指定</p> <p>例外:</p> <p><code>SocketException</code> - <code>SO_REUSEADDR</code> ソケットオプションの有効化または無効化時にエラーが発生した場合、またはソケットがクローズされている場合</p>
<p><code>public boolean getReuseAddress()</code> throws <code>SocketException</code></p>	<p><code>SO_REUSEADDR</code> が有効かどうかを調べる。</p> <p>戻り値: <code>SO_REUSEADDR</code> が有効かどうかを示す <code>boolean</code> 値</p> <p>例外: <code>SocketException</code> - 使用しているプロトコルでエラー (TCP エラーなど) が発生した場合</p>
<p><code>public String toString()</code></p>	<p>このソケットの実装アドレスと実装ポートを <code>String</code> として返す。</p> <p>オーバーライド: クラス <code>Object</code> 内の <code>toString</code></p>
<p><code>public static void setSocketFactory(SocketImplFactory fac)</code> throws <code>IOException</code></p>	<p>アプリケーションのサーバーソケット実装ファクトリを設定。ファクトリを指定できるのは一度だけである。</p> <p>アプリケーションで新しいサーバーソケットを作成すると、ソケット実装ファクトリの <code>createSocketImpl</code> メソッドが呼び出され、実際のソケットが作成される。このメソッドに <code>null</code> を渡しても、ファクトリがすでに設定されていないかぎり、それは無操作になる。</p> <p>セキュリティーマネージャーが存在する場合、このメソッドは最初にセキュリティーマネージャーの <code>checkSetFactory</code> メソッドを呼び出すことにより、この操作が許可されていることを確認する。この結果、<code>SecurityException</code> がスローされることがある。</p> <p>パラメタ:</p> <p>fac - 目的のファクトリ</p> <p>例外:</p> <p><code>IOException</code> - ソケットファクトリの設定中に入出力エラーが発生した場合</p> <p><code>SocketException</code> - ファクトリがすでに定義されている場合</p> <p><code>SecurityException</code> - セキュリティーマネージャーが存在し、その <code>checkSetFactory</code> メソッドがこの操作を許可しない場合</p>
<p><code>public void setReceiveBufferSize(int size)</code> throws <code>SocketException</code></p>	<p>この <code>ServerSocket</code> から受け入れられたソケットの <code>SO_RCVBUF</code> オプションのデフォルト推奨値を設定。受け入れられたソケット内に実際に設定されている値を確認のこと。それには、<code>accept()</code> からソケットが返されたあとで <code>Socket.getReceiveBufferSize()</code> を呼び出す。</p> <p><code>SO_RCVBUF</code> の値は、内部ソケット受信バッファのサイズの設定と、リモートピアに通知される TCP 受信ウィンドウのサイズの設定の両方に使用される。</p> <p>その後、<code>Socket.setReceiveBufferSize(int)</code> を呼び出すことで値を変更できる。ただし、アプリケーションが RFC1323 で定義されている 64K バイトを超える受信ウィンドウを使用可能にする必要がある場合には、ローカルアドレスにバインドする前に推奨値を <code>ServerSocket</code> で設定する必要がある。つまり、引数なしコンストラクタを使って <code>ServerSocket</code> を作成し、次に <code>setReceiveBufferSize()</code> を呼び出し、最後に <code>bind()</code> を呼び出して <code>ServerSocket</code> をアドレスにバインドする必要があることを意味する。</p> <p>これに失敗してもエラーは発生せず、バッファ・サイズは要求された値に設定される。ただし、この <code>ServerSocket</code> から受け取るソケットの TCP 受信ウィンドウは 64K バイト以下になる。</p> <p>パラメタ:</p> <p>size - 受信バッファ・サイズの設定サイズ。この値は 0 より大きくなければならない</p>

	<p>例外:  <b>SocketException</b> - 使用しているプロトコルでエラー (TCP エラーなど) が発生した場合  <b>IllegalArgumentException</b> - 値が 0 または負の値である場合</p>
<p><b>public int getReceiveBufferSize()</b>  throws <b>SocketException</b></p>	<p>この <b>ServerSocket</b> で使われる <b>SO_RCVBUF</b> オプションの値を取得。これは、この <b>ServerSocket</b> から受け取るソケットに使用される推奨バッファ・サイズである。  受け入れたソケットに実際に設定された値は、<b>Socket.getReceiveBufferSize()</b> を呼び出して判定することに注意のこと。  戻り値:  この <b>Socket</b> の <b>SO_RCVBUF</b> オプションの値  例外:  <b>SocketException</b> - 使用しているプロトコルでエラー (TCP エラーなど) が発生した場合</p>
<p><b>public void setPerformancePreferences(int connectionTime, int latency, int bandwidth)</b></p>	<p>この <b>ServerSocket</b> のパフォーマンス設定を行う。  ソケットはデフォルトで、TCP/IP プロトコルを使用する。実装によっては、TCP/IP とは異なるパフォーマンス特性を持つ代替プロトコルを提供することもある。アプリケーションは、このメソッドを使用することで、実装で利用可能なプロトコルの選択時に、これらのかね合いの取り方を示す独自の設定を表現できる。  パフォーマンス設定は、接続時間の長さ、応答時間の速さ、および帯域幅の広さの相対的な重要度を示す 3 つの整数値によって記述される。これらの整数の絶対値は重要ではない。ある特定のプロトコルを選択するために、これらの値が単純に比較されるが、その際、値が大きければより強い設定を示す。たとえば、アプリケーションが応答時間の速さや帯域幅の広さよりも接続時間の短さを優先する場合には、値 (1, 0, 0) を指定してこのメソッドを呼び出すことができる。アプリケーションが応答時間の速さよりも帯域幅の広さを優先し、接続時間の短さよりも待ち時間の短さを優先する場合には、値 (0, 1, 2) を指定してこのメソッドを呼び出すことができる。  このソケットのバインド後にこのメソッドを呼び出しても、何の効果もない。つまり、この機能を使用するには、引数なしコンストラクタでソケットを作成する必要がある。  パラメタ:  <b>connectionTime</b> - 接続時間の長さの相対的な重要度を表す <b>int</b>  <b>latency</b> - 応答時間の速さの相対的な重要度を表す <b>int</b>  <b>bandwidth</b> - 帯域幅の広さの相対的な重要度を表す <b>int</b></p>

### 3.4.3 DatagramSocket クラス

以下は Javadoc からの転記である:

このクラスは、データグラムパケットを送受信するためのソケットを表現している。データグラムソケットは、パケット配信サービスの送信点または受信点である。データグラムソケット上で送信または受信する各パケットは、それぞれ異なるアドレスで経路を指定される。あるマシンから別のマシンに複数のパケットを送信する場合、各パケットは異なる経路で送信される可能性があり、宛先には任意の順序で到達する可能性がある。またパケットの欠落も存在し得る。アプリケーションはそのような状態に対応できなければならない。

UDP ブロードキャストの送信は、**DatagramSocket** 上で常に有効になっている。ブロードキャストパケットを受信するときは、**DatagramSocket** をワイルドカードアドレスにバインドする。実装によっては、**DatagramSocket** が特定のアドレスにバインドされていてもブロードキャストパケットを受信する場合もあり得る。

例:**DatagramSocket s = new DatagramSocket(null); s.bind(new InetSocketAddress(8888));**。これは、**DatagramSocket s = new DatagramSocket(8888);** と同等である。どちらの方法でも、UDP ポート 8888 上でブロードキャストを受信する **DatagramSocket** を作成できる。

表 3-4: DatagramSocket クラスのコンストラクタとメソッド

コンストラクタ	
<p><b>public DatagramSocket()</b> throws SocketException</p>	<p>データグラムソケットを構築し、ローカルホストマシン上の使用可能なポートにバインドする。ソケットはワイルドカードアドレス (カーネルによって選択された任意の IP アドレス) にバインドされる。</p> <p>セキュリティーマネージャーが存在する場合、最初にその <code>checkListen</code> メソッドが 0 を引数に指定して呼び出され、この操作が許可されるかどうかを確認される。この結果、<code>SecurityException</code> がスローされることがある。</p> <p>例外:  <code>SocketException</code> - ソケットを開くことができなかった場合、または指定されたローカルポートにソケットをバインドできなかった場合  <code>SecurityException</code> - セキュリティーマネージャーが存在し、その <code>checkListen</code> メソッドがこの操作を許可しない場合</p>
<p><b>protected DatagramSocket(DatagramSocketImpl impl)</b></p>	<p>指定された <code>DatagramSocketImpl</code> を使用してバインドされていないデータグラムソケットを作成。</p> <p>パラメタ:  <code>impl</code> - サブクラスが <code>DatagramSocket</code> 上で使用する <code>DatagramSocketImpl</code> のインスタンス</p>
<p><b>public DatagramSocket(SocketAddress bindaddr)</b> throws SocketException</p>	<p>指定されたローカルアドレスにバインドされたデータグラムソケットを作成する。アドレスが <code>null</code> の場合は、バインドされていないソケットを作成する。</p> <p>セキュリティーマネージャーが存在する場合は、まずセキュリティーマネージャーの <code>checkListen</code> メソッドがソケットアドレスのポートを引数として呼び出され、操作が許可されるかどうかを確認される。この結果、<code>SecurityException</code> がスローされることがある。</p> <p>パラメタ:  <code>bindaddr</code> - バインドするローカルソケットアドレス。バインドされていないソケットの場合は <code>null</code></p> <p>例外:  <code>SocketException</code> - ソケットを開くことができなかった場合、または指定されたローカルポートにソケットをバインドできなかった場合  <code>SecurityException</code> - セキュリティーマネージャーが存在し、その <code>checkListen</code> メソッドがこの操作を許可しない場合</p>
<p><b>public DatagramSocket(int port)</b> throws SocketException</p>	<p>データグラムソケットを構築し、ローカルホストマシン上の指定されたポートにバインドする。ソケットはワイルドカードアドレス (カーネルによって選択された任意の IP アドレス) にバインドされる。</p> <p>セキュリティーマネージャーが存在する場合、その <code>checkListen</code> メソッドが <code>port</code> 引数をその引数として指定して呼び出され、この操作が許可されるかどうかを確認される。この結果、<code>SecurityException</code> がスローされることがある。</p> <p>パラメタ:  <code>port</code> - 使用するポート</p> <p>例外:  <code>SocketException</code> - ソケットを開くことができなかった場合、または指定されたローカルポートにソケットをバインドできなかった場合  <code>SecurityException</code> - セキュリティーマネージャーが存在し、その <code>checkListen</code> メソッドがこの操作を許可しない場合</p>
<p><b>public DatagramSocket(int port, InetAddress laddr)</b> throws SocketException</p>	<p>指定されたローカルアドレスにバインドされたデータグラムソケットを作成。ローカルポートは、0 ~ 65535 の範囲で指定する。IP アドレスが 0.0.0.0 の場合、ソケットはワイルドカードアドレス (カーネルによって選択された任意の IP アドレス) にバインドされる。</p> <p>セキュリティーマネージャーが存在する場合、その <code>checkListen</code> メソッドが <code>port</code> 引数をその引数として指定して呼び出され、この操作が許可されるかどうかを確認される。この結果、<code>SecurityException</code> がスローされることがある。</p> <p>パラメタ:  <code>port</code> - 使用するローカルポート</p>

	<p><b>laddr</b> - バインド先のローカルアドレス</p> <p>例外:</p> <p><b>SocketException</b> - ソケットを開くことができなかった場合、または指定されたローカルポートにソケットをバインドできなかった場合</p> <p><b>SecurityException</b> - セキュリティーマネージャーが存在し、その <b>checkListen</b> メソッドがこの操作を許可しない場合</p>
メソッド	
<p><b>public void bind(SocketAddress addr)</b> throws <b>SocketException</b></p>	<p>この <b>DatagramSocket</b> を特定のアドレスおよびポートにバインドする。アドレスが <b>null</b> の場合は、システムにより一時的なポートと有効なローカルアドレスが選択されてソケットがバインドされる。</p> <p>パラメタ:</p> <p><b>addr</b> - バインド先のアドレスおよびポート</p> <p>例外:</p> <p><b>SocketException</b> - バインド時にエラーが発生した場合、またはソケットがすでにバインドされている場合</p> <p><b>SecurityException</b> - セキュリティーマネージャーが存在し、その <b>checkListen</b> メソッドがこの操作を許可しない場合</p> <p><b>IllegalArgumentException</b> - <b>addr</b> がこのソケットでサポートされていない <b>SocketAddress</b> サブクラスである場合</p>
<p><b>public void connect(InetAddress address, int port)</b></p>	<p>ソケットをこのソケットのリモートアドレスに接続。ソケットがリモートアドレスに接続されると、そのアドレスとの間でしかパケットを送受信できなくなる。デフォルトでは、データグラムソケットは接続されない。</p> <p>ソケットを接続するリモート接続先が存在しないか到達不可能の場合、およびそのアドレスに対する <b>ICMP</b> 転送先到達不能パケットを受信した場合は、以降の送信または受信呼び出しで <b>PortUnreachableException</b> がスローされることがある。例外が必ずスローされるとは限らないことに注意のこと。</p> <p>指定されたホストおよびポートとのデータグラムの送受信を行うための呼び出し側のアクセス権は、接続時に確認される。ソケットが接続されたとき、送受信ではパケットの受信および送信に対するセキュリティチェックを行わない。ただし、パケットとソケットのアドレスおよびポートが一致するかどうかの確認は行う。送信の処理では、パケットのアドレスが設定されている場合に、パケットのアドレスとソケットのアドレスが一致しないときは、<b>IllegalArgumentException</b> がスローされる。マルチキャストアドレスに接続されているソケットは、送信パケットだけに使用できる。</p> <p>パラメタ:</p> <p><b>address</b> - ソケットが使うリモートアドレス</p> <p><b>port</b> - ソケットが使うリモートポート</p> <p>例外:</p> <p><b>IllegalArgumentException</b> - アドレスが <b>null</b> である場合、またはポートが範囲外の場合</p> <p><b>SecurityException</b> - 指定されたアドレスおよびポートとのデータグラムの送受信が、呼び出し側に許可されていない場合</p>
<p><b>public void connect(SocketAddress addr)</b> throws <b>SocketException</b></p>	<p>このソケットをリモートソケットアドレス (<b>IP</b> アドレス + ポート番号) に接続する。</p> <p>パラメタ:</p> <p><b>addr</b> - リモートアドレス</p> <p>例外:</p> <p><b>SocketException</b> - 接続に失敗した場合</p> <p><b>IllegalArgumentException</b> - <b>addr</b> が <b>null</b> である場合、または <b>addr</b> がこのソケットでサポートされていない <b>SocketAddress</b> サブクラスである場合</p>
<p><b>public void disconnect()</b></p>	<p>ソケットを切断する。ソケットが接続されていない場合は、何も行わない。</p>
<p><b>public boolean isBound()</b></p>	<p>ソケットのバインディング状態を返す。</p> <p>戻り値:</p> <p>ソケットが正常にアドレスにバインドされている場合は <b>true</b></p>

<code>public InetAddress <b>getInetAddress</b>()</code>	このソケットの接続先のアドレスを返す。ソケットが接続されていない場合は <code>null</code> を返す。
<code>public int <b>getPort</b>()</code>	このソケットのためのポートを返す。ソケットが接続されていない場合は <code>-1</code> を返す。
<code>public SocketAddress <b>getRemoteSocketAddress</b>()</code>	このソケットが接続されている端点のアドレスを返す。ソケットが接続されていない場合は <code>null</code> を返す。 戻り値: このソケットのリモート端点を表す <code>SocketAddress</code> 。ソケットがまだ接続されていない場合は <code>null</code>
<code>public SocketAddress <b>getLocalSocketAddress</b>()</code>	このソケットがバインドされている端点のアドレスを返す。ソケットがバインドされていない場合は <code>null</code> を返す。 戻り値: このソケットのローカル端点を表す <code>SocketAddress</code> 。ソケットがまだバインドされていない場合は <code>null</code>
<code>public void <b>send</b>(DatagramPacket p) throws IOException</code>	このソケットからデータグラムパケットを送信する。 <code>DatagramPacket</code> には、送信するデータ、データの長さ、リモートホストの IP アドレス、およびリモートホスト上のポート番号などの情報が格納されている。 セキュリティマネージャーが存在し、ソケットが現在リモートアドレスに接続されていない場合、このメソッドは最初にいくつかのセキュリティチェックを行う。まず、 <code>p.getAddress().isMulticastAddress()</code> が <code>true</code> である場合、このメソッドは <code>p.getAddress()</code> を引数としてセキュリティマネージャーの <code>checkMulticast</code> メソッドを呼び出す。その式の評価が <code>false</code> の場合、このメソッドは代わりに、セキュリティマネージャーの <code>checkConnect</code> メソッドを、引数 <code>p.getAddress().getHostAddress()</code> と <code>p.getPort()</code> を指定して呼び出す。それぞれのセキュリティマネージャーメソッド呼び出しの結果、操作が許可されない場合は <code>SecurityException</code> がスローされる。 パラメタ: <code>p</code> - 送信される <code>DatagramPacket</code> 例外: <code>IOException</code> - 入出力エラーが発生した場合 <code>SecurityException</code> - セキュリティマネージャーが存在し、その <code>checkMulticast</code> または <code>checkConnect</code> メソッドが送信を許可しない場合 <code>PortUnreachableException</code> - 現在到達不可能になっている宛先にソケットが接続されている場合にスローされる可能性がある。例外が必ずスローされるとは限らないことに注意 <code>IllegalBlockingModeException</code> - ソケットに関連したチャンネルが存在し、そのチャンネルが非ブロックモードの場合
<code>public void <b>receive</b>(DatagramPacket p) throws IOException</code>	このソケットからのデータグラムパケットを受信する。このメソッドが復帰すると、 <code>DatagramPacket</code> のバッファには受信したデータが格納される。データグラムパケットには、送信者の IP アドレスと、送信者のマシンのポート番号も格納されている。 このメソッドはデータグラムが受信されるまでブロックされる。データグラムパケットオブジェクトの <code>length</code> フィールドには、受信されるメッセージの長さの情報が格納されている。メッセージがパケットよりも長い場合、メッセージは切り詰められる。 セキュリティマネージャーが存在する場合、セキュリティマネージャーの <code>checkAccept</code> メソッドがパケットの受信を許可しないときは、パケットの受信は行われない。 パラメタ: <code>p</code> - 受信したデータを保存する <code>DatagramPacket</code> 例外: <code>IOException</code> - 入出力エラーが発生した場合 <code>SocketTimeoutException</code> - <code>setSoTimeout</code> が以前に呼び出されて、タイムアウトが過ぎた場合 <code>PortUnreachableException</code> - 現在到達不可能になっている宛先にソケットが接続されている場合にスローされる可能性がある。例外が必ずスローされるとは限らないことに注意 <code>IllegalBlockingModeException</code> - ソケットに関連したチャンネルが存在し、そのチャンネルが非ブロックモードの場合

<p><code>public InetAddress getLocalAddress()</code></p>	<p>ソケットのバインド先のローカルアドレスを取得する。 セキュリティマネージャが存在する場合は、まずセキュリティマネージャの <code>checkConnect</code> メソッドがホストアドレスおよび <code>-1</code> を引数として呼び出され、操作が許可されるかどうかを確認される。 戻り値: ソケットのバインド先のローカルアドレス。ソケットがバインドされないか、またはセキュリティマネージャの <code>checkConnect</code> メソッドがこの操作を許可しない場合は、任意のローカルアドレスを表す <code>InetAddress</code></p>
<p><code>public int getLocalPort()</code></p>	<p>このソケットのバインド先となる、ローカルホスト上のポート番号を返す。</p>
<p><code>public void setSoTimeout(int timeout) throws SocketException</code></p>	<p>指定されたタイムアウト (ミリ秒) を使って <code>SO_TIMEOUT</code> を有効または無効にする。このオプションにゼロ以外の待ち時間を設定した場合、この <code>DatagramSocket</code> に対して <code>receive()</code> を呼び出すと、設定した時間だけブロックされる。タイムアウトの期限が切れると、<code>DatagramSocket</code> がまだ有効であっても <code>java.net.SocketTimeoutException</code> が発行される。このオプションは、ブロック処理に入る前に有効にしておく必要がある。タイムアウトは 0 より大きい値を指定する。タイムアウト 0 は無限のタイムアウトとして解釈される。 パラメタ: <code>timeout</code> - ミリ秒で表される、指定された待ち時間 例外: <code>SocketException</code> - 基本となるプロトコルで UDP エラーなどのエラーが発生した場合</p>
<p><code>public int getSoTimeout() throws SocketException</code></p>	<p><code>SO_TIMEOUT</code> の設定を取得する。このオプションが無効 (タイムアウトが無限) の場合は 0 を返す。 戻り値: <code>SO_TIMEOUT</code> の設定 例外: <code>SocketException</code> - 基本となるプロトコルで UDP エラーなどのエラーが発生した場合</p>
<p><code>public void setSendBufferSize(int size) throws SocketException</code></p>	<p><code>SO_SNDBUF</code> オプションを、この <code>DatagramSocket</code> に指定された値に設定。 <code>SO_SNDBUF</code> オプションは、使用するネットワーク入出力バッファのサイズを設定するヒントとして、ネットワーク実装が使う。また、ネットワーク実装は、このソケットで送信できるパケットの最大サイズを判定するためにも <code>SO_SNDBUF</code> 設定を使う。 <code>SO_SNDBUF</code> はヒントなので、アプリケーションでバッファのサイズを調べる必要がある場合は、<code>getSendBufferSize()</code> を呼び出すこと。 バッファ・サイズを大きくすると、送信速度が高い場合にネットワーク実装により複数の送信パケットをキューに入れることが可能になる。 注:<code>send(DatagramPacket)</code> を使用して <code>SO_SNDBUF</code> の設定より大きい <code>DatagramPacket</code> を送信する場合、パケットが送信されるか破棄されるかは実装によって異なる。 パラメタ: <code>size</code> - 送信バッファ・サイズの設定サイズ。この値は 0 より大きくなければならない</p>
<p><code>public int getSendBufferSize() throws SocketException</code></p>	<p>この <code>DatagramSocket</code> で使われる <code>SO_SNDBUF</code> オプションの値を取得する。これは、この <code>DatagramSocket</code> で出力用としてプラットフォームが使うバッファのサイズである。 戻り値: この <code>DatagramSocket</code> の <code>SO_SNDBUF</code> オプションの値 例外: <code>SocketException</code> - 基本となるプロトコルで UDP エラーなどのエラーが発生した場合</p>
<p><code>public void setReceiveBufferSize(int size) throws SocketException</code></p>	<p><code>SO_RCVBUF</code> オプションを、この <code>DatagramSocket</code> に指定された値に設定する。 <code>SO_RCVBUF</code> オプションは、使用するネットワーク入出力バッファのサイズを設定するヒントとして、ネットワーク実装が使う。また、ネットワーク実装は、このソケットで受信できるパケットの最大サイズを判定するためにも <code>SO_RCVBUF</code> 設定を使う。 <code>SO_RCVBUF</code> はヒントなので、アプリケーションでバッファのサイズ設定を調べる必要がある場合は、<code>getReceiveBufferSize()</code> を呼び出すこと。 <code>SO_RCVBUF</code> の値を大きくすると、<code>receive(DatagramPacket)</code> を使用した受信より速くパケットが到達する場合に、ネットワーク実装による複数のパケットのバッファリングが可能になる場合がある。</p>

	<p>注:SO_RCVBUF より大きいパケットを受信できるかどうかは実装によって異なる。  パラメタ:  size - 受信バッファ・サイズの設定サイズ。この値は 0 より大きくなければならない  例外:  SocketException - 基本となるプロトコルで UDP エラーなどのエラーが発生した場合  IllegalArgumentException - 値が 0 または負の値である場合</p>
<p>public int <b>getReceiveBufferSize()</b>  throws SocketException</p>	<p>この DatagramSocket で使われる SO_RCVBUF オプションの値を取得する。これは、この DatagramSocket で入力用としてプラットフォームが使うバッファのサイズである。  戻り値:  この DatagramSocket の SO_RCVBUF オプションの値  例外:  SocketException - 基本となるプロトコルで UDP エラーなどのエラーが発生した場合</p>
<p>public void <b>setReuseAddress(boolean on)</b>  throws SocketException</p>	<p>SO_REUSEADDR ソケットオプションを有効または無効にする。  UDP ソケットの場合、複数のソケットを同じソケットアドレスにバインドすることが必要になることがある。マルチキャストパケットを受信するためには通常このようにする (MulticastSocket を参照)。bind(SocketAddress) を使用してソケットをバインドする前に SO_REUSEADDR ソケットオプションが有効になっていれば、SO_REUSEADDR ソケットオプションを使って複数のソケットを同一のソケットアドレスにバインドすることができる。  注:この機能は、既存のすべてのプラットフォームでサポートされているわけではない。したがって、このオプションが無視されるかどうかは実装によって異なる。ただし、この機能がサポートされていない場合、getReuseAddress() は常に false を返す。  DatagramSocket が作成されると、SO_REUSEADDR の初期設定は無効になる。  ソケットがバインドされた (isBound() を参照) あとで SO_REUSEADDR を有効または無効にする場合の動作は定義されていない。  パラメタ:  on - 有効にするか無効にするかを指定  例外:  SocketException - SO_REUSEADDR ソケットオプションの有効化または無効化時にエラーが発生した場合、またはソケットがクローズされている場合</p>
<p>public boolean <b>getReuseAddress()</b>  throws SocketException</p>	<p>SO_REUSEADDR が有効かどうかを調べる。  戻り値:  SO_REUSEADDR が有効かどうかを示す boolean 値  例外:  SocketException - 基本となるプロトコルで UDP エラーなどのエラーが発生した場合</p>
<p>public void <b>setBroadcast(boolean on)</b>  throws SocketException</p>	<p>SO_BROADCAST を有効または無効にする。  パラメタ:  on - ブロードキャストをオンにするかどうかを指定  例外:  SocketException - 基本となるプロトコルで UDP エラーなどのエラーが発生した場合</p>
<p>public boolean <b>getBroadcast()</b>  throws SocketException</p>	<p>SO_BROADCAST が有効かどうかを調べる。  戻り値:  SO_BROADCAST が有効かどうかを示す boolean 値  例外:  SocketException - 基本となるプロトコルで UDP エラーなどのエラーが発生した場合</p>
<p>public void <b>setTrafficClass(int tc)</b>  throws SocketException</p>	<p>この DatagramSocket から送信されるデータグラムの IP データグラムヘッダーのトラフィッククラスまたはサービス型 octet を設定する。使用するネットワーク実装がこの値を無視することがあるので、アプリケーションではこの値をヒントと考えること。  tc の範囲は 0 ≤ tc ≤ 255 でなければならない。そうでない場合は、IllegalArgumentException がスローされる。  注 -</p>



	<p>IP (Internet Protocol) Version 4 の場合、RFC 1349 で説明されているように、この値は優先度の高い octet と TOS フィールドで構成される。TOS フィールドは、次のようにビット単位の論理和によって作成されるビットセットである。</p> <p> IPTOS_LOW COST (0x02)  IPTOS_RELIABILITY (0x04)  IPTOS_THROUGHPUT (0x08)  IPTOS_LOW DELAY (0x10) </p> <p>最下位ビットは、MBZ (0 でなければならない) ビットに対応するので、常に無視される。優先フィールドにビットを設定すると、操作が許可されないことを示す <code>SocketException</code> になることがある。</p> <p>IP (Internet Protocol) Version 6 の場合、tc は IP ヘッダーの <code>sin6_flowinfo</code> フィールドに格納される値である。</p> <p>パラメタ:  tc - ビットセットの int 値</p> <p>例外:  <code>SocketException</code> - トラフィッククラスまたはサービスタイプの設定時にエラーが発生した場合</p>
<p><code>public int getTrafficClass()</code>  throws <code>SocketException</code></p>	<p>この <code>DatagramSocket</code> から送信されるパケットの IP データグラムヘッダーのトラフィッククラスまたはサービス型を取得する。使用するネットワーク実装が、<code>setTrafficClass(int)</code> を使用して設定されたトラフィッククラスまたはサービス型を無視することがあるので、この <code>DatagramSocket</code> で <code>setTrafficClass(int)</code> メソッドを使用して以前に設定された値とは異なる値がこのメソッドから返されることがある。</p> <p>戻り値:  すでに設定されているトラフィッククラスまたはサービス型</p> <p>例外:  <code>SocketException</code> - トラフィッククラスまたはサービスタイプの値を取得する際にエラーが発生した場合</p>
<p><code>public void close()</code></p>	<p>このデータグラムソケットを閉じる。現在このソケットの <code>receive(java.net.DatagramPacket)</code> でブロックされているすべてのスレッドが <code>SocketException</code> をスローする。このソケットに関連するチャンネルが存在する場合は、そのチャンネルも閉じられる。</p>
<p><code>public boolean isClosed()</code></p>	<p>ソケットが閉じたかどうかを返す。</p> <p>戻り値:  ソケットが閉じた場合は <code>true</code></p>
<p><code>public DatagramChannel getChannel()</code></p>	<p>このデータグラムソケットに関連する固有の <code>DatagramChannel</code> オブジェクトを返す (存在する場合)。チャンネル自体が <code>DatagramChannel.open</code> メソッドを使用して作成された場合にだけ、データグラムソケットにチャンネルが存在する。</p> <p>戻り値:  このデータグラムソケットに関連するデータグラムチャンネル。このソケットがチャンネル用に作成されなかった場合は <code>null</code></p>
<p><code>public static void setDatagramSocketImplFactory(DatagramSocketImplFactory fac)</code>  throws <code>IOException</code></p>	<p>アプリケーションのデータグラムソケット実装ファクトリを設定する。ファクトリを指定できるのは一度だけである。</p> <p>アプリケーションで新しいデータグラムソケットを作成すると、ソケット実装ファクトリの <code>createDatagramSocketImpl</code> メソッドが呼び出され、実際のデータグラムソケット実装が作成される。</p> <p>このメソッドに <code>null</code> を渡しても、ファクトリがすでに設定されていないかぎり、それは無操作になる。</p> <p>セキュリティマネージャーが存在する場合、このメソッドは最初にセキュリティマネージャーの <code>checkSetFactory</code> メソッドを呼び出すことにより、この操作が許可されていることを確認する。この結果、<code>SecurityException</code> がスローされることがある。</p> <p>パラメタ:  fac - 目的のファクトリ</p> <p>例外:  <code>IOException</code> - データグラムソケットファクトリの設定中に入出力エラーが発生した場合</p>

	<p>SocketException - ファクトリがすでに定義されている場合</p> <p>SecurityException - セキュリティーマネージャーが存在し、その checkSetFactory メソッドがこの操作を許可しない場合</p>
--	---

### 3.4.4 DatagramPacket クラスのコンストラクタとメソッド

以下は Javadoc からの転記である:

このクラスはデータグラムパケットを表している。データグラムパケットは、無接続パケット配布サービスを実装する際に使用する。各メッセージは、パケット内に含まれている情報だけを基に、あるマシンから別のマシンへ送信される。あるマシンから別のマシンに複数のパケットが送信される場合、それらの各パケットは異なる経路で送信される可能性があり、その到着順序もさまざまな可能性がある。パケットの配信は保証されない。

表 3-5 : DatagramPacket クラスのコンストラクタとメソッド

コンストラクタ	
<p><b>public DatagramPacket</b>(byte[] buf, int offset, int length)</p>	<p>バッファへのオフセットを指定して、長さが length のパケットを受信するための DatagramPacket を構築する。引数 length の値は、buf.length の値以下である必要がある。</p> <p>パラメタ:</p> <p>buf - 着信データグラムを保持するためのバッファ</p> <p>offset - バッファへのオフセット</p> <p>length - 読み込むバイト数</p>
<p><b>public DatagramPacket</b>(byte[] buf, int length)</p>	<p>長さが length のパケットを受信するための DatagramPacket を構築する。引数 length の値は、buf.length の値以下である必要がある。</p> <p>パラメタ:</p> <p>buf - 着信データグラムを保持するためのバッファ</p> <p>length - 読み込むバイト数</p>
<p><b>public DatagramPacket</b>(byte[] buf, int offset, int length, InetAddress address, int port)</p>	<p>長さ length、オフセット offset のパケットを指定されたホスト上の指定されたポート番号に送信するためのデータグラムパケットを構築する。引数 length の値は、buf.length の値以下である必要がある。</p> <p>パラメタ:</p> <p>buf - パケットデータ</p> <p>offset - パケットデータのオフセット</p> <p>length - パケットデータの長さ</p> <p>address - 転送先アドレス</p> <p>port - 転送先ポート番号</p>
<p><b>public DatagramPacket</b>(byte[] buf, int offset, int length, SocketAddress address) throws SocketException</p>	<p>長さ length、オフセット offset のパケットを指定されたホスト上の指定されたポート番号に送信するためのデータグラムパケットを構築する。引数 length の値は、buf.length の値以下である必要がある。</p> <p>パラメタ:</p> <p>buf - パケットデータ</p> <p>offset - パケットデータのオフセット</p> <p>length - パケットデータの長さ</p> <p>address - 転送先ソケットアドレス</p> <p>例外:</p> <p>IllegalArgumentException - アドレス型がサポートされていない場合</p> <p>SocketException</p>

<pre>public DatagramPacket(byte[] buf,     int length,     InetAddress address,     int port)</pre>	<p>長さ <b>length</b> のパケットを指定されたホスト上の指定されたポート番号に送信するためのデータグラムパケットを構築する。引数 <b>length</b> の値は、<b>buf.length</b> の値以下である必要がある。</p> <p>パラメタ:</p> <ul style="list-style-type: none"> <li><b>buf</b> - パケットデータ</li> <li><b>length</b> - パケットの長さ</li> <li><b>address</b> - 転送先アドレス</li> <li><b>port</b> - 転送先ポート番号</li> </ul>
<pre>public DatagramPacket(byte[] buf,     int length,     SocketAddress address)     throws SocketException</pre>	<p>長さ <b>length</b> のパケットを指定されたホスト上の指定されたポート番号に送信するためのデータグラムパケットを構築する。引数 <b>length</b> の値は、<b>buf.length</b> の値以下である必要がある。</p> <p>パラメタ:</p> <ul style="list-style-type: none"> <li><b>buf</b> - パケットデータ</li> <li><b>length</b> - パケットの長さ</li> <li><b>address</b> - 転送先アドレス</li> </ul> <p>例外:</p> <ul style="list-style-type: none"> <li><b>IllegalArgumentException</b> - アドレス型がサポートされていない場合</li> <li><b>SocketException</b></li> </ul>
メソッド	
<pre>public InetAddress getAddress()</pre>	このデータグラムの送信先であるマシン、またはデータグラムの送信元であるマシンの IP アドレスを返す。
<pre>public int getPort()</pre>	このデータグラムの送信先、またはデータグラムの送信元の、リモートホスト上のポート番号を返す。
<pre>public byte[] getData()</pre>	データバッファを返す。受信したデータまたは送信するデータは、バッファ内の <b>offset</b> から始まり、 <b>length</b> の長さだけ続く。
	戻り値: データを受信または送信するのに使うバッファ
<pre>public int getOffset()</pre>	送信するデータのオフセット、または受信したデータのオフセットを返す。
<pre>public int getLength()</pre>	送信するデータの長さ、または受信したデータの長さを返す。
<pre>public void setData(byte[] buf,     int offset,     int length)</pre>	このパケットのデータバッファを設定する。これにより、パケットのデータ、長さ、およびオフセットが設定される。
	パラメタ: <b>buf</b> - このパケット用として設定するバッファ <b>offset</b> - データへのオフセット <b>length</b> - データの長さまたはデータ受信に使用するバッファの長さ、あるいはその両方
	例外: <b>NullPointerException</b> - 引数が <b>null</b> の場合
<pre>public void setAddress(InetAddress iaddr)</pre>	このデータグラムの送信先であるマシンの IP アドレスを設定する。
	パラメタ: <b>iaddr</b> - <b>InetAddress</b>
<pre>public void setPort(int iport)</pre>	このデータグラムの送信先であるリモートホストのポート番号を設定する。
<pre>public void setSocketAddress(SocketAddress address)</pre>	このデータグラムの送信先であるリモートホストの <b>SocketAddress</b> (通常は IP アドレス + ポート番号) を設定する。
	パラメタ: <b>address</b> - <b>SocketAddress</b>
	例外: <b>IllegalArgumentException</b> - アドレスが <b>null</b> であるか、このソケットによってサポートされていない <b>SocketAddress</b> サブクラスである場合

<b>public SocketAddress getSocketAddress()</b>	このパケットの送信先または送信元であるリモートホストの <b>SocketAddress</b> (通常は IP アドレス + ポート番号) を取得する。
<b>public void setData(byte[] buf)</b>	このパケットのデータバッファを設定する。この <b>DatagramPacket</b> のオフセットは 0 に、長さは <b>buf</b> の長さに、それぞれ設定される。 パラメタ: <b>buf</b> - このパケット用として設定するバッファ 例外: <b>NullPointerException</b> - 引数が <b>null</b> の場合
<b>public void setLength(int length)</b>	このパケットの長さを設定する。パケットの長さとは、パケットのデータ・バッファ内の送信対象バイト数、パケットのデータバッファ内でデータ受信に使用されるバイト数、のいずれかである。 <b>length</b> は、オフセットとパケット・バッファ長を足した値以下でなければならない。 パラメタ: <b>length</b> - このパケット用として設定する長さ 例外: <b>IllegalArgumentException</b> - <b>length</b> が負の場合、または <b>length</b> がパケットのデータ・バッファの長さよりも大きい場合

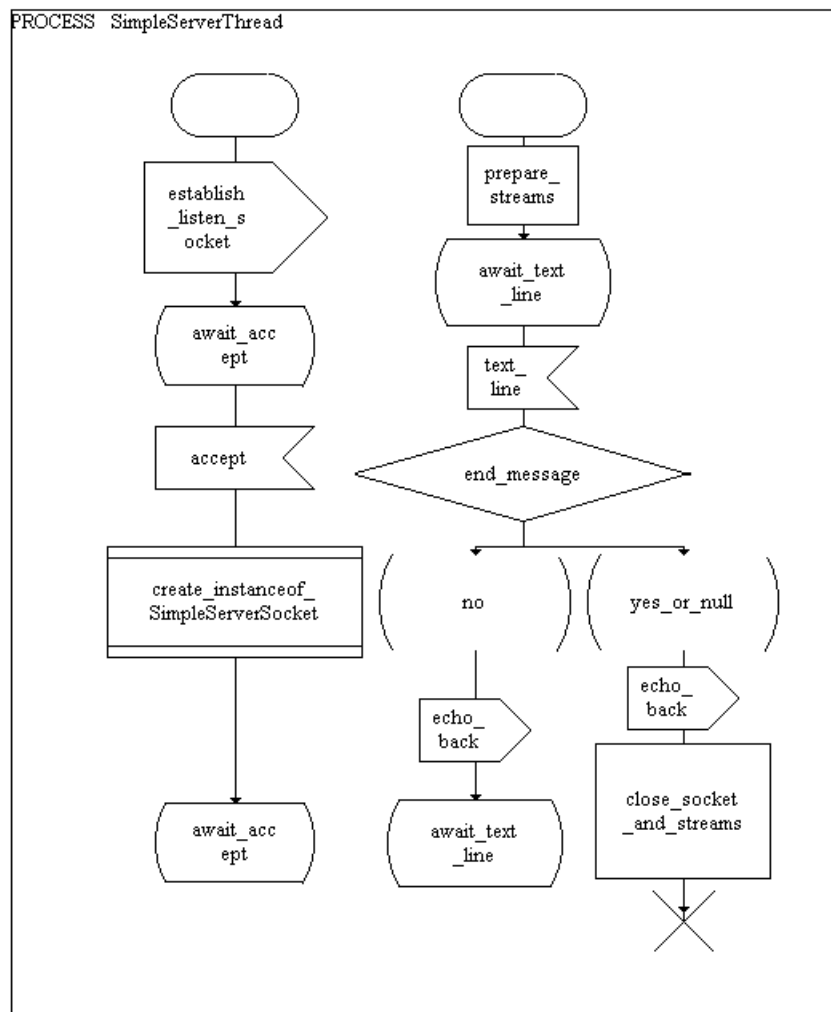
## 第4章 Java NET による TCP 通信のプログラム

これまでのネットワークとソケット・インターフェイスに関する知識をもとに、基本的な TCP 通信を以下に示す。受付窓口ポートが 1000 のサーバ (`SimpleServerThread`) を考えよう。このサーバは非常に簡単なサービス、即ちエコーバックをおこなう。クライアントからテキストを1行受け取ったら、それに受信文字数の情報をつけて送り返す。サーバのクラスはクライアント毎に自分のインスタンスを作るのが常套手段(一寸読みづらい難点があるが)である。

### 4.1節 エコーバック・サーバの動作記述

これを SDL で記述すると下図のようになる。SDL (Specification and Description Language) は当時の CCITT (現在は ITU-T) という国連の国際通信組織が 1976 年ごろから採択された通信システムの仕様と動作記述のための言語で、オブジェクト指向のソフトウェア開発が生まれるよりもずっと前からイベント・ドリブンの記述が採用されていて、現在も通信の世界では一般的な言語である。

図 4-1: `SimpleServerThread` の SDL プロセス図



この図では、ふたつの独立した流れが存在する。左は常駐する部分であり、右は接続してきたクライアント毎に存在するスレッドである。左の流れを見ると、はじめに本サーバの受付窓口のポートをもつソケットを用意する。クライアントが接続要求をしてきて、接続が確立するとソケットは **accept** を返してくる。そしたらそのソケットでこのプロセス(自分自身)のインスタンスを生成して起動する。ポート番号の管理は TCP が行うのでこれに気にかける必要はない。ひとつのスレッドを起したら次のクライアントからの接続要求を待つ。

右側の流れは新しく生成されたクライアント毎のインスタンスの流れである。はじめにソケット及びデータを送受信するストリームを用意する。ついでクライアントからのテキスト1行分の着信を待つ。テキストが受信されたら、"end"かどうか調べる。"end"でなければこれに簡単なヘッダをつけて送り返し、次なるテキスト着信を待つ。"end"であればそれをそのまま送り返し、該ソケットとストリームを閉じて終了する。await\_text\_line の状態でタイム監視が入っていないことに疑問を感じられるであろう。これについては後述する。

このように終了に際しては、アプリケーションの層にあっても相互の了解のもとに終了することが重要である。せっきゃく TCP が前節で述べたように、きちんとした終了(「グレースフル・クローズ: graceful close」と呼ばれる)をやってくれているのに、アプリケーションがいい加減では芳しくない。この際検討すべき事項がある。双方が同じソケットを一緒に閉じたらどうなるか?、また一方が勝手にソケットを閉じたらどうなるか?、相手が異常終

了したらどうなるか？などの異常終了である。例外が発生する場合と別の事象が発生することもある。ネットワークワーキングの場合はこのような異常終了処理が欠かせない。メソッド `close()` は TCP に対してコネクションの切断を指示する。TCP は前に記したようにいわゆるグレースフル・クローズを行い、下位層には問題を残さない。従ってこれらの問題はアプリケーションの問題である。

`close()` がかかったソケットに対してアクセスすると例外が生じる。あとで説明するが、クライアントが勝手にソケットを閉めると、サーバではソケットの例外でなくキャラクタストリームから `null` が帰ってきて、その結果相手からの切断を知る場合がある。

回線の断に関してはもう一つ知っておくべきことがある。どうしても回復できない誤りを一方の TCP が発見したときは、コネクションをリセットするように他方の TCP に指示し、強制的に切断する。リセット指示のセグメントが到来すると TCP は直ちにそのコネクションに関する資源を解放する。その際ユーザに異常終了したことを通知する。この場合の例外動作はまだ確認に至っていない。

ネットワークにからむ例外は以下のようである：

- **BindException**: ソケットをローカルアドレスとポートにバインドしようと試みたがエラーが発生した。そのポートが使用中や要求したローカルアドレスを割り付けることが出来なかったとき発生する。
- **ConnectException**: リモートの IP アドレスとポートに接続を試みたがエラーが発生した。通常リモートのホストが拒絶した(リモートのアドレス/ポートが `listen` していない)とき発生する。
- **IllegalBlockingModeException**: `ServerSocket` クラスで、ソケットがチャンネルを持ち、そのチャンネルが非ブロック・モードである場合に、`accept` および `implAccept` メソッドがスローする。チャンネル(新しいプリミティブ入出力の抽象化)は Java の 1.4 版で導入されている。
- **MalformedURLException**: 異常な URL が発生した。指定された文字列を処理するプロトコルが存在しないか、その文字列が構文解析出来ないとき発生。
- **NoRouteToHostException**: ソケットをリモートのアドレス/ポートに接続しようとしてエラーが発生した。通常介在するファイアウォールやルータが原因で相手に届かないとき発生する。
- **ProtocolException**: TCP など下位層プロトコルでエラーが発生した。
- **SecurityException**: セキュリティ・マネージャが存在してその接続を拒否した。
- **SocketException**: TCP など下位層プロトコルでエラーが発生した。
- **SocketTimeoutException**: タイムアウト時間を過ぎても接続されない。
- **UnknownHostException**: IP アドレスが求まらなかった。通常ドメインネームとそのサーバに起因する。
- **UnknownServiceException**: 読みとり専用の URL 接続に書き込みしようとしたとか、URL 接続に MIME type が帰ってきた(意味がない)時などに発生する。

殆どの場合遭遇する例外は `SocketException` であろう。いずれにしても極力検出した例外はログに残すべきである。そうしたプログラムリストをまず示す。

## 4.2節 エコーバック・サーバのプログラム

以下にこのサーバ(`SimpleServerThread`)のコードを示す。このクラスの名前に `Thread` がついているのは、`Runnable` インターフェイスを実装したスレッド・クラスであるからである。クライアントからの接続待ち受けの為のクラスと、各接続に対する個々のサービス(今回はエコーバック)を同じクラスで構成しているの、わかりづら

いが、このほうがコンパクトなので一般的に使われている。この方法がトリッキーで気に入らない人は、個々のサービスの部分を独立させたスレッド・クラスにしても構わない。

```
package TCPIP_Programming;

import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketAddress;

/**
 * **** SimpleServerThread クラス ****
 * このクラスは、TCP/IP ソケットでクライアントにサービスする
 * サーバスレッドの基本的な実装の見本である。run () メソッドを
 * 必要なアプリケーションにあわせて改造すれば、種々のアプリケーションに
 * 適用可能である。
 * この型は VisualAge で作成されました。
 */

public class SimpleServerThread implements Runnable {
    private Socket server_socket; //相手のクライアントと実際に接続したソケット

    /**
     * SimpleServerThread コンストラクター・コメント。
     * コンストラクタは、指定されたソケットのクライアントと通信を行うための
     * 準備をする。
     */
    public SimpleServerThread(Socket socket) {
        Thread thread;
        server_socket = socket; //相手のクライアントを指定
        thread = new Thread(this); //自分をスレッドにして
        thread.start(); //開始する。
    }

    /**
     * main() では、まず接続要求受理用ポートを開設し、
     * クライアントからの接続要求を待つ。
     * 接続要求を受理し、接続処理を終了したら相手のソケットを
     * もとに新しい SimpleServerSocket のオブジェクトを作り、
     * これを開始させる。従って複数のクライアント/ポート
     * に対応して同数のスレッドオブジェクトが走ることになる。
     * このメソッドは VisualAge で作成されました。
     */
    public static void main(String args[]) {
        ServerSocket listen_socket; //クライアントからの接続要求受理窓口のソケット
        try { //ここではポート番号を 1000 とする
            listen_socket = new ServerSocket(1000);
        } catch (java.io.IOException e) { //例外はコンソールに出力
            System.err.println("Failed to create listen socket.");
            e.printStackTrace();
            System.exit(-1); //しかるのち終了
            return;
        }
        while (true) {
            Socket socket; //クライアントのソケット
            try {
                socket = listen_socket.accept(); //接続要求受理と接続
                new SimpleServerThread(socket); //接続したらそのソケットで新しい
            }
        }
    }
}
```



```

        //自分のインスタンスを作って実行
    }catch(java.io.IOException e){
        e.printStackTrace();
    }
}
}
/**
オーバーライドされた run() メソッドは、単純に受信したメッセージをエコーバックする。
これはシンプルな見本のため、終了処理は単純で、クライアントからの"end"のメッセージを契機とする。
スレッドはこれをクライアントに送り返してソケット、ストリームを閉じて終了する。
* このメソッドは VisualAge で作成されました。
*/
public void run() {
    java.io.BufferedReader inbound_stream;          //受信はBufferedReaderをつかう
    java.io.BufferedWriter echo_back_stream;       //送信はBufferedWriterを使う
    String data;
    SocketAddress client_address;
    try{                                             //ふたつのストリームの構築
        echo_back_stream = new java.io.BufferedWriter(new
java.io.OutputStreamWriter(server_socket.getOutputStream(), "Windows-31J"));
        inbound_stream = new java.io.BufferedReader(new
java.io.InputStreamReader(server_socket.getInputStream(), "Windows-31J"));
    }
    catch (java.io.IOException e){
        e.printStackTrace();
        return;
    }
    client_address = server_socket.getRemoteSocketAddress();
    System.out.println("Start serving : " + client_address);
    boolean end_flag = false;
    do{                                             //エコーバックのループ
        try{
            data = inbound_stream.readLine();      //1行読み込む
            System.out.println(client_address + " : " + data);
            if (data == null){                    //相手が勝手に切断するとnullが帰る!
                end_flag = true;
                break;}
            else{
                if (data.startsWith("end") == true){ // "end"だったらそのまま
                    end_flag = true;              //終了フラグをたてる
                }
                else{                             //でなければ
                    data = "Server received(" + data.length() + " characters): " + data;
                }
            }
        }
        /*                                         // 回線障害テスト用10秒タイマ
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        */
        echo_back_stream.write(data);             //文字数つきでエコーバック
        echo_back_stream.newLine();
        echo_back_stream.flush();

```

```

    }          //クライアントが停止したか勝手にTCP切断したときの例外
    catch (java.net.SocketException se){
        System.out.println("Client disconnected.");
        se.printStackTrace();
        end_flag = true;
        break;
    }
    catch(java.io.IOException e){
        System.out.println("IO Exception occurred.");
        e.printStackTrace();
        end_flag = true;
        break;
    }
}while (end_flag == false);
if (end_flag == true) System.out.println("Completed serving : " + client_address);
//クライアントから"end"がきたらループを脱して
try{    //クリーンアップ処理
    echo_back_stream.close();        //ふたつのストリームを閉じる
    inbound_stream.close();
    server_socket.close();            //ソケットを閉じる
}      //IOエラーの処理
catch (java.io.IOException e){
    System.err.println("IO Exception occurred.");
    e.printStackTrace();
}
}
//このスレッドの終了
}

```

#### 4.2.1.1 このプログラムの動作

このプログラムリストとSDL図を比較すると理解が早い。やたらと例外処理が入っているので見づらい点は容赦いただきたい。はじめにこのクラスの構造を見よう。このクラスは、コンストラクタ、main()、run()の3つのメソッドからなっている。コンストラクタは、相手のクライアントと接続したソケットをパラメタにして、自分自身のインスタンスをスレッド化して開始させる。main()は、常駐部分であり、新しいサーバの待ち受け用のソケットを用意し、クライアントからの接続を待つ。クライアントが接続してきたら、その新しいソケットをもとに前記コンストラクタを呼ぶ。しかるのち次のクライアントからの接続を待つ。これは「TCPのコンセプト」の節で説明したように、待ち受けのソケット・アドレス(即ち待ち受けポートとIPアドレスの組み合わせ)に対し、接続後にはTCP応答データのソケット・アドレスとして、接続してきたクライアントのIPアドレスとポート番号が付加されるからである。そのソケット・アドレスはsocket.getRemoteSocketAddress()で取得できる。

run()は接続したクライアントごとに動作するスレッドの実行の部分である。始めにソケットとのデータ通信用に二つのストリームを用意する。ストリームがBufferedReaderあるいはBufferedWriterでラップされているのは、そうすることが効率化のため推奨されている、またInputStreamReaderおよびOutputStreamWriterが使われているのは我々のように非英語圏での文字コード変換ができるからである。つぎにクライアントからの1行分のメッセージ受信を待ち、受信されたらそれに簡単なヘッダをつけてクライアントに送り返すことを繰り返す。クライアントが”end”というメッセージを送ってきたら、これを終了要求とみなし、確認応答の意味でクライアントに送り返す。しかる後、ソケットとストリームを解放してこのスレッドは終了する。

#### 4.2.1.2 ストリームと文字エンコーディング

New java.io.BufferedWriter(new java.io.OutputStreamWriter(server\_socket.getOutputStream(), "Windows-31J"));あるいは new java.io.BufferedReader(new java.io.InputStreamReader(server\_socket.getInputStream(), "Windows-31J"));というオブジェクトに関して説明すると、出力ストリームに関しては:

1. socket.getOutputStream というメソッドはバイト単位での出力ストリームを取得している。
2. OutputStreamWriter は文字ストリームからバイトストリームへの橋渡しの役目を持つ。コンストラクタの OutputStreamWriter(OutputStream out, String charsetName)は指定された文字コードを扱う OutputStreamWriter を生成する。ここでは文字ストリームを一般的によく使われている Windows-31J という文字エンコーディングによってバイト・ストリームに変換している。つまり TCP で交換されるデータは Java の文字コード(ユニコード)ではなくて、2 バイト形式の Windows-31J コードの形でのバイト・データにしている。
3. 最大限に効率化するには、コンバータを頻繁に呼び出さないようにするために BufferedWriter の内部に OutputStreamWriter をラップする。BufferedWriter は文字をバッファリングすることによって、文字、配列、または文字列を効率良く文字型出力ストリームに書き込む。BufferedWriter で注意しなければならないのは、newLine というメソッドで、これはシステムにあわせた改行情報を書き込む。

入力ストリームも同じであるので、説明は省略する。

文字セット変換に関しては、[「文字セット変換」](#)の節で更に説明する。

### 4.3節 クライアントのプログラム

以下は一般的なクライアント側のプログラムである。これも例外処理を除けばきわめて簡単なコードである。

```
package TCPIP_Programming;

import java.net.Socket;

/**
 * **** SimpleClient クラス ****
 * このクラスは、前述の SimpleServerThread に対応したクライアントのクラスで、
 * ポート番号 1000 でローカルホストに接続し、コンソールから入力されたストリングを
 * サーバに送出し、サーバからのメッセージはそのままコンソールに出力する。
 * * この型は VisualAge で作成されました。
 */

public class SimpleClient {

    /**
     * main() でこのクラスの総てを記述する。
     * 詳細は各ラインに記述したコメントを参照のこと。
     * * このメソッドは VisualAge で作成されました。
     */

    public static void main(String args[]) {
        Socket client_socket; //サーバと接続したソケット
        java.io.BufferedReader downward_stream; //下りはBufferedReaderを使う
        java.io.PrintStream upward_stream; //上りはBufferedWriterを使用

        try{ //自分のホストと接続開始
            client_socket = new Socket("localhost", 1000);
        }
    }
}
```

```

//指定したホストが存在しないときの例外
catch (java.net.UnknownHostException unknown){
    System.err.println("Failed to establish IP connection with the host.");
    unknown.printStackTrace();
    System.exit(-1);
    return;
} //指定したホストが立ち上がっていないか死んでいるときの例外
catch (java.net.SocketException se){
    System.err.println("Failed to establish Socket connection with the host.");
    se.printStackTrace();
    System.exit(-1);
    return;
} //IOエラーの処理
catch (java.io.IOException e){
    e.printStackTrace();
    System.exit(-1);
    return;
}

String data; //ワーキング用ストリング変数
//コンソール入力の構築
java.io.BufferedReader console_in = new java.io.BufferedReader(new
java.io.InputStreamReader(System.in));

do{ //ここからはループ
    try {
        //下りと下りのストリームの構築 (毎回生成したほうが回線障害に強い)
        downward_stream = new java.io.BufferedReader(new
java.io.InputStreamReader(client_socket.getInputStream(), "Windows-31J"));
        upward_stream = new java.io.PrintStream(client_socket.getOutputStream(), true,
"Windows-31J");

        //コンソールから1行読み込み
        data = console_in.readLine();
        //サーバにこれを送信
        upward_stream.print(data);
        upward_stream.printf(" ... Time: %tTS \n", java.util.Calendar.getInstance());
        //サーバからの戻りを受信
        long t0 = System.currentTimeMillis() + 15000;
        //15秒間の監視窓を設定
        while ((System.currentTimeMillis() < t0) && (downward_stream.ready() == false)){
        }
        if (System.currentTimeMillis() >= t0){
            data = null;
        }else{
            data = downward_stream.readLine();
        }
        //サーバがソケットを閉じるかタイムアウトだとnullが帰ってくる
        if (data == null){data = "end: Server disconnected!";}
        //コンソールに出力
        System.out.println(data);
    } //サーバが停止したか勝手にTCP切断したときの例外
    catch (java.net.SocketException se){
        System.err.println("Server disconnected.");
        se.printStackTrace();
    }
}

```

```

        System.exit(-1);
        return;
    } //IOエラー例外の処理
    catch (java.io.IOException e){
        e.printStackTrace();
        return;
    }
}while (data.startsWith("end") == false);
//サーバから"end"が帰ってきたらループを脱して
try{ //クリーンアップ処理
    console_in.close();
    downward_stream.close();
    upward_stream.close();
} //IOエラーの処理
catch (java.io.IOException e){
    e.printStackTrace();
    System.exit(-1);
    return;
}
} //終了
}

```

このクラスは `main()` のメソッドだけからなる簡単なクラスである。例外処理がたくさん入っているが、それを外すとすごく簡単なプログラムである。始めにサーバのホスト名と受付ポート番号を指定してソケットを作成する。ソケットは直ちにサーバとのコネクションの確立を試み、成功すればそのソケットを返す。続いてコンソール、ソケット間のデータ交換用ストリームを用意する。そのあとはコンソール入力をサーバに送信して、エコーバックされた受信データをコンソールに出力することの繰り返しになる。コンソールから "end" が入力されると、これを終了要求としてサーバに送り、同じメッセージがサーバから帰ってくればサーバからの確認応答とみなしてソケットとストリームを解放して終了する。サーバに送信後にサーバからのデータ受信に 15 秒間のタイマを設定しているが、これは「[例外処理の確認](#)」の節で説明する。

#### 4.3.1.1 テキスト行ベースの通信における行終端

またこのプログラムでは、サーバが改行コードが最後についたメッセージを送信することを想定している。`java.io.BufferedReader.readLine()` は 1 行の終端は、改行 ("n") か、復帰 ("r)、または復行とそれに続く改行のいずれかで認識されている。`SimpleServerThread` ではクライアントから送られた復帰と改行コード付きのメッセージの頭に新たなメッセージを追加して返送しているので問題はないが、他のサーバでは復帰と改行の順番が違っていたり、そのあとで余分な空白が入っていたりしないでもない。従って読み出しのストリームはループの中で毎回新しいものを用意するようにしている。そのような問題を避けるには文字配列に読み出すことも考えられる。

#### 4.3.1.2 ローカル・ホスト(Localhost)

ここでテストなどで良く使われるローカル・ホスト(Localhost)について説明する。"localhost" はループバック・アドレスを意味する文字列で、これは DOS の TCP/IP プロトコル・スタックのレベルで認識される。ループバック・アドレスは「[IP のアドレス](#)」の節で説明したように、IPv4 では 127.0.0.0/8 (通常は 127.0.0.1)、IPv6 では 0:0:0:0:0:0:0:1 である。このアドレスはネットワーク・インターフェイスの出口で折り返す(その IP データグラムはネットワークには出力されない)ことを意味し、他のホストとネットワークで接続することなく自分が開発したネッ

トワーク・アプリケーションをテストできる目的で用意されている。"127.0.0.1"などの文字列を使うことも可能であるが、IPv4 と IPv6 のいずれでも使えることから"localhost"の使用をお勧めする。

ここではサーバの IP アドレスとして localhost が指定されているが、これは実験用で本来はそのサーバの IP アドレスが使われることになる。即ちこのサーバは他のクライアントからの接続も無論受付可能である。

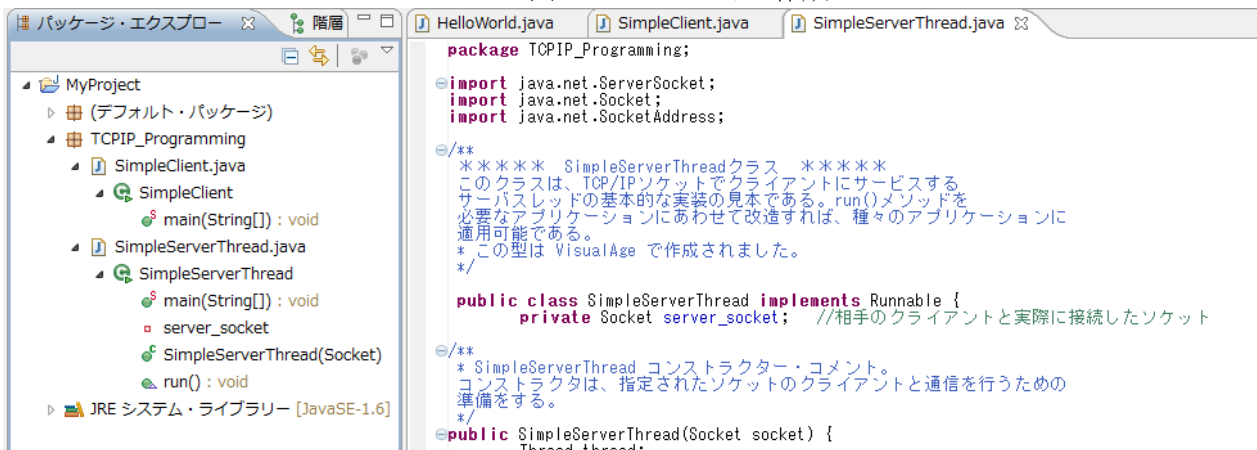
## 4.4節 Eclipse による確認

サーバとクライアントのプログラムを同時に自分の PC 上で走らせて、その動作を理解しよう。

### 4.4.1 プログラムの作成

1. 「ファイル」→「新規」→「プロジェクト」→「Java プロジェクト」→プロジェクト名入力して、次のように TCPIP\_Programming というパッケージを作る。

図 4-2: プロジェクトの作成

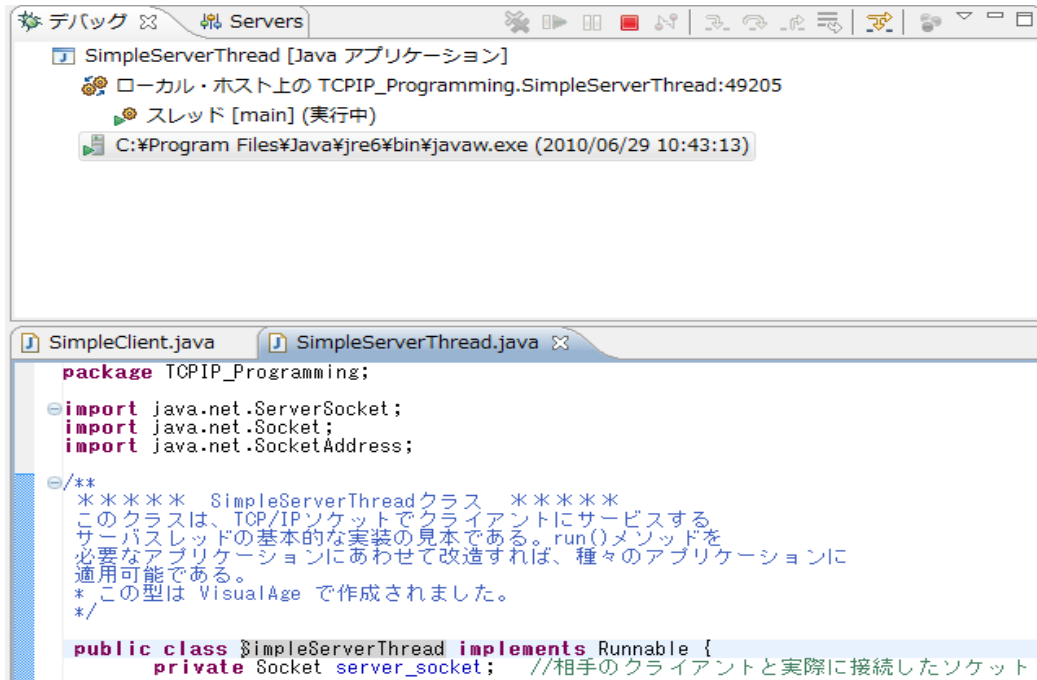


2. このパッケージに 2.5.2 節で示した SimpleServerThread クラスを追加する
3. 同じくこのパッケージに 2.5.3 節で示した SimpleClient クラスを追加する

### 4.4.2 サーバとクライアントをデバッガ上で走らせる

1. デバッガの開始  
「ウインドウ」→「パースペクティブを開く」→「デバッグ」を実行するか、スクリーン右上のパースペクティブ選択の「デバッグ」をクリックするかして、デバッグのパースペクティブを開く。
2. エディタ・エリア上で SimpleServerThread を開く。
3. エディタ・エリア上の SimpleServerThread を右クリックして、「デバッグ」→「Java アプリケーション」を選択すると、下図のようにこのプログラムが実行される（インスタンス化と main() の実行）。

図 4-3: サーバの開始



4. 同じように SimpleClient も開始させる。下図からわかるように SimpleClient が起動して、サーバとの間で TCP 接続が確立された為に、SimpleServerThread のなかに新しくスレッド [Thread-0] が発生している。

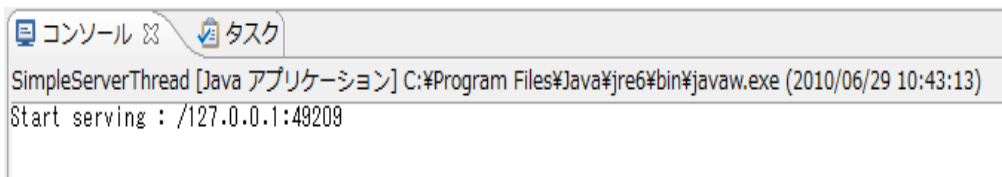
図 4-4: クライアントの開始



コンソールには下図のようにサーバがクライアント(ここではループバック・アドレス)と接続したことを報告している。

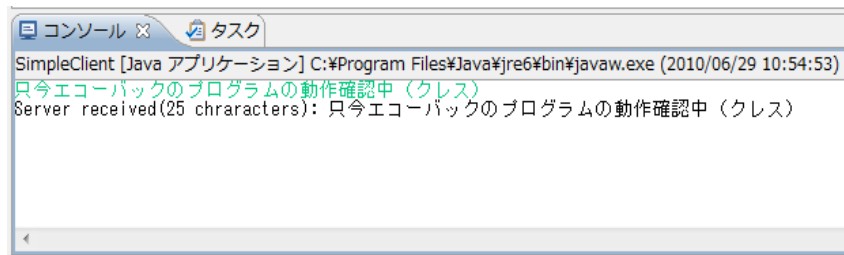
5. クライアントとサーバ間のエコーバック通信の確認

図 4-5: クライアントとの接続



下図のようにクライアントのコンソールを開いて適当な文字列を1行送信(改行)してみる。サーバ側から受信文字数つきで送り返された同じ文字列がコンソールに表示されることを確認する

図 4-6: クライアントのコンソール



#### 6. クライアントを終了させる

クライアントが”end”という文字列をサーバに送信したときは、サーバはこれを終了と見なしソケットを開放(つまり TCP 接続を終了)させる。またクライアントは”end”が帰ってきたことで自分を終了させている。ローカル・ホスト上の SimpleClient の接続が切断され、終了しているだけでなく、SimpleServerThread にあったスレッド[Thread-0]も消滅していることに注意されたい。サーバ側のスレッドをきちんと終了させないとスレッドが沢山残ってしまい、システムに悪影響を与えるので注意すること。

### 4.4.3 複数のクライアントを立ち上げる

#### 1. クライアントを複数立ち上げて、クライアント間の混乱が起きないことを確認する

「選択プロセスの出力の表示」をさせて各クライアントのコンソールからデータを送信しても別のクライアントにサーバが送信してしまわないことを確認する。これはクライアントの Socket 接続が異なったポートでなされていて、きちんと接続の区別が出来ているからである。

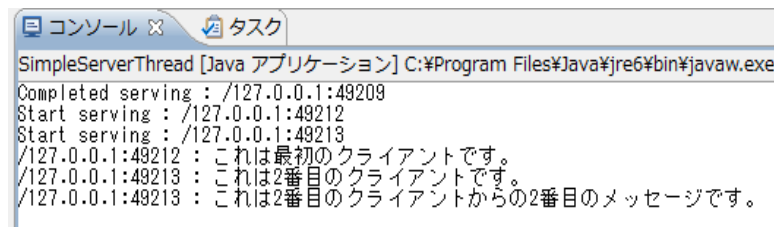
図 4-7: 複数のクライアント対応





同じ IP アドレスであっても、最初のクライアントとはポート 49212 で、2 番目のクライアントからはポート 49213 で接続がなされていることが、サーバのコンソール出力でわかる。

図 4-8: 複数のクライアントとの接続



## 2. 互いの PC を使って動作を確認する

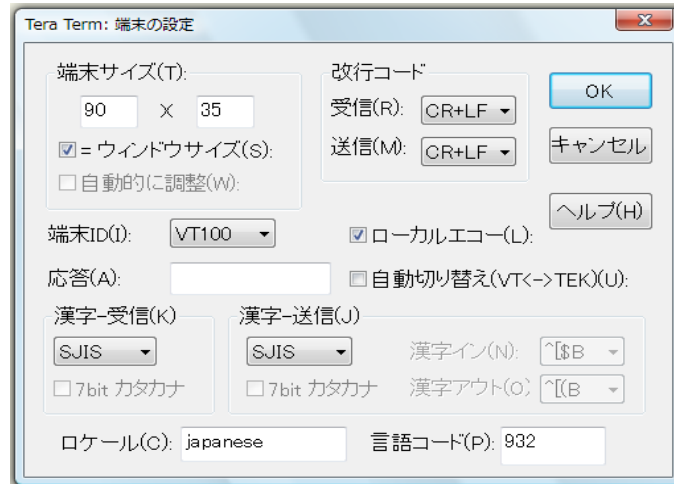
1. 自分のサーバが送り返すテキストに適当なサーバ識別用文字列(例えば”サーバ山本:”など)を付加するようサーバのコードを変更する
2. クライアントのコードを変更して、”localhost”の代わりに接続する他人のサーバの IP アドレスを指定する。例えば相手の PC の IP アドレスが 192.168.153.49 だったとしたら、`current_socket = new Socket("192.168.153.49", 1000);`のように変更すればよい。自分の PC の IP アドレスを知るには、アクセサリにあるコマンド・プロンプトを開始して、”`ipconfig /all`”(Enter)を入力し、IPv4 アドレスを確認する
3. この状態である同僚のサーバを他の人が同時にクライアントとしてアクセスしてどうなるか確認する

## 4.5節 Tera Term による確認

Telnet のプログラムはクライアントのソフトウェアをもう少し高度化したものである。サーブレットの開発時点で、問題をネットワーク・レベルで調査するのによく使われるので、十分マスターすることが必要である。Telnet は IP ネットワークにおいて、遠隔地(リモート)にあるサーバを端末から操作できるようにする仮想端末ソフトウェア(端末エミュレータ)、またはそれを可能にするプロトコルのことを指し、RFC 854 で規定されている歴史的なアプリケーションのひとつである。Tera Term(テラターム)は、Windows 向けのオープンな Telnet プログラムで、広く利用されている。

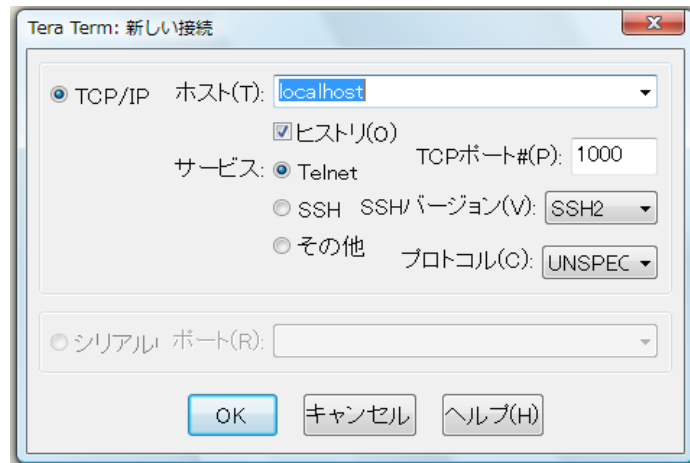
1. [ダウンロードのページ](#)から実行ファイル(例えば teraterm-4.66.exe)をダウンロードしてインストールする
2. このプログラムを実行して、「設定」→「端末」で端末を次のように設定する

図 4-9: Tera Term の端末設定



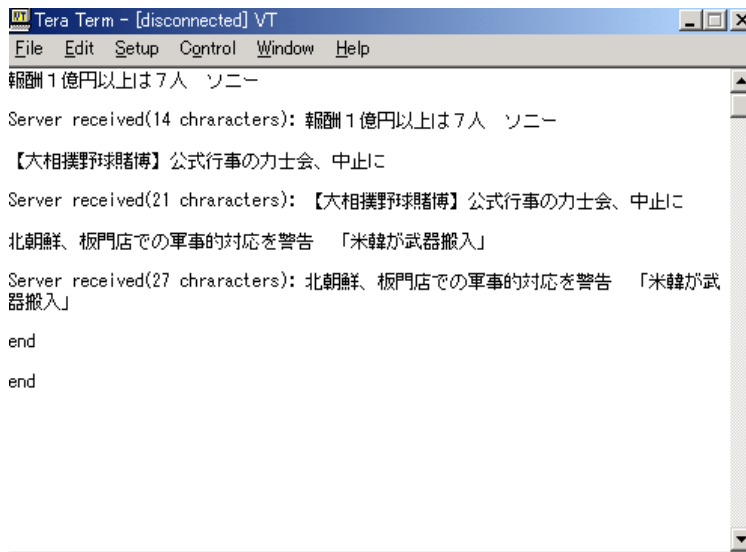
3. 「設定」→「TCP/IP の設定」で、「Telnet」はチェック、「自動的にウインドウを閉じる」はチェックを外す。また「ポート」は 1000 にする。
4. 「設定」→「設定の保存」でこの設定を保存する。
5. 「ファイル」→「新しい接続」で次のように接続する。

図 4-10: Tera Term による TCP 接続



6. 接続が取れたら一行単位でサーバと通信をする。
7. 接続の終了は"end"を送信するか、「ファイル」→「接続断」による強制的な接続を行う。強制的な接続ではサーバ側では null が受かるので、終了を知ることができる。ここが Eclipse のデバッグ上での強制的なクライアントの終了の場合と違うところで、Tera Term はきちんとした TCP 接続解放手順をとっている。

図 4-11: 交信例



```
Tera Term - [disconnected] VT
File Edit Setup Control Window Help
報酬1億円以上は7人 ソニー
Server received(14 characters): 報酬1億円以上は7人 ソニー
【大相撲野球賭博】公式行事の力士会、中止に
Server received(21 characters): 【大相撲野球賭博】公式行事の力士会、中止に
北朝鮮、板門店での軍事的対応を警告 「米韓が武器搬入」
Server received(27 characters): 北朝鮮、板門店での軍事的対応を警告 「米韓が武器搬入」
end
end
```

## 4.6節 Java.net における TCP の接続と開放

### 4.6.1 TCP の接続

TCP では(ローカル IP アドレス、ローカル・ポート番号、外部またはリモート IP アドレス、外部またはリモート・ポート番号)の組で接続一意的に識別される。java.net.Socket での TCP 接続は以下のようになされている。

- コンストラクタによる接続
  - Socket(InetAddress address, int port): ストリーム・ソケットを作成し、指定された IP アドレスの指定されたポート番号に接続
  - Socket(String host, int port): ストリーム・ソケットを作成し、指定されたホスト上の指定されたポート番号に接続
  - Socket(InetAddress address, int port, InetAddress localAddr, int localPort): ソケットを作成し、指定されたリモート・ポート上の指定されたリモート・アドレスに接続
  - Socket(String host, int port, InetAddress localAddr, int localPort): ソケットを作成し、指定されたリモート・ポート上の指定されたリモート・ホストに接続
- connect による方法
  - 接続されていないソケットを作成した後で connect(SocketAddress endpoint): このソケットをサーバーに接続
  - 接続されていないソケットを作成した後で connect(SocketAddress endpoint, int timeout): 指定されたタイムアウト値を使って、このソケットをサーバーに接続

Java ではストリーム・ソケット単なるソケットと区別しているが、ストリーム・ソケットは TCP ベースのソケットのことを呼び、ソケットとは UDP (Java ではデータグラム・ソケットと称している)も含めた基となるソケットのことを称している。しかしこれは混乱を起ししやすい。TCP も UDP もリモート・ポートだけでなくローカル・ポートを指定す

ることは可能であるが、UDP には接続という概念が存在しないのである。したがって javadoc の記述はおかしい。Java の抽象化と現実 (TCP 及び UDP) との間がしっくりいっていない例であろう。

connect メソッドによる接続は、これまでの SimpleClient.java のコードを以下のように変更して確認できる。

```
import java.net.InetSocketAddress;
.....
try{
    //自分のホストと接続開始
    client_socket = new Socket();
    client_socket.connect(new InetSocketAddress("localhost", 1000));
```

この場合は Javadoc のいう「接続されていないソケット」を作成したあとで、InetSocketAddress のオブジェクトをパラメータにしてサーバと接続している。

ところで Socket.bind(SocketAddress bindpoint) というメソッドは何であろうか？ ために次のようなコードを作ってデバッグ・モードでこの行を実行してみよう。

```
client_socket.bind(new InetSocketAddress("localhost", 1000));
```

そうするとサーバが動作中のときは、アドレスが既に使われているというエラーが返される。したがって、これは TCP プロトコルのなかのテーブルにこのアドレスを追加するメソッドであるかに見える。つまりこれは特にサーバなどで複数のネットワーク・インターフェイスを実装している場合に自分のホストのどのネットワーク・インターフェイス (ローカル・アドレスとポート) を使うかを指定する為のものである。bind というメソッドは ServerSocket にも存在しており、むしろサーバでよく使用される。つまり

```
listen_socket = new ServerSocket(1000);
```

の代わりに、

```
import java.net.InetSocketAddress;
.....
listen_socket = new ServerSocket();
listen_socket.bind(new InetSocketAddress("localhost", 1000));
```

を使ってでもサーバが接続待ちの状態になる。

## 4.6.2 TCP の開放

TCP 接続の開放は 2.3 節で説明したように、サーバ側からもクライアント側からの開始できる。時にはほぼ同時に双方が開放を始めることもあり得る。TCP プロトコルではいろんな状況においても確実な開放ができるよう設計されている。

- クライアント側での開放
  - Socket.close(): このソケットを閉じる。現在このソケットの入出力操作でブロックされているすべてのスレッドが SocketException をスローする。また、このソケットの InputStream と OutputStream もクローズされる。したがって双方のストリームとも flush メソッド等ですべて吐き出されている必要がある
- サーバ側での開放
  - ServerSocket.close(): このソケットを閉じる。accept() で現在待機中であるスレッドはすべて、SocketException をスローする。また、このソケットに関連するチャンネルが存在する場合は、そのチャンネルも閉じられる。

### 4.6.3 setReuseAddress メソッドについて

ここで注意すべきことは、`setReuseAddress(boolean on)`というメソッドである。これはソケットの `SO_REUSEADDR` オプションを有効あるいは無効とするものである。これは `java.net.Socket`、及び `java.net.ServerSocket` で存在するメソッドであるが、後で紹介する MINA の `SocketAcceptor` でも存在している。[TCPコネクションの開放の節](#)で説明したように、TCP 接続の開放に際しては、解放を開始したホスト側で  $2 * \text{MSL}$  タイムアウト待ち状態を持っている。つまり相手の FIN に対する ACK を返したら直ちに CLOSED に移行はしないで一定時間 TCB を保持しているが、これは返した ACK が相手に万一届かないと、ホスト B はタイムアウトでまた FIN を送ってくることになり、これに対応できるようにしばらくこの接続を存続させるためである。MSL はインターネット上での最大セグメント生存時間(Maximum Segment Lifetime)で通常 2 分間に設定されている。ただしこれは現在のインターネットの環境ではかなり長い時間であり、実装している OS によってもっと短時間に設定されていることもある。

Javadoc では、このタイム稼働中においても `setReuseAddress(true)`により、同じ `SocketAddress` で再バインドが可能ないようにしていると説明している。しかし本来は例えば同じクライアントから再接続を要求してきた場合でも、別のクライアント側のポート番号が設定されることで、接続管理テーブルは増えるものの、待ち状態にはならないはずである。従ってこの設定は、デバッグなどの場合にサーバを再立ちあげることなく前回と同じポート番号が使われるような場合を考えているのであろうか？

ちなみに、次のようなサーバを立ち上げて、Tera Term で 5 回これに接続してみる：

```
package TCPIP_Programming;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketAddress;
import java.net.InetSocketAddress;

public class ReuseAddressTestServer {

    /**
     * @param args
     */
    public static void main(String[] args) {
        for (int i = 0; i < 5; i++) {
            try {
                ServerSocket ss = new ServerSocket();
                ss.setReuseAddress(true);
                ss.bind(new InetSocketAddress("localhost", 1000));
                Socket s = ss.accept();
                SocketAddress ca = s.getRemoteSocketAddress();
                System.out.printf("Connection Established: #%d", i);
                System.out.printf(", Time: %tTS \n",
java.util.Calendar.getInstance());
                System.out.println("Start serving : " + ca);
                ss.close();
            } catch (Exception ex) {
                ex.printStackTrace();
                return;
            }
        }
    }
}
```

このサーバは自分の IP アドレスが"localhost"で、自分の待ち受けポート番号が 1000 を自分のソケットにバインドし、クライアントからの接続が発生したら自分でその接続の開放を起動することを 5 回繰り返している。自分で開放を起動したので、これらのソケットは 2\*MSL タイムアウト(4 分間) 待ち状態に入っているはずである。

Tera Term で接続・開放を繰り返すと、次のようなコンソール出力となる:

```
Connection Established: #0, Time: 15:21:58S
Start serving : /127.0.0.1:3138
Connection Established: #1, Time: 15:22:09S
Start serving : /127.0.0.1:3139
Connection Established: #2, Time: 15:22:19S
Start serving : /127.0.0.1:3143
Connection Established: #3, Time: 15:22:29S
Start serving : /127.0.0.1:3144
Connection Established: #4, Time: 15:22:40S
Start serving : /127.0.0.1:3145
```

しかしながら、ここで `setReuseAddress(false)`としても同じような結果が得られ、バインドの例外が発生しない。クライアントのポート番号はダイナミックに変えられているので、支障は無いのである。

この状況は `ss.accept()`;という行をコメント・アウトしても変化しない。したがって `setReuseAddress` というメソッドの用途は、クライアントのポートも固定されるような特別な状態に限られよう。もうひとつの用途は、サーバ側で同じ(IP アドレス、ポート番号)を複数のサーバ・プロセスが使う場合である。あるプロセスが生成されたソケットを別のプロセスに渡して、そのプロセスがそのクライアントとの通信を継続するというシナリオは考えられるかもしれない。この場合は、`setReuseAddress(false)`としないと、バインドでエラーが生じる。しかしこれも特別な例であり、2つのプロセスが同じ(IP アドレス、ポート番号)を使っても、クライアントがアクセスしたときにどちらのプロセスが受けるかは保証されない。それならばソケットのオブジェクトそのものをもうひとつのプロセスに渡せば良いだけである。

## 4.7節 例外処理の確認

ここで再びこのサーバ/クライアント・システムの終了について説明し、このコードがネットワークのいろんな異常事態に対応していることを示す。各自これを確認されたい。

### 4.7.1 ネットワーク障害

多分下位ネットワーク障害で一番多いのは回線断の事態であろう。基本的に例外が発生したらスレッドの場合はそのスレッドを終了させる配慮が必要である。

1. サーバが立ち上がっていないのにクライアントがアクセスしたとき:  
クライアント側ではタイムアウトで `SocketException` が発生する。"Failed to establish Socket connection with the host."とクライアントのコンソールに表示される。
2. セッションがはらわれている(書き込み時)のにサーバが勝手にソケットをクローズしたとき:  
クライアント側では送信の際に同様に `SocketException` が発生する。"Server disconnected."が表示される。
3. セッションがはらわれている(エコーバック時)のにサーバが勝手にソケットをクローズしたとき:

クライアント側では送信の際に同様に `SocketException` が発生する。”Server disconnected.”が表示される。

4. セッションがはられているのに(読み込み時)クライアントが勝手にソケットをクローズしたとき:  
サーバ側では `SocketException` 例外が発生してこれを検出し、スレッドが終了する。
5. セッションがはられているのに(書き込み時)クライアントが勝手にソケットをクローズしたとき:  
サーバ側では `SocketException` 例外が発生してこれを検出し、スレッドが終了する。

もう一度サーバ/クライアント双方のプログラムリストを見ると、これらの状況のすべてに対応ができていることが判るであろう。

なお Eclipse で動作中のスレッドを終了させると、TCP 接続はきちんとした接続開放の手順を踏まないで TCP をリセットしてしまうので、相手側には `null` が送られない。これは下位ネットワーク上の障害をシミュレートするのに都合が良い。Tera Term のようなプログラムの場合は正しい TCP 接続開放がされていることは前節で説明した。

ネットワーク障害で一番多く遭遇するのは回線断であろう。TCP の再送制御機能でカバーできない長時間の回線断は、コネクションの強制切断が行われることは以前説明した。そのような事態は2つの PC を使ってその間のイーサネットを外すことで再現できよう。たとえばエコーバック・サーバの受信から返送までに間に

```
try {
    Thread.sleep(10000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

と10秒間のタイマーを置いて、その間にイーサネットを外すことで前記項目の5番目を試すことができる。そのときのコンソール出力されたエラー・メッセージは次のようになる:

```
java.net.SocketException: Connection reset by peer: socket write error
```

したがって、前記項目の5番目を確認される。

**クライアントのプログラムで15秒間のタイマ監視が付いているのは、このような障害がおきたときは、クライアント側ではいつまでたってもサーバからの応答待ちとなってしまうユーザ側に見ればプログラムが止まってしまうという問題があるからである。**このような監視を入れないプログラムはサーバが必ず返信してくることを想定している。起きる可能性は極めて少ないものの、残念ながら実際の運用の段階では得てしてそのようなことが起きるものである。これに対処する為にはクライアント側ではタイマー監視が必要である。

タイマー監視はサーバ側でも必要があることがある。それはクライアントがたとえば PC を入れたまま帰社したなど TCP 接続をしたまま長時間放置したときで、そのような事態はクライアントの数が大きいアプリケーションではスレッド数が増大するとかオーバーヘッドが大きくなるとかの問題になろう。

タイマー監視は時刻を使うものから新たなタイマーのスレッドを使うものまでである。ここでは一番簡単な時刻を使うものが採用されている。これは `SimpleClient` のデータ送信からサーバからの受信の箇所に15秒間のタイマーを挿入したもので、15秒経過してもサーバからの戻りがない場合は、サーバが終了したと判断している。`while` のループは15秒間経過か `downward_stream` からデータ(`null`を含めて)が読み出し可能になるまで抜けない。

```
//サーバからの戻りを受信
long t0 = System.currentTimeMillis() + 15000;
```

```

//15 秒間の監視窓を設定
while ((System.currentTimeMillis() <
t0)&&(downward_stream.ready() == false)){
}
if (System.currentTimeMillis() >= t0){
    data = null;
}else{
    data = downward_stream.readLine();
}
//サーバがソケットを閉じるかタイムアウトだと null が帰ってくる

```

#### 4.7.2 キープ・アライブ

互いが通信できる状態にあることを確認することは一般的にはキープ・アライブ(keep alive)と呼ばれている。

TCP 層ではホストが RFC 2018 にもとづく検査セグメント(KeepAlive プローブ)を定期的にピアに送ることが出来る。この場合ピアは以下の4つの状態のいずれかとなる:

- ピアの TCP は通常どおり応答する。ホストはキープ・アライブ・タイマをセットし、その時点でトラフィックがあれば、キープ・アライブ・タイマをリセットする。
- ピアの TCP 無応答。その後 10 回リトライしても応答なし。ホストはピアがダウンしたと判断し、接続を終了する。
- ピアの TCP は応答するが、応答は RESET。ホストは接続を閉じる。
- ピアは稼動しているが、ホストから到達不可能。ホストはピアがダウンしたと判断し、接続を閉じる。

ただしこのオプションは Java では `java.net.SocketOptions` で用意されている。SO\_KEEPALIVE というオプション・フィールドがそうである。TCP ソケットに KeepAlive オプションが設定されていて、ソケットを介してどちらの方向にもデータが 2 時間(注: 実際の値は実装による)の間交換されていない場合、TCP は自動的に KeepAlive プローブをピアへ送信する。

アプリケーション層で良く使われるのは、双方で定期的に自分が送信した特定のメッセージ(例えば"Hello")が送り返されてくるのを確認する手段である。そのようなコードを双方に入れておくことも有用である。



## 第5章 Java NIO

2002年のJava 1.4からjava.nio(俗称Java NIO、「エン・アイ・オー」、「ナイオ」、あるいは「ニーオ」とか発音されている。new I/Oの略語)というパッケージ群が新しく追加されている。これにはソケットAPIの大幅なアップデートが含まれている。次節の[非ブロッキング・ソケットI/O](#)もそれに含まれる。

[Sun\(現在はOracle\)の説明](#)によれば、java.nioのAPIの特徴は次のようである:

- プリミティブな形式(short, int, double etc)たちのデータの為のバッファたち
- 文字セットのエンコーダとデコーダ
- Perlスタイルの正規表現に基づいたパタン・マッチング機能
- 新しいプリミティブなI/Oの抽象化であるチャンネル(Channels)
- ロックとメモリ・マッピングに対応するファイル・インターフェイス
- 拡張可能性のあるサーバ作成の為の多重化された非ブロックのI/O機能

パッケージは次のような構成になっている:

- java.nio: NIOのAPIたちにわたって使われているバッファたち(ShortBuffer, IntBuffer, DoubleBuffer etc.)
- java.nio.channels: チャンネルとセレクタ。
- java.nio.charset: 文字エンコーディング
- java.nio.channels.spi: チャンネルたちのためのサービス・プロバイダのクラスたち
- java.nio.charset.spi: 文字セットたちのためのサービス・プロバイダ・クラスたち
- java.util.regex: 正規表現で指定されたパターンに対して文字シーケンスをマッチングするためのクラス
- java.lang.CharSequence: これはインターフェイスで、java.util.regexパッケージのメソッドたちに引数として渡すことができるオブジェクトによって実装される。このインターフェイスを実装しているのはString、StringBuffer、及びjava.nio.CharBufferのクラスたちである

その中でTCP/IPネットワークのアプリケーションで多用されるのはjava.nio.ByteBuffer、java.nio.charset.CharsetDecoder(及びEncoder)、及びjava.nio.channels.ServerSocketChannelであろう。

### 5.1節 ByteBuffer

ネットワークではデータの交換はバイト(ネットワークの用語ではオクテット(octet)という)単位でなされている。しかしJava 1.4以前はバイト・データを取り扱うjava.io.InputStream(あるいはOutputStream)を基に、これにjava.io.InputStreamReader(あるいはOutputStreamWriter)で文字変換し、さらにjava.io.BufferedReader(あるいはBufferedWriter)でラップする方法が一般に使われていた。Java.nioでは、ブロックとして扱われ、その為のJava.nio.ByteBufferはストリームに比べバイト単位でより多様な処理が可能なものとなっているし、高速化

がされている。ただしこのクラスは `java.nio.channels.SocketChannel` で使われているが **java.net.Socket** では直接的には使えない。

### 5.1.1 ByteBuffer のイメージ

このバッファはリング形式ではなくてリニアなバイト列の固定容量のもので、低レベルの処理に適したものといえよう。このバッファは多様な操作が可能ではあるが、その分やや複雑になっており、使いこなすにはきちんとした理解が必要である。バッファのクリア、フィル、読み出し、及び書き込みは明示的に行わねばならない。

このバッファの特徴は:

- バイト・アレー (あるいはメモリファイルのブロック) を、現在の読み書きポジション、最大ポジション、バイト順序などが扱える機能を持ったバッファとして処理できる。
- 特定のデータ・サイズに特定の場所からアクセスできる `putInt()`、`putLong()` などの便利なメソッドが用意されている。
- JVM によって最適化されており、例えば `Buffer.get()` メソッドは文字どおりマシン・コードのレベルでの `move` 命令に変換されている。
- あるファイルのマッピングされたビューが用意されており、例えば特定のポジションでの `putShort()` 呼び出しは文字どおりそのファイルのそのオフセットに書き込まれる。

このバッファの基本的な要素 (上位クラスの `java.nio.Buffer` で定義) は次のようになっている:

マーク(mark) <= ポジション(position) <= リミット(limit) <= 容量(capacity)

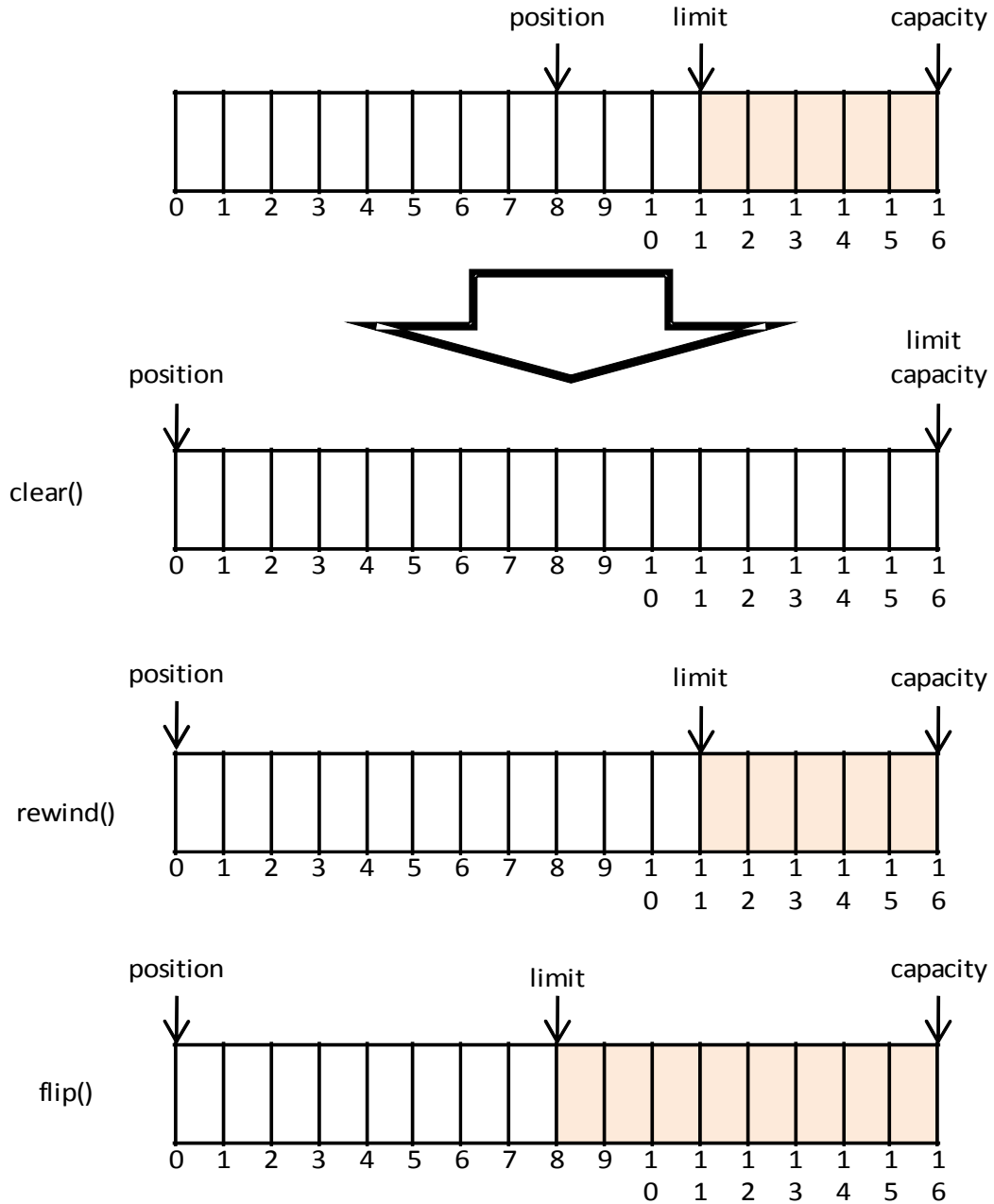
- 容量: これはこのバッファのサイズのこと。あるいはこのバッファの終わりの次のスロットのインデックスともいえる。
- リミット: このバッファがいっぱいのときはリミットは容量と同じ値になる。バイト・データを読み出すときはこのバッファが最後に埋められたバイトの次のスロットのインデックスになる。
- ポジション: このバッファにバイト・データを埋め込んで (書き込んで) いるときは、このポジションはこのバッファに満たされている最後のバイトの次となる。このバッファからデータを取り出して (読み出して) いるときは、このポジションはこのバッファから最後に書きだされたバイトの次になる。即ち次に読み書きするこのバッファの最初の位置からのオフセット (インデックス) である。ポジションの最大値はリミットである。
- マーク: これはオプション的なマーク点で、後で戻りたい場所をセットする。 `mark()` を呼ぶと現在のポイントがマークとなり、 `reset()` を呼ぶとそのマーク位置に戻る。つまり `reset` メソッドを実行したときに戻る位置を指定するインデックスである。定義されていない場合もあるが、定義されている場合は必ずポジション以下の正の値になる。ポジションやリミットの値がマークの値よりも小さい場合、マークは破棄される。

これらの要素のうち容量は不変だが、残りは変化する、あるいは変化させることができる:

- `clear()` は、新しい一連のチャンネル読み込み操作または相対「put」操作のためにバッファを準備する。リミットを容量の値に設定し、ポジションを 0 に設定する。
- `flip()` は、新規チャンネル書き込みシーケンス (相対「get」) 等のためにバッファを準備する。リミットの値を現在ポジションの値に合わせたあと、ポジションの値を 0 にする。
- `rewind()` は、すでにバッファ格納されているデータを再度読み込めるように、バッファを準備する。リミットの値はそのまま、ポジションの値を 0 にする。

これらの操作を 16 バイトの容量を持った例で図示すると下図のようになる:

図 5-1:3 つのメソッドによる要素の位置



ダイレクト・バッファと非ダイレクト・バッファ(ヒープ・バッファ)について:

バイト・バッファには、「ダイレクト」バッファと「非ダイレクト」バッファが用意されている。ダイレクト・バッファは Java のヒープ領域外に割り当てられた固定メモリ領域へのアクセスを可能とするものである。現在はこのバッファのサイズは 2GB になっている。

ダイレクト バイト・バッファの場合、Java 仮想マシンは効率化のために、ネイティブの入出力操作を直接実行しようとする。これは、基本となる OS 固有の I/O 操作を呼び出す際、中間バッファを介さないということである。

ダイレクト バイト・バッファは、このクラスのファクトリ・メソッドの `allocateDirect` を呼び出すと作成される。通常は、こちらのバッファのほうが、非ダイレクト・バッファよりも割り当ておよび解放コストがやや高くなる。ダイレクト・バッファの内容が標準のガベージ・コレクションされたヒープの外部にあるなら、アプリケーションのメモリ領域に対する影響は僅かである。このことから、ダイレクト・バッファには、基本となるシステム固有の入出力操作に従属する、寿命が長く容量の大きいバッファを指定することが勧められる。一般に、ダイレクト・バッファ割り当ては、プログラムの性能を十分に改善できる見込みがある場合にのみ行うべきである。

ダイレクト バイト・バッファは、ファイルの特定の領域をメモリに直接マッピングする方法でも作成できる。Java プラットフォームの実装によっては、JNI(Java Native Interface)を介してネイティブ・コードからダイレクト・バイト・バッファを生成する機能がオプションでサポートされている可能性がある。こうした種類のバッファのインスタンスが、メモリ内のアクセスできない領域を参照した場合、その領域にアクセスしようとしてもバッファのコンテンツは変更されず、アクセス時またはアクセス後に何らかの例外がスローされる。

あるバイト・バッファがダイレクト・バッファ、非ダイレクト・バッファのどちらであるかを判断するには、`isDirect` メソッドを呼び出す。このメソッドを使用すると、性能が重視されるコード内で明示的にバッファ管理が行える。

プリミティブ型の読み書きについて:

このクラスには、すべての Java プリミティブ型 (`boolean` を除く) の値の読み込みと書き込みを行うメソッドが定義されている。プリミティブ値とバイト・シーケンスとの相互変換は、バッファの現在のバイト順序 (8 ビットのうちどちら側が MSB、即ち 10 進数の 128 とするか) に従って行われる。バイト順序を取得および変更するには `order` メソッドを使用する。特定のバイト順序は、`ByteOrder` クラスのインスタンスで表される。バイト・バッファのデフォルト順序は、常に `BIG_ENDIAN`、即ち MSB が左である。これは TCP/IP で標準的に使われているやり方なので、これをいじることはまず無いだろう。

異種バイナリ・データ、すなわち型の異なる値のシーケンスにアクセスできるようにするため、このクラスは、型ごとに一連の絶対および相対 `get/put` メソッドのファミリーを定義している。たとえば、32 ビットの浮動小数点数 (`float` 値) の場合、次のメソッドが定義されている。

- `float getFloat()`
- `float getFloat(int index)`
- `void putFloat(float f)`
- `void putFloat(int index, float f)`

`char`, `short`, `int`, `long`, `double` の各型にも、同様のメソッドが定義されている。絶対 `get/put` メソッドのインデックス・パラメタの単位は、読み込みまたは書き込みの対象となる型ではなく、バイトである。

同種のバイナリ・データ、すなわち同じ型の値のシーケンスにアクセスできるようにするため、このクラスには、指定されたバイト・バッファの「ビュー」を作成するメソッドが定義されている。「ビュー・バッファ」とは、バイト・バッファに連動した内容を持つ、別のバッファのことである。バイト・バッファの内容に変更を加えると、ビュー・バッファにもその内容が反映される。反対に、ビュー・バッファの内容に変更を加えると、バイト・バッファにもその内容が反映される。この 2 つのバッファの位置、リミット、マークの値は、それぞれ独立している。たとえば、`asFloatBuffer` メソッドは、このメソッドの呼び出し元のバイト・バッファに連動した `FloatBuffer` クラスのインスタンスを生成する。`char`, `short`, `int`, `long`, `double` の各型に対しても、同様のビュー作成メソッドが定義されている。

ビュー・バッファには、前述した一連の型固有の `get/put` メソッドに勝る重要な利点が3つある。

1. ビュー・バッファには、バイトではなく、その値の型固有のサイズによってインデックスが付けられる
2. ビュー・バッファは、バッファと配列または同じ型のその他のバッファ間で連続した値のシーケンスをやりとりできる、相対一括 `get/put` メソッドを提供している
3. ビュー・バッファは、補助バイト・バッファがダイレクト・バッファである場合に限りダイレクト・バッファになるという点で、潜在的に効率がよい

ビュー・バッファのバイト順序は、ビューの生成時にバイト・バッファと同じものに固定される。

### 5.1.2 Buffer と ByteBuffer クラスのメソッド

以下は `ByteBuffer` クラスのコンストラクタとメソッド、及び親の `Buffer` 抽象クラスのメソッドの表である。

この `ByteBuffer` クラスでは、非常に豊富なメソッドが用意されているが、バイト・バッファに対する操作は次の6つのカテゴリに区分される。

1. 単一バイト値の読み込みと書き込みを行う絶対および相対 `get/put` メソッド
2. 連続した `byte` シーケンスをこのバッファから配列へと転送する相対一括 `get` メソッド
3. 連続したバイト・シーケンスをバイト配列やその他のバイト・バッファからこのバッファへ転送する相対一括 `put` メソッド
4. その他のプリミティブ型の値の読み込みと書き込みを行い、これらの値とバイト・シーケンスを特定のバイト順序で相互変換する、絶対および相対 `get/put` メソッド
5. その他のプリミティブ型の値を格納するバッファとしてバイト・バッファを表示できる、「ビュー・バッファ」の作成メソッド
6. バイト・バッファの圧縮、複製、スライス用メソッド

表 5-1 : `ByteBuffer` クラスのメソッド

メソッド	
<code>public static ByteBuffer allocateDirect(int capacity)</code>	新しいダイレクト バイト・バッファを割り当てる。新しいバッファのポジションは 0、リミットは容量と同じ値になる。マークは定義されない。補助配列を利用するかどうかは指定されない。 パラメタ: <code>capacity</code> - 新しいバッファの容量 (バイト) 戻り値: 新しい バイト・バッファ 例外: <code>IllegalArgumentException</code> - <code>capacity</code> が負の整数のとき
<code>public static ByteBuffer allocate(int capacity)</code>	新しい バイト・バッファを割り当てる。新しいバッファのポジションは 0、リミットは容量と同じ値になる。マークは定義されない。このバッファは補助配列を利用し、その配列オフセットは 0 になる。 パラメタ: <code>capacity</code> - 新しいバッファの容量 (バイト) 戻り値: 新しい バイト・バッファ 例外: <code>IllegalArgumentException</code> - <code>capacity</code> が負の整数である場合
<code>public static ByteBuffer wrap(byte[] array,</code>	バイト配列をバッファにラップする。新しいバッファは指定されたバイト配列によって補助

<p>int offset, int length)</p>	<p>される。バッファに変更を加えると配列も変更され、配列に変更を加えるとバッファも変更される。新しいバッファの容量は <code>array.length</code>、ポジションは <code>offset</code>、リミットは <code>offset + length</code> となる。マークは定義されない。指定された配列が補助配列となり、その配列オフセットは 0 になる。</p> <p>パラメタ:</p> <p><code>array</code> - 新しいバッファを補助する配列</p> <p><code>offset</code> - 使用するサブ配列のオフセット。<code>array.length</code> 以下の負でない値でなければならない。新しいバッファのポジションは、この値に設定される</p> <p><code>length</code> - 使用するサブ配列の長さ。<code>array.length - offset</code> 以下の負でない値でなければならない。新しいバッファのリミットは、<code>offset + length</code> に設定される</p> <p>戻り値: 新しいバイト・バッファ</p> <p>例外: <code>IndexOutOfBoundsException</code> - <code>offset</code> パラメタと <code>length</code> パラメタの前提条件が満たされていない場合</p>
<p>public static ByteBuffer wrap(byte[] array)</p>	<p>バイト配列をバッファにラップする。</p> <p>新しいバッファは指定されたバイト配列によって補助される。バッファに変更を加えると配列も変更され、配列に変更を加えるとバッファも変更される。新しいバッファの容量とリミットは <code>array.length</code>、ポジションは 0 になる。マークは定義されない。指定された配列が補助配列となり、その配列オフセットは 0 になる。</p> <p>パラメタ:</p> <p><code>array</code> - 現在のバッファを補助する配列</p> <p>戻り値: 新しいバイト・バッファ</p>
<p>public abstract ByteBuffer slice()</p>	<p>このバッファの共有のサブシーケンスを内容とする新しいバイト・バッファを作成する。</p> <p>新しいバッファのコンテンツは、現在のバッファの現在ポジションから始まる。現在のバッファのコンテンツに変更を加えると、その内容が新しいバッファに反映される。新しいバッファのコンテンツに変更を加えると、その内容が現在のバッファに反映される。2 つのバッファのポジション、リミット、マークの値はそれぞれ異なる。</p> <p>新しいバッファのポジションは 0、容量とリミットはこのバッファ内に残っているバイト数になる。マークは定義されない。新しいバッファは、現在のバッファがダイレクト・バッファである場合に限りダイレクト・バッファになる。また、現在のバッファが読み取り専用バッファである場合に限り読み取り専用バッファになる。</p> <p>戻り値: 新しいバイト・バッファ</p>
<p>public abstract ByteBuffer duplicate()</p>	<p>このバッファの内容を共有する新しいバイト・バッファを作成する。</p> <p>新しいバッファのコンテンツは、現在のバッファのコンテンツと同じになる。現在のバッファのコンテンツに変更を加えると、その内容が新しいバッファに反映される。新しいバッファのコンテンツに変更を加えると、その内容が現在のバッファに反映される。2 つのバッファのポジション、リミット、マークの値はそれぞれ異なる。</p> <p>新しいバッファの容量、リミット、ポジション、マークの値は、現在のバッファの対応する値と同じになる。新しいバッファは、現在のバッファがダイレクト・バッファである場合に限りダイレクト・バッファになる。また、現在のバッファが読み取り専用バッファである場合に限り読み取り専用バッファになる。</p> <p>戻り値: 新しいバイト・バッファ</p>
<p>public abstract ByteBuffer asReadOnlyBuffer()</p>	<p>このバッファの内容を共有する新しい読み込み専用バイト・バッファを作成する。</p> <p>新しいバッファのコンテンツは、現在のバッファのコンテンツと同じになる。現在のバッファのコンテンツに変更を加えると、その内容が新しいバッファに反映される。しかし、新しいバッファ自体は読み取り専用であり、その共有コンテンツを変更することはできない。2 つのバッファのポジション、リミット、マークの値はそれぞれ異なる。</p>

	<p>新しいバッファの容量、リミット、ポジション、マークの値は、現在のバッファの対応する値と同じになる。</p> <p>現在のバッファが読み取り専用の場合、このメソッドの動作は <code>duplicate</code> メソッドとまったく同じになる。</p> <p>戻り値: 新しい読み込み専用バイト・バッファ</p>
<code>public final int capacity()</code>	<p>このバッファの容量を返す。</p> <p>戻り値: このバッファの容量</p>
<code>public final int position()</code>	<p>このバッファのポジションを返す。</p> <p>戻り値: このバッファのポジション</p>
<code>public final Buffer position(int newPosition)</code>	<p>このバッファのポジションを設定する。新しいポジションの値よりもマークの値のほうが大きい場合、マークの定義は破棄される。</p> <p>パラメタ: <b>newPosition</b> - 新しいポジションの値は、現在のリミット以下の負でない値でなければならない</p> <p>戻り値: 現在のバッファ</p> <p>例外: <code>IllegalArgumentException</code> - <b>newPosition</b> の前提条件が満たされていない場合</p>
<code>public final int limit()</code>	<p>このバッファのリミットを返す。</p> <p>戻り値: このバッファのリミット</p>
<code>public final Buffer limit(int newLimit)</code>	<p>このバッファのリミットを設定する。ポジションの値が新しいリミットより大きい場合、リミットと同じ値に変更される。マークの値が新しいリミットより大きい場合、マークの定義は破棄される。</p> <p>パラメタ: <b>newLimit</b> - 新しいリミット値は、このバッファの容量以下の負でない値でなければならない</p> <p>戻り値: 現在のバッファ</p> <p>例外: <code>IllegalArgumentException</code> - <b>newLimit</b> の前提条件が満たされていない場合</p>
<code>public final Buffer mark()</code>	<p>このバッファの現在ポジションにマークを設定する。</p> <p>戻り値: 現在のバッファ</p>
<code>public final Buffer reset()</code>	<p>バッファのポジションを以前にマークしたポジションに戻す。</p> <p>このメソッドを呼び出しても、マークの値は変更されない。マークが破棄されることもない。</p> <p>戻り値: 現在のバッファ</p> <p>例外: <code>InvalidMarkException</code> - マークが設定されていない場合</p>
<code>public final Buffer clear()</code>	<p>このバッファをクリアする。バッファのポジションは <code>0</code>、リミットは容量の値に設定される。マークは破棄される。</p> <p>一連のチャンネル読み込み操作または「<code>put</code>」操作を使用してこのバッファにデータを格納</p>

	<p>する前に、このメソッドを呼び出す。例を示す。</p> <pre>buf.clear(); // Prepare buffer for reading in.read(buf); // Read data</pre> <p>このメソッドはバッファ内のデータを実際に消去するわけではない。しかし、そうした状況で使用されるため、クリア (clear) と命名されている。</p> <p>戻り値: 現在のバッファ</p>
public final Buffer flip()	<p>このバッファをフリップ (反転) する。リミットは現在ポジションの値に設定され、現在ポジションを表す値は 0 に設定される。マークが定義されている場合、そのマークは破棄される。</p> <p>チャンネル読み込み操作 (put) のあと、このメソッドを呼び出してチャンネル書き込み操作 (相対「get」) の準備を行う。例を示す。</p> <pre>buf.put(magic); // Prepend header in.read(buf); // Read data into rest of buffer buf.flip(); // Flip buffer out.write(buf); // Write header + data to channel</pre> <p>ある場所から別の場所にデータを転送する際、このメソッドを compact メソッドと組み合わせ使用することがある。</p> <p>戻り値: 現在のバッファ</p>
public final Buffer rewind()	<p>このバッファをリワインド (巻き戻し) する。ポジションは 0 に設定され、マークは破棄される。</p> <p>このメソッドは、リミットを正しく設定したあと、チャンネル書き込み操作 (get) の前に呼び出す。例を示す。</p> <pre>out.write(buf); // Write remaining data buf.rewind(); // Rewind buffer buf.get(array); // Copy data into array</pre> <p>戻り値: 現在のバッファ</p>
public final int remaining()	<p>現在ポジションからリミットまでに存在する要素の数を返す。</p> <p>戻り値: このバッファ内に残っている要素数</p>
public final boolean hasRemaining()	<p>現在ポジションからリミットまでに要素が 1 つでも存在するかどうかを判断する。</p> <p>戻り値: このバッファ内に要素が 1 個以上存在する場合にかぎり true</p>
public abstract boolean isReadOnly()	<p>このバッファが読み取り専用であるかどうかを判断する。</p> <p>戻り値: このバッファが読み込み専用である場合にかぎり true</p>
public abstract boolean hasArray()	<p>現在のバッファがアクセス可能な配列に連動するかどうかを判断する。</p> <p>このメソッドの戻り値が true であれば、array メソッドおよび arrayOffset メソッドを安全に呼び出すことができる。</p> <p>戻り値: 現在のバッファが配列に連動しており、読み取り専用でない場合にかぎり true</p>
public abstract Object array()	<p>現在のバッファを補助する配列を返す (オプション)。</p> <p>このメソッドは、配列を利用するバッファをネイティブコードにより効率よく渡すために使用する。具象サブクラスは、このメソッドの戻り値として、より強く型付けされた値を返す。現在のバッファのコンテンツに変更を加えると、返される配列のコンテンツも変更される。その逆も同様である。</p>



	<p>このメソッドを呼び出す前に <code>hasArray</code> メソッドを呼び出し、現在のバッファがアクセス可能な補助配列を持っていることを確認する。</p> <p>戻り値: 現在のバッファを補助する配列</p> <p>例外: <code>ReadOnlyBufferException</code> - 現在のバッファが配列に連動しており、しかも読み込み専用である場合 <code>UnsupportedOperationException</code> - 現在のバッファがアクセス可能な配列を利用しない場合</p>
<code>public abstract int arrayOffset()</code>	<p>現在のバッファの補助配列内にある、このバッファの最初の要素のオフセットを返す(オプション)。</p> <p>現在のバッファが配列に連動していれば、そのポジション <code>p</code> が配列のインデックス <code>p + arrayOffset()</code> と一致する。</p> <p>このメソッドを呼び出す前に <code>hasArray</code> メソッドを呼び出し、現在のバッファがアクセス可能な補助配列を持っていることを確認する。</p> <p>戻り値: 現在のバッファの配列内にある、このバッファの最初の要素のオフセット</p> <p>例外: <code>ReadOnlyBufferException</code> - 現在のバッファが配列に連動しており、しかも読み込み専用である場合 <code>UnsupportedOperationException</code> - 現在のバッファがアクセス可能な配列を利用しない場合</p>
<code>public abstract boolean isDirect()</code>	<p>このバッファがダイレクト・バッファであるかどうかを判断する。</p> <p>戻り値: 現在のバッファがダイレクト・バッファである場合にかぎり <code>true</code></p>
<code>public abstract byte get()</code>	<p>相対「<code>get</code>」メソッドです。このバッファの現在ポジションからバイトを読み込み、現在ポジションの値を増加する。</p> <p>戻り値: バッファの現在ポジションのバイト</p> <p>例外: <code>BufferUnderflowException</code> - バッファの現在ポジションがリミット以上である場合</p>
<code>public abstract ByteBuffer put(byte b)</code>	<p>相対「<code>put</code>」メソッドです (オプション)。</p> <p>バッファの現在ポジションに指定されたバイトを書き込み、現在ポジションの値を増加する。</p> <p>パラメタ: <code>b</code> - 書き込まれるバイト</p> <p>戻り値: 現在のバッファ</p> <p>例外: <code>BufferOverflowException</code> - 現在のバッファの現在ポジションがリミット以上である場合 <code>ReadOnlyBufferException</code> - 現在のバッファが読み込み専用バッファである場合</p>
<code>public abstract byte get(int index)</code>	<p>絶対「<code>get</code>」メソッドです。指定されたインデックスポジションのバイトを読み込む。</p> <p>パラメタ: <code>index</code> - バイトの読み込みポジションを示すインデックス</p> <p>戻り値: 指定されたインデックスポジションのバイト</p> <p>例外: <code>IndexOutOfBoundsException</code> - <code>index</code> が負の数である場合、またはバッファのリミット以上である場合</p>
<code>public abstract ByteBuffer put(int)</code>	<p>絶対「<code>put</code>」メソッドです (オプション)。</p>

<p>index,  byte b)</p>	<p>このバッファの指定されたインデックスポジションに指定されたバイトを書き込む。          パラメタ:          index - バイトの書き込み先を示すインデックス          b - 書き込まれるバイト値          戻り値:          現在のバッファ          例外:          IndexOutOfBoundsException - index が負の数である場合、またはバッファのリミット以上である場合          ReadOnlyBufferException - 現在のバッファが読み込み専用バッファである場合</p>
<p>public ByteBuffer <b>get</b>(byte[] dst, int offset, int length)</p>	<p>相対一括「get」メソッドである。          このメソッドは、このバッファから指定された配列へバイトを転送する。このバッファ内に残っているバイト数が要求に満たない場合 (つまり、length &gt; remaining() である場合)、バイトは一切転送されず、BufferUnderflowException がスローされる。          それ以外の場合、このメソッドは、このバッファの現在ポジションから length バイトを指定された配列の指定されたオフセットポジションへコピーする。そのたびに、現在のバッファのポジションが length ずつ増加する。          このメソッドを src.get(dst, off, len) の形式で呼び出すと、次のループとまったく同じ結果になる。  <pre>for (int i = off; i &lt; off + len; i++)     dst[i] = src.get();</pre>         ただし、このバッファ内に十分な数のバイト・数があることを最初に確認する動作は除く。また、このメソッドを使用したほうがループよりもはるかに効率的である。          パラメタ:          dst - バイトの書き込み先となる配列          offset - 最初のバイトの書き込み先となる配列内のオフセット。dst.length 以下の負でない値でなければならない          length - 指定された配列に書き込まれる最大バイト数。dst.length - offset 以下の負でない値でなければならない          戻り値:          現在のバッファ          例外:          BufferUnderflowException - このバッファ内に残っているバイト数が length よりも少ない場合          IndexOutOfBoundsException - offset パラメタと length パラメタの前提条件が満たされていない場合</p>
<p>public ByteBuffer <b>get</b>(byte[] dst)</p>	<p>相対一括「get」メソッドである。          このメソッドは、このバッファから指定された配列へバイトを転送する。このメソッドを src.get(a) の形式で呼び出すと、次の呼び出しと同じ結果になる。  <pre>src.get(a, 0, a.length)</pre>         戻り値:          現在のバッファ          例外:          BufferUnderflowException - このバッファ内に残っているバイト数が length よりも少ない場合</p>
<p>public ByteBuffer <b>put</b>(ByteBuffer src)</p>	<p>相対一括「put」メソッドである(オプション)。          このメソッドは、指定されたソース・バッファ内に残っているバイトをこのバッファへ転送する。ソース・バッファ内に残っているバイト数がこのバッファ内に残っているバイト数よりも多い場合 (つまり、src.remaining() &gt; remaining() である場合)、バイトは一切転送されず、BufferOverflowException がスローされる。          それ以外の場合、このメソッドは、指定されたバッファの現在ポジションからこのバッファの現在ポジションへ n = src.remaining() バイトをコピーする。そのたびに、両方のバッファのポジションが n ずつ増加する。</p>

	<p>このメソッドを <code>dst.put(src)</code> の形式で呼び出すと、次のループとまったく同じ結果になる。</p> <pre>while (src.hasRemaining())     dst.put(src.get());</pre> <p>ただし、現在のバッファ内に十分な容量があることを最初に確認する動作は、このメソッドに固有である。また、このメソッドのほうがループよりもずっと効率的である。</p> <p>パラメタ:</p> <p><code>src</code> - バイトの読み込み先となるソース・バッファ (このバッファ以外)</p> <p>戻り値: 現在のバッファ</p> <p>例外: <b>BufferOverflowException</b> - このバッファに、ソース・バッファ内に残っているバイトを格納できるだけの容量がない場合 <b>IllegalArgumentException</b> - ソース・バッファとして現在のバッファを指定した場合 <b>ReadOnlyBufferException</b> - 現在のバッファが読み込み専用バッファである場合</p>
<p><code>public ByteBuffer put(byte[] src, int offset, int length)</code></p>	<p>相対一括「put」メソッドである (オプション)。</p> <p>このメソッドは、指定されたソース配列からこのバッファへバイトを転送する。配列からコピーするバイト数がこのバッファ内に残っているバイト数より多い場合 (つまり、<code>length &gt; remaining()</code> である場合)、バイトは一切転送されず、<b>BufferOverflowException</b> がスローされる。</p> <p>それ以外の場合、このメソッドは、指定された配列の指定されたオフセットポジションからこのバッファの現在ポジションへ <code>length</code> 分バイトをコピーする。そのたびに、現在のバッファのポジションが <code>length</code> ずつ増加する。</p> <p>このメソッドを <code>dst.put(src, off, len)</code> の形式で呼び出すと、次のループとまったく同じ結果になる。</p> <pre>for (int i = off; i &lt; off + len; i++)     dst.put(a[i]);</pre> <p>ただし、現在のバッファ内に十分な容量があることを最初に確認する動作は、このメソッドに固有である。また、このメソッドのほうがループよりもずっと効率的である。</p> <p>パラメタ:</p> <p><code>src</code> - バイトの読み込み先となる配列</p> <p><code>offset</code> - 最初のバイトの読み込み先となる配列内のオフセット。 <code>array.length</code> 以下の負でない値でなければならない</p> <p><code>length</code> - 指定された配列から読み取られるバイト数。 <code>array.length - offset</code> 以下の負でない値でなければならない</p> <p>戻り値: 現在のバッファ</p> <p>例外: <b>BufferOverflowException</b> - 現在のバッファ内に残っている容量が不足している場合 <b>IndexOutOfBoundsException</b> - <code>offset</code> パラメタと <code>length</code> パラメタの前提条件が満たされていない場合 <b>ReadOnlyBufferException</b> - 現在のバッファが読み込み専用バッファである場合</p>
<p><code>public final ByteBuffer put(byte[] src)</code></p>	<p>相対一括「put」メソッドである (オプション)。</p> <p>このメソッドは、ソースとなる指定されたバイト配列の内容全体をこのバッファへ転送する。このメソッドを <code>dst.put(a)</code> の形式で呼び出すと、次の呼び出しと同じ結果になる。</p> <pre>dst.put(a, 0, a.length)</pre> <p>戻り値: 現在のバッファ</p> <p>例外: <b>BufferOverflowException</b> - 現在のバッファ内に残っている容量が不足している場合 <b>ReadOnlyBufferException</b> - 現在のバッファが読み込み専用バッファである場合</p>
<p><code>public final boolean hasArray()</code></p>	<p>このバッファがアクセス可能なバイト配列に連動するかどうかを判断する。</p> <p>このメソッドの戻り値が <code>true</code> であれば、<code>array</code> メソッドおよび <code>arrayOffset</code> メソッドを安全に</p>

	<p>呼び出すことができる。</p> <p>定義: クラス Buffer 内の hasArray</p> <p>戻り値: 現在のバッファが配列に連動しており、読み取り専用でない場合にかぎり true</p>
public final byte[] array()	<p>このバッファを補助するバイト配列を返す (任意操作)。 現在のバッファのコンテンツに変更を加えると、返される配列のコンテンツも変更される。その逆も同様である。</p> <p>このメソッドを呼び出す前に hasArray メソッドを呼び出し、現在のバッファがアクセス可能な補助配列を持っていることを確認する。</p> <p>定義: クラス Buffer 内の array</p> <p>戻り値: 現在のバッファを補助する配列</p> <p>例外: ReadOnlyBufferException - 現在のバッファが配列に連動しており、しかも読み込み専用である場合 UnsupportedOperationException - 現在のバッファがアクセス可能な配列を利用しない場合</p>
public final int arrayOffset()	<p>現在のバッファの補助配列内にある、このバッファの最初の要素のオフセットを返す(オプション)。現在のバッファが配列に連動していれば、そのポジション p が配列のインデックス p + arrayOffset() と一致する。</p> <p>このメソッドを呼び出す前に hasArray メソッドを呼び出し、現在のバッファがアクセス可能な補助配列を持っていることを確認する。</p> <p>定義: クラス Buffer 内の arrayOffset</p> <p>戻り値: 現在のバッファの配列内にある、このバッファの最初の要素のオフセット</p> <p>例外: ReadOnlyBufferException - 現在のバッファが配列に連動しており、しかも読み込み専用である場合 UnsupportedOperationException - 現在のバッファがアクセス可能な配列を利用しない場合</p>
public abstract ByteBuffer compact()	<p>現在のバッファを圧縮する(オプション)。 バッファの現在ポジションからリミットまでの間にバイトが存在する場合、これらをバッファの先頭にコピーする。つまり、インデックスポジション p = position() のバイトがインデックス 0 にコピーされ、インデックスポジション p + 1 のバイトがインデックス 1 にコピーされるということである。インデックスポジション limit() - 1 のバイトがインデックス n = limit() - 1 - p にコピーされるまで、同様の処理が繰り返される。最終的にバッファのポジションは n+1 に設定され、リミットは容量の値と等しくなる。マークは破棄される。</p> <p>バッファのポジションは、0 ではなく、コピーされるバイト数と等しくなる。したがって、このメソッドを呼び出したあと、すぐに別の相対「put」メソッドを呼び出すことができる。</p> <p>このメソッドは、書き込みが終了しなかった場合にバッファからのデータの書き込みを実行する前に呼び出す。次のループは、バッファ buf を使って、あるチャンネルから別のチャンネルにバイトをコピーする。</p> <pre>buf.clear(); // Prepare buffer for use while (in.read(buf) &gt;= 0    buf.position != 0) {     buf.flip();     out.write(buf);     buf.compact(); // In case of partial write }</pre> <p>戻り値:</p>

	<p>現在のバッファ</p> <p>例外:  <b>ReadOnlyBufferException</b> - 現在のバッファが読み込み専用バッファである場合</p>
<b>public abstract boolean isDirect()</b>	<p>このバイト・バッファがダイレクト・バッファであるかどうかを判断する。</p> <p>定義:          クラス <b>Buffer</b> 内の <b>isDirect</b></p> <p>戻り値:          現在のバッファがダイレクト・バッファである場合にかぎり <b>true</b></p>
<b>public String toString()</b>	<p>このバッファの状態を要約した文字列を返す。</p> <p>オーバーライド:          クラス <b>Object</b> 内の <b>toString</b></p> <p>戻り値:          概要文字列</p>
<b>public int hashCode()</b>	<p>現在のバッファの現在のハッシュコードを返す。</p> <p><b>byte</b> バッファのハッシュコードは、バッファ内に残っている残りの要素、すなわち <b>position()</b> ~ <b>limit() - 1</b> の要素だけに依存する。</p> <p>バッファのハッシュコードはコンテンツ依存型である。今後バッファのコンテンツが変更されないことが明らかでないかぎり、バッファをハッシュマップその他のデータ構造のキーとして使用することは避けること。</p> <p>オーバーライド:          クラス <b>Object</b> 内の <b>hashCode</b></p> <p>戻り値:          現在のバッファの現在のハッシュコード</p> <p>関連項目:  <b>Object.equals(java.lang.Object)</b>, <b>Hashtable</b></p>
<b>public boolean equals(Object ob)</b>	<p>現在のバッファが別のオブジェクトと等価であるかどうかを判断する。</p> <p>2つのバイト・バッファは、次の場合にかぎり等価である。</p> <ul style="list-style-type: none"> <li>• 要素の型が同じである</li> <li>• バッファ内に残っている要素数が同じである</li> <li>• バッファ内に残っている要素のシーケンス (開始ポジションとは無関係) が各点で等しい (<b>pointwise equal</b>)</li> </ul> <p><b>byte</b> バッファが、その他の型のオブジェクトと等価になることはない。</p> <p>オーバーライド:          クラス <b>Object</b> 内の <b>equals</b></p> <p>パラメタ:  <b>ob</b> - 現在のバッファと比較するオブジェクト</p> <p>戻り値:          現在のバッファが指定されたオブジェクトと等価である場合にかぎり <b>true</b></p> <p>関連項目:  <b>Object.hashCode()</b>, <b>Hashtable</b></p>
<b>public int compareTo(ByteBuffer that)</b>	<p>現在のバッファを別のバッファと比較する。</p> <p>2つのバイト・バッファを比較する際は、バッファ内に残っている要素のシーケンスが辞書順に比較される。このとき、双方のバッファ内に残っているシーケンスの開始ポジションは考慮されない。</p> <p>このとき、双方のバッファ内に残っているシーケンスの開始ポジションは考慮されない。</p> <p>定義:          インタフェース <b>Comparable&lt;ByteBuffer&gt;</b> 内の <b>compareTo</b></p> <p>パラメタ:  <b>that</b> - the object to be compared.</p> <p>戻り値:</p>

	このバッファが指定されたバッファより小さい場合は負の整数、等しい場合は 0、大きい場合は正の整数
<b>public final ByteOrder order()</b>	現在のバッファのバイト順序を取得する。 バイト順序は、複数バイトの値を読み取るときや書き込むとき、そしてこのバイト・バッファのビューとなるバッファを作成するときに使用する。新しく作成されたバイト・バッファの順序は常に <b>BIG_ENDIAN</b> になる。 戻り値: 現在のバッファのバイト順序
<b>public final ByteBuffer order(ByteOrder bo)</b>	このバッファのバイト順序を変更する。 パラメタ: <b>bo</b> - 新しいバイト順序。 <b>BIG_ENDIAN</b> 、 <b>LITTLE_ENDIAN</b> のいずれか 戻り値: 現在のバッファ
<b>public abstract char getChar()</b>	<b>char</b> 値を読み取る相対「 <b>get</b> 」メソッドである。 このバッファの現在ポジションから 2 バイトを読み込み、現在のバイト順序に従って、これらを <b>char</b> 値に変換する。ポジションの値は、そのたびに 2 ずつ増加する。 戻り値: バッファの現在ポジションの <b>char</b> 値 例外: <b>BufferUnderflowException</b> - このバッファ内に残っているバイト数が 2 バイトより少ない場合
<b>public abstract ByteBuffer putChar(char value)</b>	<b>char</b> 値を書き込む相対「 <b>put</b> 」メソッドである(任意操作)。 このバッファの現在ポジションに、現在のバイト順序に従って、指定された <b>char</b> 値を含む 2 バイトを書き込む。ポジションの値は、そのたびに 2 ずつ増加する。 パラメタ: <b>value</b> - 書き込まれる <b>char</b> 値 戻り値: 現在のバッファ 例外: <b>BufferOverflowException</b> - このバッファ内に残っているバイト数が 2 バイトより少ない場合 <b>ReadOnlyBufferException</b> - 現在のバッファが読み込み専用バッファである場合
<b>public abstract char getChar(int index)</b>	<b>char</b> 値を読み取る絶対「 <b>get</b> 」メソッド。指定されたインデックスポジションにある 2 バイトを読み込み、現在のバイト順序に従って、これらを <b>char</b> 値に変換する。 パラメタ: <b>index</b> - バイトの読み込みポジションを示すインデックス 戻り値: 指定されたインデックスポジションの <b>char</b> 値 例外: <b>IndexOutOfBoundsException</b> - <b>index</b> が負の値である場合、またはバッファのリミット以上である場合、 -1
<b>public abstract ByteBuffer putChar(int index, char value)</b>	<b>char</b> 値を書き込む絶対「 <b>put</b> 」メソッドである(任意操作)。 このバッファの指定されたインデックスポジションに、現在のバイト順序に従って、指定された <b>char</b> 値を含む 2 バイトを書き込む。 パラメタ: <b>index</b> - バイトの書き込み先を示すインデックス <b>value</b> - 書き込まれる <b>char</b> 値 戻り値: 現在のバッファ 例外:

	<p><b>IndexOutOfBoundsException</b> - index が負の値である場合、またはバッファのリミット以上である場合、-1</p> <p><b>ReadOnlyBufferException</b> - 現在のバッファが読み込み専用バッファである場合</p>
<p><b>public abstract CharBuffer asCharBuffer()</b></p>	<p>文字バッファとしてこのバイト・バッファのビューを作成する。</p> <p>新しいバッファのコンテンツは、現在のバッファの現在ポジションから始まる。現在のバッファのコンテンツに変更を加えると、その内容が新しいバッファに反映される。新しいバッファのコンテンツに変更を加えると、その内容が現在のバッファに反映される。2つのバッファのポジション、リミット、マークの値はそれぞれ異なる。</p> <p>新しいバッファのポジションは 0、容量とリミットはこのバッファ内に残っているバイト数の1/2になる。マークは定義されない。新しいバッファは、現在のバッファがダイレクト・バッファである場合に限りダイレクト・バッファになる。また、現在のバッファが読み取り専用バッファである場合に限り読み取り専用バッファになる。</p> <p>戻り値: 新しい文字バッファ</p>
<p><b>public abstract short getShort()</b></p>	<p>このバッファの現在ポジションから 2 バイトを読み込み、現在のバイト順序に従って、これらを short 値に変換する。ポジションの値は、そのたびに 2 ずつ増加する。</p> <p>戻り値: バッファの現在ポジションの short 値</p> <p>例外: <b>BufferUnderflowException</b> - このバッファ内に残っているバイト数が 2 バイトより少ない場合</p>
<p><b>public abstract ByteBuffer putShort(short value)</b></p>	<p>short 値を書き込む相対「put」メソッドである(任意操作)。</p> <p>このバッファの現在ポジションに、現在のバイト順序に従って、指定された short 値を含む 2 バイトを書き込む。ポジションの値は、そのたびに 2 ずつ増加する。</p> <p>パラメタ: value - 書き込まれる short 値</p> <p>戻り値: 現在のバッファ</p> <p>例外: <b>BufferOverflowException</b> - このバッファ内に残っているバイト数が 2 バイトより少ない場合</p> <p><b>ReadOnlyBufferException</b> - 現在のバッファが読み込み専用バッファである場合</p>
<p><b>public abstract short getShort(int index)</b></p>	<p>short 値を読み取る絶対「get」メソッドである。</p> <p>指定されたインデックスポジションにある 2 バイトを読み込み、現在のバイト順序に従って、これらを short 値に変換する。</p> <p>パラメタ: index - バイトの読み込みポジションを示すインデックス</p> <p>戻り値: 指定されたインデックスポジションの short 値</p> <p>例外: <b>IndexOutOfBoundsException</b> - index が負の値である場合、またはバッファのリミット以上である場合、-1</p>
<p><b>public abstract ByteBuffer putShort(int index, short value)</b></p>	<p>short 値を書き込む絶対「put」メソッドである(任意操作)。</p> <p>このバッファの指定されたインデックス・ポジションに、現在のバイト順序に従って、指定された short 値を含む 2 バイトを書き込む。</p> <p>パラメタ: index - バイトの書き込み先を示すインデックス value - 書き込まれる short 値</p> <p>戻り値: 現在のバッファ</p> <p>例外:</p>

	<p><b>IndexOutOfBoundsException</b> - index が負の値である場合、またはバッファのリミット以上である場合、-1</p> <p><b>ReadOnlyBufferException</b> - 現在のバッファが読み込み専用バッファである場合</p>
<p><b>public abstract ShortBuffer asShortBuffer()</b></p>	<p>short バッファとしてこのバイト・バッファのビューを作成する。</p> <p>新しいバッファのコンテンツは、現在のバッファの現在ポジションから始まる。現在のバッファのコンテンツに変更を加えると、その内容が新しいバッファに反映される。新しいバッファのコンテンツに変更を加えると、その内容が現在のバッファに反映される。2つのバッファのポジション、リミット、マークの値はそれぞれ異なる。</p> <p>新しいバッファのポジションは 0、容量とリミットはこのバッファ内に残っているバイト数の 1/2 になる。マークは定義されない。新しいバッファは、現在のバッファが直接・バッファである場合に限り直接・バッファになる。また、現在のバッファが読み取り専用バッファである場合に限り読み取り専用バッファになる。</p> <p>戻り値: 新しい short バッファ</p>
<p><b>public abstract int getInt()</b></p>	<p>int 値を読み取る相対「get」メソッドである。</p> <p>このバッファの現在ポジションから 4 バイトを読み込み、現在のバイト順序に従って、これらを int 値に変換する。ポジションの値は、そのたびに 4 ずつ増加する。</p> <p>戻り値: バッファの現在ポジションの int 値</p> <p>例外: <b>BufferUnderflowException</b> - このバッファ内に残っているバイト数が 4 バイトより少ない場合</p>
<p><b>public abstract ByteBuffer putInt(int value)</b></p>	<p>int 値を書き込む相対「put」メソッドである(任意操作)。</p> <p>このバッファの現在ポジションに、現在のバイト順序に従って、指定された int 値を含む 4 バイトを書き込む。ポジションの値は、そのたびに 4 ずつ増加する。</p> <p>パラメタ: value - 書き込まれる int 値</p> <p>戻り値: 現在のバッファ</p> <p>例外: <b>BufferOverflowException</b> - このバッファ内に残っているバイト数が 4 バイトより少ない場合 <b>ReadOnlyBufferException</b> - 現在のバッファが読み込み専用バッファである場合</p>
<p><b>public abstract int getInt(int index)</b></p>	<p>int 値を読み取る絶対「get」メソッドである。</p> <p>指定されたインデックス・ポジションにある 4 バイトを読み込み、現在のバイト順序に従って、これらを int 値に変換する。</p> <p>パラメタ: index - バイトの読み込みポジションを示すインデックス</p> <p>戻り値: 指定されたインデックス・ポジションの int 値</p> <p>例外: <b>IndexOutOfBoundsException</b> - index が負の値である場合、またはバッファのリミット以上である場合、-3</p>
<p><b>public abstract ByteBuffer putInt(int index, int value)</b></p>	<p>int 値を書き込む絶対「put」メソッドである(任意操作)。</p> <p>このバッファの指定されたインデックス・ポジションに、現在のバイト順序に従って、指定された int 値を含む 4 バイトを書き込む。</p> <p>パラメタ: index - バイトの書き込み先を示すインデックス value - 書き込まれる int 値</p> <p>戻り値: 現在のバッファ</p>



	<p>例外:  <b>IndexOutOfBoundsException</b> - index が負の値である場合、またはバッファのリミット以上である場合、-3  <b>ReadOnlyBufferException</b> - 現在のバッファが読み込み専用バッファである場合</p>
<p><b>public abstract IntBuffer asIntBuffer()</b></p>	<p>int バッファとしてこのバイト・バッファのビューを作成する。  新しいバッファのコンテンツは、現在のバッファの現在ポジションから始まる。現在のバッファのコンテンツに変更を加えると、その内容が新しいバッファに反映される。新しいバッファのコンテンツに変更を加えると、その内容が現在のバッファに反映される。2つのバッファのポジション、リミット、マークの値はそれぞれ異なる。  新しいバッファのポジションは0、容量とリミットはこのバッファ内に残っているバイト数の1/4になる。マークは定義されない。新しいバッファは、現在のバッファがダイレクト・バッファである場合に限りダイレクト・バッファになる。また、現在のバッファが読み取り専用バッファである場合に限り読み取り専用バッファになる。  戻り値:  新しい int バッファ</p>
<p><b>public abstract long getLong()</b></p>	<p>long 値を読み取る相対「get」メソッドである。  このバッファの現在ポジションから8バイトを読み込み、現在のバイト順序に従って、これらを long 値に変換する。ポジションの値は、そのたびに8ずつ増加する。  戻り値:  バッファの現在ポジションの long 値  例外:  <b>BufferUnderflowException</b> - このバッファ内に残っているバイト数が8バイトより少ない場合</p>
<p><b>public abstract ByteBuffer putLong(long value)</b></p>	<p>long 値を書き込む相対「put」メソッドである(任意操作)。  このバッファの現在ポジションに、現在のバイト順序に従って、指定された long 値を含む8バイトを書き込む。ポジションの値は、そのたびに8ずつ増加する。  パラメタ:  value - 書き込まれる long 値  戻り値:  現在のバッファ  例外:  <b>BufferOverflowException</b> - このバッファ内に残っているバイト数が8バイトより少ない場合  <b>ReadOnlyBufferException</b> - 現在のバッファが読み込み専用バッファである場合</p>
<p><b>public abstract long getLong(int index)</b></p>	<p>long 値を読み取る絶対「get」メソッドである。  指定されたインデックス・ポジションにある8バイトを読み込み、現在のバイト順序に従って、これらを long 値に変換する。  パラメタ:  index - バイトの読み込みポジションを示すインデックス  戻り値:  指定されたインデックスポジションの long 値  例外:  <b>IndexOutOfBoundsException</b> - index が負の値である場合、またはバッファのリミット以上である場合、-7</p>
<p><b>public abstract ByteBuffer putLong(int index, long value)</b></p>	<p>long 値を書き込む絶対「put」メソッドである(任意操作)。  このバッファの指定されたインデックス・ポジションに、現在のバイト順序に従って、指定された long 値を含む8バイトを書き込む。  パラメタ:  index - バイトの書き込み先を示すインデックス  value - 書き込まれる long 値  戻り値:</p>

	<p>現在のバッファ 例外: IndexOutOfBoundsException - index が負の値である場合、またはバッファのリミット以上である場合、-7 ReadOnlyBufferException - 現在のバッファが読み込み専用バッファである場合</p>
public abstract LongBuffer asLongBuffer()	<p>long バッファとしてこのバイト・バッファのビューを作成する。 新しいバッファのコンテンツは、現在のバッファの現在ポジションから始まる。現在のバッファのコンテンツに変更を加えると、その内容が新しいバッファに反映される。新しいバッファのコンテンツに変更を加えると、その内容が現在のバッファに反映される。2つのバッファのポジション、リミット、マークの値はそれぞれ異なる。 新しいバッファのポジションは 0、容量とリミットはこのバッファ内に残っているバイト数の1/8になる。マークは定義されない。新しいバッファは、現在のバッファがダイレクト・バッファである場合に限りダイレクト・バッファになる。また、現在のバッファが読み取り専用バッファである場合に限り読み取り専用バッファになる。 戻り値: 新しい long バッファ</p>
public abstract float getFloat()	<p>float 値を読み取る相対「get」メソッドである。 このバッファの現在ポジションから 4 バイトを読み込み、現在のバイト順序に従って、これらを float 値に変換する。ポジションの値は、そのたびに 4 ずつ増加する。 戻り値: バッファの現在ポジションの float 値 例外: BufferUnderflowException - このバッファ内に残っているバイト数が 4 バイトより少ない場合</p>
public abstract ByteBuffer putFloat(float value)	<p>float 値を書き込む相対「put」メソッドである(任意操作)。 このバッファの現在ポジションに、現在のバイト順序に従って、指定された float 値を含む 4 バイトを書き込む。ポジションの値は、そのたびに 4 ずつ増加する。 パラメタ: value - 書き込まれる float 値 戻り値: 現在のバッファ 例外: BufferOverflowException - このバッファ内に残っているバイト数が 4 バイトより少ない場合 ReadOnlyBufferException - 現在のバッファが読み込み専用バッファである場合</p>
public abstract float getFloat(int index)	<p>float 値を読み取る絶対「get」メソッドである。 指定されたインデックスポジションにある 4 バイトを読み込み、現在のバイト順序に従って、これらを float 値に変換する。 パラメタ: index - バイトの読み込みポジションを示すインデックス 戻り値: 指定されたインデックス・ポジションの float 値 例外: IndexOutOfBoundsException - index が負の値である場合、またはバッファのリミット以上である場合、-3</p>
public abstract ByteBuffer putFloat(int index, float value)	<p>float 値を書き込む絶対「put」メソッドである(任意操作)。 このバッファの指定されたインデックス・ポジションに、現在のバイト順序に従って、指定された float 値を含む 4 バイトを書き込む。 パラメタ: index - バイトの書き込み先を示すインデックス value - 書き込まれる float 値</p>

	<p>戻り値: 現在のバッファ 例外: <b>IndexOutOfBoundsException</b> - index が負の値である場合、またはバッファのリミット以上である場合、-3 <b>ReadOnlyBufferException</b> - 現在のバッファが読み込み専用バッファである場合</p>
<p><b>public abstract FloatBuffer asFloatBuffer()</b></p>	<p>float バッファとしてこのバイト・バッファのビューを作成する。 新しいバッファのコンテンツは、現在のバッファの現在ポジションから始まる。現在のバッファのコンテンツに変更を加えると、その内容が新しいバッファに反映される。新しいバッファのコンテンツに変更を加えると、その内容が現在のバッファに反映される。2つのバッファのポジション、リミット、マークの値はそれぞれ異なる。 新しいバッファのポジションは 0、容量とリミットはこのバッファ内に残っているバイト数の 1/4 になる。マークは定義されない。新しいバッファは、現在のバッファがダイレクト・バッファである場合に限りダイレクト・バッファになる。また、現在のバッファが読み取り専用バッファである場合に限り読み取り専用バッファになる。 戻り値: 新しい float バッファ</p>
<p><b>public abstract double getDouble()</b></p>	<p>このバッファの現在ポジションから 8 バイトを読み込み、現在のバイト順序に従って、これらを double 値に変換する。ポジションの値は、そのたびに 8 ずつ増加する。 戻り値: バッファの現在ポジションの double 値 例外: <b>BufferUnderflowException</b> - このバッファ内に残っているバイト数が 8 バイトより少ない場合</p>
<p><b>public abstract ByteBuffer putDouble(double value)</b></p>	<p>double 値を書き込む絶対「put」メソッドである(任意操作)。 このバッファの現在ポジションに、現在のバイト順序に従って、指定された double 値を含む 8 バイトを書き込む。ポジションの値は、そのたびに 8 ずつ増加する。 パラメタ: value - 書き込まれる double 値 戻り値: 現在のバッファ 例外: <b>BufferOverflowException</b> - このバッファ内に残っているバイト数が 8 バイトより少ない場合 <b>ReadOnlyBufferException</b> - 現在のバッファが読み込み専用バッファである場合</p>
<p><b>public abstract double getDouble(int index)</b></p>	<p>double 値を読み取る絶対「get」メソッドである。 指定されたインデックス・ポジションにある 8 バイトを読み込み、現在のバイト順序に従って、これらを double 値に変換する。 パラメタ: index - バイトの読み込みポジションを示すインデックス 戻り値: 指定されたインデックス・ポジションの double 値 例外: <b>IndexOutOfBoundsException</b> - index が負の値である場合、またはバッファのリミット以上である場合、-7</p>
<p><b>public abstract ByteBuffer putDouble(int index, double value)</b></p>	<p>double 値を書き込む絶対「put」メソッドである(任意操作)。 このバッファの指定されたインデックス・ポジションに、現在のバイト順序に従って、指定された double 値を含む 8 バイトを書き込む。 パラメタ: index - バイトの書き込み先を示すインデックス value - 書き込まれる double 値</p>

	戻り値: 現在のバッファ 例外: <b>IndexOutOfBoundsException</b> - index が負の値である場合、またはバッファのリミット以上である場合、-7 <b>ReadOnlyBufferException</b> - 現在のバッファが読み込み専用バッファである場合
<b>public abstract DoubleBuffer asDoubleBuffer()</b>	<b>double</b> バッファとしてこのバイト・バッファのビューを作成する。 新しいバッファのコンテンツは、現在のバッファの現在ポジションから始まる。現在のバッファのコンテンツに変更を加えると、その内容が新しいバッファに反映される。新しいバッファのコンテンツに変更を加えると、その内容が現在のバッファに反映される。2つのバッファのポジション、リミット、マークの値はそれぞれ異なる。 新しいバッファのポジションは0、容量とリミットはこのバッファ内に残っているバイト数の1/8になる。マークは定義されない。新しいバッファは、現在のバッファがダイレクト・バッファである場合に限りダイレクト・バッファになる。また、現在のバッファが読み取り専用バッファである場合に限り読み取り専用バッファになる。 戻り値: 新しい <b>double</b> バッファ

### 5.1.3 インスタンスの生成

ByteBuffer はコンストラクタを持っていない。その代わりに以下に示すように、バイト配列をラップする (ByteBuffer.wrap(byte[]))、あるいは allocate() という割り当てメソッドを呼ぶ (ByteBuffer.allocate(int)) などの手段でインスタンスが生成される。

```
byte[] ba = ...
ByteBuffer bb = ByteArray.wrap(ba);
```

または、

```
ByteBuffer bb = ByteArray.allocate(100);
```

これら以外にもダイレクト・バッファ生成のための AllocateDirect というメソッドも存在する。また slice、duplicate という現在のオブジェクトをコピーしたものを作るメソッド、あるいは asShortBuffer などのプリミティブ処理に適したビュー・バッファという一種のコピーを用意するメソッドたちも存在する。

### 5.1.4 ByteBuffer への読み書き

- 現在ポジションからの読み出しは次のようになる。ポジションは自動的にバイト数分増加する:

```
ByteBuffer bb = ByteBuffer.wrap(...);
byte b1 = bb.get(); // 最初のバイトを読み出す
byte b2 = bb.get(); // 2番目のバイトを読み出す
```

- バイトの数値を文字リテラルで書き込むときは、明示的なキャストを行う。

```
bb.put((byte) 40);
```

- 特定のオフセットからの読み書き

現在のポジション以外の読み書きをする場合には二つの手段がある。最初は position()メソッドを読んで現在ポジションを知り、そのポジションを変更してそこから読み書きをする方法である。ポジション

はその操作に応じて変化する。もうひとつは明示的にオフセットを指示して `get()`あるいは `put()`メソッドを呼ぶ方法で、この場合はこのバッファの現在ポジション値は変化しないことに注意すること。他追えば、

```
bb.put(140, (byte) 32);
```

では 32 というバイト値をポジション 140 に書き込む。

- 符号なしバイト値処理

Java ではバイトというプリミティブは-128 から+127 までの符号付数字として取り扱われているが、これは通信の世界からすると混乱を起ししやすい。符号なしバイトを書き込むときは `int` 型で作った値を `byte` にキャストする。符号なしとして読み出すときは `int` で読み出して、`0xff` で AND をとる。たとえば以下のコードでは、

```
ByteBuffer bb = ByteBuffer.allocate(10);
int i = 254;
System.out.println((byte) i);
bb.put((byte) i);
bb.flip();
i = bb.get() & 0xff;
System.out.println(i);
```

254 という数値を `byte` 型で書き込むと-2 というバイト値として書き込まれるが、これを `bb.get() & 0xff` という `int` 型でとりだすと 254 と正しく取り出されることが確認されよう。

- `byte` 以外のプリミティブとして読み書きするには、その型に対応した `put/get` メソッド (例えば `getInt()`, `getDouble()`, `putShort()` 等) を使用するとよい。また `ByteBuffer.asIntBuffer()` などのメソッドで生成する `IntBuffer`, `LongBuffer`, `FloatBuffer` などのラップ・バッファ (ビュー・バッファともいう) もその型の配列の読み書きに有用である。

```
ByteBuffer bb = ByteBuffer.allocate(50);
IntBuffer ib = bb.asIntBuffer();
int[] data = {100, 200, 300};
ib.put(data);
```

この例では、整数の配列をバイト配列に書き込んでいる。`bb` と `ib` は同じデータを別のビューで見ていることに注意されたい。これにより同じバッファ上に異なった型のデータをミックスさせることが容易になる。

## 5.2節 文字セット変換

Java では Unicode が内部文字コードとして使われている。Unicode そのものは多言語対応で、当初は 16 ビットで定義されていたが、16 ビットでは全く不十分だった。漢字圏だけでの文字コードを考えただけでも、大漢和辞典だけでも親文字は 5 万字を超え、16 ビットの限界の 6.4 万字に迫る。昔はコンピュータの能力上から 2 バイトを超える文字コードは現実的ではなかったが、現在は古書仏典など貴重な各国文化の記録と継承を考えた場合、2 バイト以上の多バイト・コードが現実化している (筆者のある後輩は国立国語研究所で 30 年前に漢字圏のための 4 バイト・コードを提唱していた)。従って Java は現在は 16 ビットの Unicode を実装しているが、今後は拡張されよう。そのような Java のコードで生成された文字列は、外部で利用できるような各

国が標準的に使っている文字コード・セットに変換される。Java の Unicode 文字列をそのまま外部と交換するには、UTF-8 などのエンコーディングが使われるが、日本では日本語文字コードとしては、Microsoft の Windows-31J (Shift-JIS と微妙な差があるので、現在はこちらが標準的に使われている) や EUC-JP などが一般的である。従って、ネットワークやファイルとのデータ交換には文字コード変換が必須である。

### 5.2.1 一般的なクラスを使った変換

第 4 章のエコーバック・サーバのプログラムで使った `java.io.InputStreamReader` と `java.io.OutputStreamWriter`、そして java 1.4 で拡張された `java.lang.String` を利用すれば、Java で使われている Unicode と各言語エンコーディング間の変換が可能である。対応文字セットは [http://download.oracle.com/docs/cd/E17476\\_01/javase/1.4.2/docs/guide/intl/encoding.doc.html](http://download.oracle.com/docs/cd/E17476_01/javase/1.4.2/docs/guide/intl/encoding.doc.html) などを参照されたい。

- `OutputStreamWriter(OutputStream out, CharsetEncoder enc)` // 与えられた文字セットエンコーダを使う `OutputStreamWriter` を作成
- `InputStreamReader(InputStream in, Charset cs)` // 与えられた文字セットを使う `InputStreamReader` を作成
- `String(byte[] bytes, Charset charset)` // 指定された 文字セット を使用して、指定されたバイト配列を復号化することによって、新しい `String` を構築
- `String(byte[] bytes, String charsetName)` // 指定された 文字セット を使用して、指定されたバイト配列を復号化することによって、新しい `String` を構築
- `byte[] String.getBytes(Charset charset)` // 指定された 文字セット を使用してこの `String` をバイトシーケンスに符号化し、結果を新規バイト配列に格納

### 5.2.2 NIO のクラスを使った変換

また java 1.4 で追加された `java.nio` には、`java.nio.charset` のパッケージの中のクラスたちの、`java.nio.charset.Charset`、`java.nio.charset.CharsetEncoder`、及び `java.nio.charset.CharsetDecoder` という文字セット変換の為のクラスが用意されている。

文字列からバイト列への一般的な文字変換は次のようである：

```
// String からバイト列への変換
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.charset.Charset;

String s = ...; // ここで文字列を用意する
Charset cs = Charset.forName("Windows-31J" /* 変換に使う文字セットを指定 */);
CharBuffer cb = CharBuffer.wrap(s); // 文字列を CharBuffer でラップ
ByteBuffer bb = cb.encode(cs); // これをエンコードして ByteBuffer にする
int lim = bb.limit(); // バイト数を取り出す
byte[] b = new byte[lim]; // これをもとにバイト列をつくる

// 上限を設定したバイト列を ByteBuffer にコピー
```

```
bb.get(b, 0 /* オフセット */, lim);
```

文字バッファとバイト・バッファ間の一般的なエンコードとデコードは次のようである:

```
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.charset.Charset;
import java.nio.charset.CharsetDecoder;
import java.nio.charset.CharsetEncoder;

// 文字コードの選択
Charset cs = Charset.forName("Windows-31J");

// バイトから文字へのデコーダの生成
CharsetDecoder dec = cs.newDecoder();

// 文字からバイトへのエンコーダの生成
CharsetEncoder enc = cs.newEncoder();

// ByteBuffer と CharBuffer のオブジェクトがあるとする

// データ読み出しの為に byte[] から char[] への実質的な変換
CharBuffer cb = dec.decode( bb );

// データ書き込みの為に char[] から byte[] への変換
ByteBuffer bb = enc.encode( cb );
```

バイト列から文字列への文字セット変換が含まれる例として、ファイル入出力のシンプルなプログラムを以下に示す:

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.channels.FileChannel;
import java.nio.charset.Charset;

public class NioTest1 {
    private static final int BSIZE = 1024;
    private static final String scs = "Windows-31J";

    /**
     * @param args
     */
    public static void main(String[] args) throws Exception {
        FileChannel fc = new FileOutputStream("test_data.txt").getChannel();
        fc.write(ByteBuffer.wrap(
            "これはNIOのテスト・データ".getBytes(scs)));
        fc.close();
        ByteBuffer bb = ByteBuffer.allocate(BSIZE);
        // 次に書き込んだデータを読み出してみる
        fc = new FileInputStream("test_data.txt").getChannel();
        bb.clear();
        fc.read(bb);
        bb.flip();
        Charset cs = Charset.forName(scs);
        CharBuffer bc = cs.decode(bb);
        System.out.println(bc);
    }
}
```

```
}  
}
```

この例では書き込みには `String.getBytes()` を、読み出しでは `Charset.forName()` と `charset.decode()` で Windows-31J と Unicode 間の文字変換を行っている。Windows-31J でファイルを作成すれば、Windows のアプリケーションではテキスト・ファイルとして読み書きが可能になる。

Eclipse では、`C:\eclipse\workspace\MyProject` のフォルダに `test_data1.txt` というテキスト・ファイルが作られている。適当な Windows ベースのテキスト・エディタで作ったファイルも読み出しが可能であることを自分で確認されたい。

### 5.2.3 PrintStream での変換

もうひとつの手法としては、同じく java 1.4 で追加された `java.io.PrintStream` のエンコーディング指定つきコンストラクタを利用することがある。

```
public PrintStream(OutputStream out, boolean autoFlush, String encoding)  
throws UnsupportedEncodingException
```

というコンストラクタは、新しい `OutputStream` を出力ストリームとし、文字エンコーディングが指定できる `PrintStream` を作成する。ここで `autoFlush` というのは `boolean` 値で、`true` の場合、バイト配列が書き込まれたとき、`println` メソッドの 1 つが呼び出されたとき、または改行文字またはバイト (`\n`) が書き込まれたときに、出力バッファがフラッシュされる。`encoding` はサポートされる「文字エンコーディング」の名前である。指定された文字エンコーディングがサポートされていない場合には `UnsupportedEncodingException` がスローされる。`PrintStream` は `OutputStream` にさまざまなデータ値の表現を簡易的に出力する機能を追加するクラスである。それには `printf()` という C 言語にはなじみのメソッドが含まれている。

例えば実習で作成した `SimpleClient.java` を次のように変更すれば、`printf` といった便利な機能が使えるようになる。

```
java.io.PrintStream upward_stream;           //上りはBufferedWriterを使用  
  
....  
  
    upward_stream = new java.io.PrintStream(client_socket.getOutputStream(),  
true, "Windows-31J");  
  
        //コンソールから1行読み込み  
    data = console_in.readLine();  
        //サーバにこれを送信  
    upward_stream.print(data);  
    upward_stream.printf(" ... Time: %tTS \n",  
java.util.Calendar.getInstance());  
        //サーバからの戻りを受信
```

## 5.3節 NIO のチャンネル



チャンネルという概念はオープンとなっているファイル、ネットワーク接続、あるいは他の I/O の実質的な識別として昔から使われてきている。よく知られるようになったのは 1960 年代の IBM のメインフレーム System/360 であろう。IBM のチャンネルはどちらかといえば I/O 処理のハードウェアが主体であったが、NIO ではソフトウェア機能である。NIO でのチャンネル(Channel)はオープンなファイルあるいは接続への読み書き、またその他のメディア固有の機能を提供するオブジェクトである。

NIO のチャンネルはこれまでの java.io のストリームと似てはいるが、総てのデータは Buffer オブジェクトを介して処理されており、直接あるチャンネルにたいしバイトを書き込むあるいは読み出すということはしないで、あるバッファにデータを書き込むあるいは読み出すことになる。特に、チャンネルとダイレクト・バイト・バッファ(ByteBuffer)とは密な関係があり、このバッファとファイルあるいは接続とのバイト・データの転送が出来ることで、これまでの Java のバイト配列または非直接バッファよりも効率的なものとなっている。

またチャンネルは java.io のストリーム(InputStream と OutputStream)とは違って双方向である。Channel は読み出し、書き込み、あるいはその双方の為にオープンできる。従って元になっている OS の I/O をより良く反映していると言える。

### 5.3.1 ファイル I/O での読み書き

これは既に前節のプログラムで示してある。一般的な手順は次のようになる。

読み出しの場合は:

```
FileInputStream fis = new FileInputStream( "test_data.txt" ); // ファイル入力ストリームを生成 (ファイルのオープン)
FileChannel fc = fis.getChannel(); // チャンネルの生成
ByteBuffer bb = ByteBuffer.allocate( 1024 ); // バイト・バッファを用意
fc.read( bb ); // そのバッファにそのチャンネルから読み取る
```

そのファイルの終わりに達したことは

```
int r = fc.read( bb );
```

で-1 が返されることで知ることが出来る。

書き込みの場合も同じように:

```
FileOutputStream fos = new FileOutputStream( "test_data.txt" ); // ファイル出力ストリームを生成 (ファイルのオープン)
FileChannel fc = fos.getChannel(); // チャンネルの生成
ByteBuffer bb = ByteBuffer.allocate( 1024 ); // バイト・バッファを用意
for (int i=0; i<message.length; ++i) { // メッセージをバイト・バッファに入力
    bb.put( message[i] );
}
buffer.flip(); // バイト・バッファのポジションとリミットを出力のために設定
fc.write( bb ); // そのバッファの中身をそのチャンネルに書き込む
```

### 5.3.2 ネットワーク I/O でのチャンネル

基本的にはネットワークとファイルとではチャンネルの振る舞いは同じである。チャンネルは `InputStream` および `OutputStreams` から取得する。しかしながら一般には(特にサーバの場合は)非同期動作、即ち相手からの接続を待つ、あるいはデータの着信や送信の終了を待つことなくプロセスを進め、そのような事象をイベントとして処理することが行われる。非同期処理に関しては次章で詳しく説明する。この場合はセレクトタというクラスが更に関与する。

サーバの場合の一般的な手順は以下のようである:

```
Selector selector = Selector.open(); // セレクトタをオープンする
ServerSocketChannel ssc = ServerSocketChannel.open(); // サーバ・ソケット・チャンネルをオープンする
ssc.configureBlocking( false ); // 非ブロッキング・モードをオンにする (そうでないときは true を指定)
ServerSocket ss = ssc.socket(); // このチャンネルに関連したサーバソケットを取得
InetSocketAddress address = new InetSocketAddress( ports[i] ); // このソケットの受付ポート番号のセットを設定
ss.bind( address ); // これをそのサーバ・ソケットにバインド
SelectionKey key = ssc.register( selector, SelectionKey.OP_ACCEPT ); // 新しくオープンされたサーバ・ソケット・チャンネルたちのこのセレクトタを必要とするイベントで登録
int num = selector.select(); // その後は登録したイベントが生起するのを待つ
// その後はイベントに応じた処理を行う
```

## 5.4節 非ブロッキング・ソケット I/O

### 5.4.1 非ブロッキング・サーバと Apache MINA

これまで説明してきたように、多くのユーザに対応しているインターネットのアプリケーションでは、TCP 接続毎に、あるいは接続しているユーザの要求ごとにスレッドを生成する、あるいはスレッドのプールからひとつのスレッドをとりだすのが一般的である。前述のエコーバック・サーバの例では `Java.net` ベースであって、TCP 接続毎にスレッドを生成し、そのスレッドがあるユーザに対応している。ウェブの世界で使われている HTTP のプロトコルでは、ユーザがある要求を出すごとに TCP 接続をするバージョン (HTTP 1.0) と TCP 接続をしたら継続して要求を出すバージョン (HTTP 1.1) があり、前者の場合には更には HTTP 要求ごとにスレッドを生成する (サーバレットの場合など) ことになる。

いずれにしても `Java.net` ベースでは利用者が多いアプリケーションの場合にはスレッドの数が非常に大きくなり、サーバにとっては割当てメモリ及び処理オーバーヘッドが増大し、サービスの質の低下をもたらしかねない。特にエコーバックのサーバの場合は、ユーザが入力をするのに時間がかかる (例えば席を外したとかたまたまアクセスするだけの場合など) ことがあり、スレッドの効率が非常に悪くなる。同じことがウェブの世界でのチャットのアプリケーションなどでもいえる。

Java.net ベースの TCP のアプリケーションで効率を悪化させているのは `accept` あるいは `read` のメソッドでのピアからの入力待ちの状態であり、これを Java ではブロックされた(`blocked`)状態だと呼んでいる。JDK 1.4 から導入された Java.nio では非ブロッキングのソケット I/O (Non-Blocking Socket I/O) が利用できる。

Java.nio では、あるスレッドはあるチャンネルに対しバッファにデータを読み込む(あるいは書き込む)ことを依頼する。そのチャンネルがそのバッファにデータを読み込んでいる(あるいは書き込んでいる)間にそのスレッドは他のことが出来る。データがそのバッファに読み込まれたら(あるいは書き込まれたら)、そのスレッドはその処理にかかることが出来る。チャンネルの中での読み書きは別のスレッドが行っていても、メインのスレッドはひとつでも効率よく複数のクライアント要求をこなすことが出来る。

但しクライアントからの接続あるいは要求ごとにスレッドを生成しないでひとつのスレッドで済ますのは、処理のほうでブロッキング状態が起きるアプリケーションでは逆効果となることがあるので、そのような場合には例えばサーブレットの非同期処理などで対処しなければならないことに注意しなければならない(後述の MINA では、処理プロセスをマルチ・スレッド化するフィルタが用意されている)。

なお次の章で紹介する Apache プロジェクトの [MINA](#) というプロジェクトが、NIO ベースではあるものの、より使いやすい非同期処理のメカニズムと API (asynchronous API) を用意している。同じようなフレームワークとして [JBoss Netty](#) も存在している。こちらのほうが、より使いやすくなっているため、NIO の非ブロッキングのソケット I/O は **現在はあまりお勧めできない**。

## 5.4.2 非ブロッキング・ソケット I/O のセレクトタとチャンネル

非ブロッキング・ソケット I/O では、セレクトタ (`java.nio.channels.Selector`) を使うことで単一のスレッド上で複数の同時のソケット接続を扱うことができる。この `Selector` オブジェクトはあるサーバ・プログラムのなかである与えられた時間にアクティブになり得る選択されたキー (`selected keys`) たちのセットを保持している(対象にする IO イベントをユーザが登録する)。そのセレクトタはソケットの状態を常に監視してキーを返す。各キーがあるクライアントとの TCP (あるいは UDP) 接続に対応している。プログラム作成者はこのキーの状態を使って個々のクライアント要求に対処する。

チャンネルとは、ハードウェアデバイス、ファイル、ネットワークソケットのほか、個別の入出力操作を実行できる接続を表している。非ブロッキング可能なチャンネルは、`java.nio.channels.SelectableChannel` クラスを継承したサブクラスで、`java.nio.channels.SocketChannel` と `java.nio.channels.ServerSocketChannel` クラスが含まれる。これらのチャンネルをセレクトタに登録することで、セレクトタがそのチャンネルを監視できるようになる。このようなイメージは、TCP の接続という基本的なコンセプトがやや薄れてしまっていて、ネットワークの人間にはちょっと判り難いものである。

キーは以下の 3 つの条件のひとつが満足されるまでは有効である:

- そのチャンネルがクローズされた
- そのセレクトタがクローズされた
- そのキー自身が `cancel()` メソッドによってキャンセルされた

## 5.4.3 一般的な処理のながれ

非ブロッキング処理の一般的な流れは次のようになる:

1. Selector の生成
2. Channel を生成し、非ブロッキング・モードに設定する
3. Selector に、どのような I/O イベントを対象にするかを登録
4. select メソッドのコール
5. 何らかの I/O 処理が発生し、select メソッドから戻る
6. I/O 処理の種類を調べ、その種類に応じた処理を行う
7. 再び select メソッドをコール(4 から 6 のループを継続)

Selector オブジェクトはそれ自身をファクトリとして使う、あるいは SelectorProvider ファクトリを使って生成される。外部の SelectorProvider ファクトリは既存の独自の非ブロッキングのソケット・ライブラリをより高度化する為に用意されている。最も簡単なソケット生成は以下のコマンドを使う方法である(Selector のコンストラクタは protected となっていて new でインスタンス化できなくなっている):

```
Selector selector = Selector.open();
```

あるいは SelectorProvider ファクトリを使って:

```
Selector selector = SelectorProvider.provider().openSelector();
```

セレクトタは、close メソッドでクローズされるまでオープンの状態を維持する。

到来データをクライアント接続毎に切り分ける為にはあるチャンネルを生成して、それにマルチプレクサである Selector に登録する。最初に ServerSocketChannel を設定して、それをローカル・ホスト上のあるポートにバインドすることで新しい接続の受付を可能とする必要がある:

```
ServerSocketChannel channel = ServerSocketChannel.open();
// channel のオープン (このクラスのコンストラクタも protected となっている)
channel.configureBlocking(false);
// 先ずこの channel を非ブロック・モードにする
InetAddress lh = InetAddress.getLocalHost();
// ローカル・ホストの IP アドレスを取得
InetSocketAddress isa = new InetSocketAddress(lh, port);
// ポートとともにソケット・アドレスを生成
channel.socket().bind(isa);
// そのソケット・アドレスをその channel のソケットにバインド
```

これらのチャンネルはそのプログラムの中で実行されるタスクに従って登録されねばならない (OP\_ACCEPT、OP\_CONNECT、OP\_READ、または OP\_WRITE)。例えば新しい接続を受け付けるチャンネルは次のように登録する:

```
SelectionKey acceptKey = channel.register( selector,
SelectionKey.OP_ACCEPT );
```

またデータを読み書きするチャンネルは次のように登録する:

```
SelectionKey readWriteKey = channel.register( selector, SelectionKey.
OP_READ | SelectionKey.OP_WRITE );
```

このセレクトタはあるクライアントが要求を出した時にキーたちを返す。一般的にはループを使って選択を行う:

```
while ((keysAdded = selector.select()) > 0) {
```

```

Set readyKeys = selector.selectedKeys();
Iterator i = readyKeys.iterator();
// レディとなったキーたちを調べ、要求処理を行う
while (i.hasNext()) {
    SelectionKey sk = (SelectionKey) i.next();
    ... (その要求を受け付けその要求を処理する)
}
}

```

この例でのキーは接続待ち可能(**acceptable**)、着信データ読み出し可能(**readable**)、送信可能(**writable**)、あるいは相手への接続可能(**connectable**)のものであり得る。サーバ側の新しい接続確立時はセレクトが返すキーは接続待ち可能キーとなる。このセレクトはその後データの送受信のイベントに対応した着信データ読み出し可能(**readable**)あるいは送信可能(**writable**)キーを返すことになる。

新しい接続が確立されたときのキーは **isAcceptable()** がセットされている。TCP 接続中はそのキーに **isWritable()** がセットされているときにそのソケットにデータを送りだすことができ、**isReadable()** がセットされているときにのみソケットからデータを読み出すことができる。当該ソケットをキーに添付することで、要求処理にこれを使うことができる。このコード片は **Iterator** を使っていて難解なものではあるが、**Sun** のチュートリアルがベースになっている。

```

while (selector.select() > 0) { // 指定したイベントの生起をここで待つ
    for (Iterator it = selector.selectedKeys().iterator(); it.hasNext();)
    {
        SelectionKey key = (SelectionKey) it.next();
        it.remove();
        if (key.isAcceptable()) {
            doAccept((ServerSocketChannel) key.channel());
        } else if (key.isReadable()) {
            SocketChannel channel = (SocketChannel) key.channel();
            doRead(channel); // (...そのソケットでの読み出しと書き込みを行う)
        }
    }
}
}

```

クライアントがサーバに要求データを送信したら、このセレクトは **isReadable() true** のキーを返す。そのデータをサーバが処理ことになる。そのチャンネルには **Socket** を張り付けてあるので、そのソケットを使ってデータの読み書きを行う。クライアントからのデータを処理した後はソケットは書き込み可能になるので、通常の場合は書き込み可能のチェックは不要である。

データのソケット・チャンネルへの書き込み及び読み出しは **ByteBuffer** しか使えないことに注意されたい。

```

ByteBuffer bb = ByteBuffer.allocate(BUF_SIZE);
Charset charset = Charset.forName("Windows-31J");

```

```

// ここまでは最初に用意しておく
    else if ( key.isWritable() ) {
        SocketChannel channel = (SocketChannel) key.attachment();
        String message = "これは書き込みテスト用のメッセージ";
        bb = charset.encode(message);
        // bbのリミットを指定してやる
        bb.flip();
        int nbytes = channel.write( bb );
    }

```

読み出しも同様である:

```

ByteBuffer bb = ByteBuffer.allocate(BUF_SIZE);
Charset charset = Charset.forName("Windows-31J");
// ここまでは最初に用意しておく
    else if ( key.isReadable() ) {
        SocketChannel channel = (SocketChannel) key.attachment();
        int nbytes = channel.read( bb );
        CharBuffer cb = charset.decode( bb );
        String message = cb.toString();
    }

```

#### 5.4.4 NonBlockingServer のプログラム

以下は非ブロッキング化したエコーバック・サーバのプログラム例である。

```

package TCPIP_Programming;

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.channels.spi.*;
import java.nio.charset.*;
import java.net.*;
import java.util.*;

public class NonBlockingServer {
    static int port = 1000;          // 待ち受けポート番号
    private Selector selector;      // セレクタ
    int keysAdded = 0;              // 追加されたキーの数
    SelectionKey acceptKey;        // アクセプトのキー

    public static void main(String[] args) {
        NonBlockingServer nonblockingServer = new NonBlockingServer();
        nonblockingServer.start();
    }
}

```

```

public void start() {
    try{
        selector = SelectorProvider.provider().openSelector();
        // セレクタの生成
        ServerSocketChannel channel = ServerSocketChannel.open();
        // channel のオープン (このクラスのコンストラクタも protected となっている)
        channel.configureBlocking(false);
        // 先ずこのチャンネルを非ブロック・モードにする
        String lh = "localhost";
        // ここではループバック・アドレスを使用
        // InetAddress lh = InetAddress.getLocalHost();
        // あるいはローカル・ホストの IP アドレスを取得
        InetSocketAddress isa = new InetSocketAddress(lh, port);
        // ポートとともにソケット・アドレスを生成
        channel.socket().bind(isa);
        // そのソケット・アドレスをその channel のソケットにバインド
        acceptKey = channel.register( selector, SelectionKey.OP_ACCEPT );
        // このチャンネルをセレクタに登録しアクセプトするキーを取得
    }
    catch(UnknownHostException e) {
        System.out.println("ローカル・ホスト不明");
        e.printStackTrace();
        System.exit(-1);}
    catch(IOException e){
        System.out.println("起動時 IOException 発生");
        e.printStackTrace();
        System.exit(-1);}

    // ここからループ
    try {
        while (( keysAdded = acceptKey.selector().select() ) > 0 ) {
            // 入出力操作の実行が可能な対応するチャンネルを持つキーセットがある限り
            Set<SelectionKey> readyKeys = selector.selectedKeys();
            // このセレクタの選択されたキーセットを一括取得し、調べる
            Iterator<SelectionKey> i = readyKeys.iterator();
            while (i.hasNext()) {
                SelectionKey key = (SelectionKey)i.next();
                // Iterator から要素を取り出す
                i.remove();
                // 取り出した要素は Iterator から削除
                if (key.isAcceptable()) {
                    doAccept((ServerSocketChannel) key.channel());
                    // Acceptable なら ServerSocketChannel にキャストしてクライアント接続待ちに
                } else if (key.isReadable()) {
                    SocketChannel channel = (SocketChannel)key.channel();
                    doRead(channel);
                    // Readable なら SocketChannel にキャストして読み出しとエコーバック
                }
            }
        }
    }
    catch(ClosedSelectorException e){
        System.out.println("セレクタが閉じている");
        e.printStackTrace();
    }
}

```

```

    }catch(IOException e){
        System.out.println("IOエラー発生");
        e.printStackTrace();
    }
}

// クライアントが接続してきたときの Accept 処理
private void doAccept(ServerSocketChannel serverChannel) {
    try {
        SocketChannel channel = serverChannel.accept();
        // このチャンネルのソケットに対する接続を受け付ける
        Socket socket = channel.socket();
        SocketAddress client_address = socket.getRemoteSocketAddress();
        System.out.println("Start serving : " + client_address);
        channel.configureBlocking(false);
        // このチャンネルを非ブロック・モードに設定
        channel.register(selector, SelectionKey.OP_READ);
        // 読み書きのためにこのセレクトタに登録
    } catch (IOException ioe) {
        ioe.printStackTrace();
    }
}

// Read 処理
private void doRead(SocketChannel channel) {
    ByteBuffer buf = ByteBuffer.allocate(500); // 500バイトの新しいバイト・バッファを用意
    Charset charset = Charset.forName("Windows-31J");
    Socket socket = channel.socket();
    SocketAddress client_address = socket.getRemoteSocketAddress();
    int bytes = 0;
    try {
        bytes = channel.read(buf);
        if (bytes < 0) {
            // チャンネルがストリームの終わりに達した場合は -1 が返る
            return;
        }
        if (bytes == 0){
            // クライアントが TCP 接続を解放すると null が返る
            System.out.println(client_address + " : Disconnected");
            return;
        }
        // 正常データ時
        buf.flip();
        boolean end = false;
        String message = charset.decode(buf).toString();
        if (message.startsWith("end") == true){
            // クライアントがセッション終了のために"end"を送ってきた
            end = true;
            System.out.print("Disconnection requested : " + client_address + " : " + message );
        }else{
            System.out.print("Received from : " + client_address + " : " + message);
            message = "Server received (" + message.length() + " characters): " + message ;
        }
        // ターミナータ付エコーバックのメッセージを用意する
    }
}

```



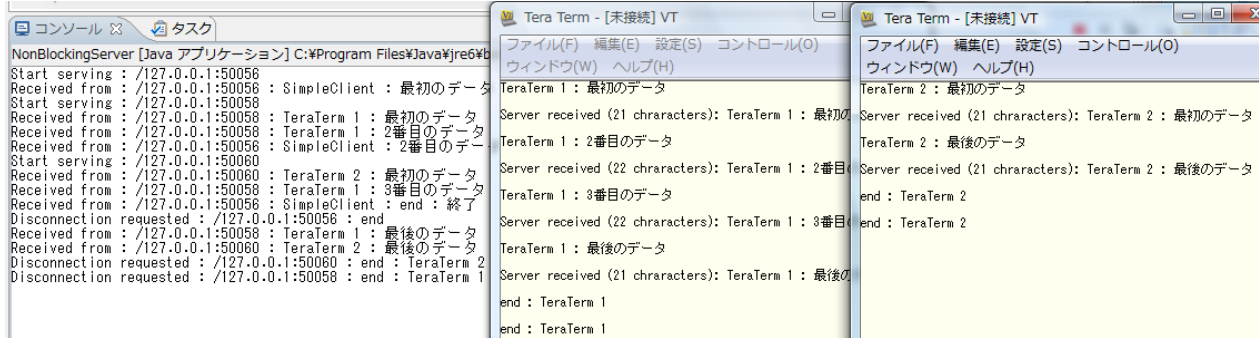
```

buf.clear();
buf = charset.encode(message); // 指定エンコーディングでバイト列化
bytes = channel.write(buf);
// これをソケット・チャンネルに書き込む
if (end){
    // endが最初のメッセージは送り返し後そのチャンネル（接続）をクローズ
    try{
        channel.close();
    }catch (IOException ee){
        System.out.println("チャンネル・クローズ時でIOエラー発生");
        ee.printStackTrace();
    }
}
return;
} catch (IOException e) {
    System.out.println("エコー・バック時でIOエラー発生");
    e.printStackTrace();
    try {
        socket.close();
    }catch (IOException ee){}
}
}
}
}

```

このサーバはスレッドが使われていないことに注意されたい。スレッドを使わなくても複数のクライアントにきちんと対応していることをクライアントのプログラムや Tera Term を複数走らせて確認できる。下図は SimpleClient を 1 個、Tera Term を 2 個同時に走らせたもので、相互干渉することなく動作していることを示している。

図 5-2: NonBlockingServer の動作確認



セレクタがチャンネルを監視する為、ソケットに関する例外は総て IOException として扱われているので、詳細はスタック・トレースを見なければならなくなる。また以前述べたようにクライアントが接続をしたまま切のを忘れてしまった場合などの為の別スレッドを使った監視タイマも、アプリケーションによっては必要になることに注意されたい。

5.4.5 実用に耐えるサーバ

最近 SourceForge で [NIO サーバのプロジェクト](#) がたちあげられている。これはイギリスのゲーム・アプリケーションのコンサルタントの Adam Martin が高負荷オンライン・ゲーム用として開発したもので、無料で使用でき、実用に十分耐えるものだと称している。彼はインターネット上の [\(O'Reilly のもの](#) を含めて) プログラムの多くは実用に堪えないと主張している。ソース・コードも総て公開されているので、興味がある人は参考にされたい。

彼が挙げている特徴は:

- 単スレッドの非ブロッキングのサーバを数分で実装できる
- 完全なこのドキュメントを加筆して必要に応じカスタマイズできる
- ソース・コードを読みまたこれがどう作られているかを理解して、自分のサーバを最初から作り上げることができる

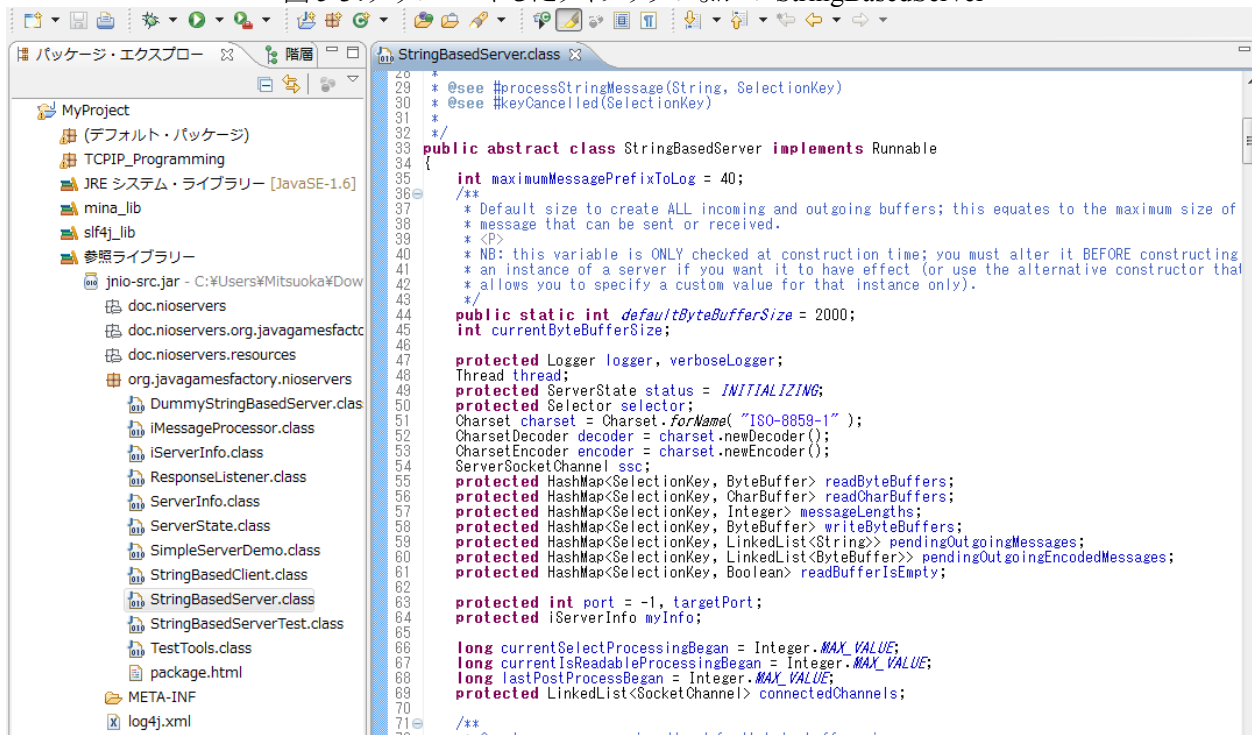
としている。

利用するには、SourceForge の [ダウンロードのリンク](#) からダウンロードした jnio-src.jar というファイルを、Eclipse の外部参照ライブラリとしてビルド・パスに取り込み、そのなかの org.javagamesfactory.nioservers というパッケージの StringBasedServer.class をエディタで見れば良い (790 行にもなっているが・・・)。

なおこのサーバは、6.3 節で説明している log4j というロギング・システムが使われている。またこのサーバは String ベースであるが、そのほうが殆どのアプリケーションで使うことが出来、またデバッグが容易である。

筆者はしかし、次の章の Apache MINA が、より使い勝手が良いフレームワークだと考えている。

図 5-3: ダウンロードしたライブラリの中への StringBasedServer



## 第6章 Apache MINA

[Apache MINA](#) (「アパッチ・ミーナ」と発音する。Multipurpose Infrastructure for Network Applications: ネットワーク・アプリケーションのための多目的インフラストラクチャの略称)は、オープン・ソースのネットワークアプリケーションのフレームワークである。ネットワークとプロトコル I/O 層の抽象化により、低レベルの NIO よりは高品質なネットワーク・アプリケーションをずっと容易に開発出来るようになっている。MINA は Java NIO (`java.nio`)を介した TCP/IP と UDP/IP だけでなく、いろんなトランスポート上での抽象化されたイベント・ドリブンの非同期(スレッドが接続、開放、送信、受信などの事象を待たない)の API を用意している。

なお同じようなフレームワークとして [JBoss Netty](#) (主開発者は同じ)がある。興味があるひとはそちらのサイトも見られたい。どちらも遜色はないが、ここではよりオープンで使用実績もある Apache MINA を紹介する。

以下は MINA プロジェクトが挙げている特徴である:

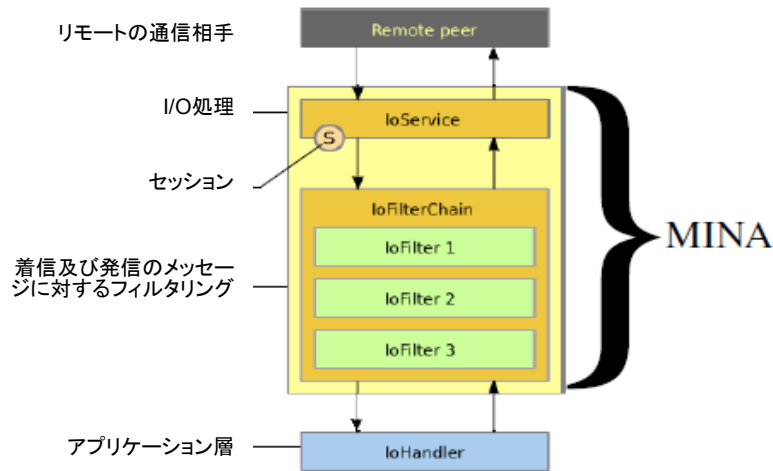
- いろんなトランスポートに対処する統一化された API
  - Java NIO を介した TCP/IP と UDP/IP
  - RxTx を介したシリアル通信(RS-232)
  - VM 内パイプ通信
  - 独自のものも実装可能
- 拡張点としてのフィルタ・インターフェイス(サブレットのフィルタ類似)
- 低レベルと高レベルの API
  - 低レベルでは `ByteBuffer` を使用
  - 高レベルではユーザが定義したメッセージのオブジェクトとコーデック
- スレッド・モデルの高度なカスタマイズ
  - 単一スレッド
  - ひとつのスレッド・プール
  - 複数のスレッド・プール(即ち SEDA)
- Java 5 `SSLEngine` を使った外部 SSL、TLS、StartTLS 対応
- 過負荷遮蔽とトラフィック制限
- モック・オブジェクトを使った単体テスト可能
- JMX 管理性
- `StreamIoHandler` を介したストリーム・ベースの I/O 対応

### 6.1節 Apache MINA のコンセプト

下図のように MINA はリモートの通信相手とアプリケーション層(`IoHandler`)間の IO 処理をこなしてくれる。アプリケーションの開発者は、`IoHandler` で用意されているメソッドをオーバーライドすればよいことになる。`IoHandler` はサブレットに似たイメージであり、いわゆるイベント・ドリブン型で、送信、受信、接続といった

I/O のいろんなイベント発生に対処するメソッドをオーバーライドすることで、アプリケーション開発が可能となる。MINA の構造は IO 処理の `IoService` と、サーブレットのフィルタ・チェーンに似た `IoFilterChain` で構成されている。また TCP の接続に対応したセッションが用意されている。このようなイメージはサーブレット・エンジンと似ており、HTTP (これは既に多くのプラットフォームが存在している) 以外のアプリケーション・プロトコルの開発にはきわめて便利で、また TCP/UDP の場合は `java.nio` をベースにしているものの、`java.nio` に比べて非常にすっきりしたものとなっている。

図 6-1: MINA のコンセプト



例えば簡単なエコーバックのサーバを考えてみよう。サーバが TCP でクライアント接続を受け付ける為には `IoService` を実装した `NioSocketAcceptor` というアクセプタを使う。このアクセプタにプログラム開発者が作成するアプリケーション、ここでは `EchoProtocolHandler` をバインドする。次にこのアクセプタに待ちうけのポート `InetSocketAddress` をバインドすると、サーバがクライアントからの接続を受け付け可能となる。メインのスレッドはソケットポート番号と IP アドレスのバインド後はなにもする必要がなく、`SocketAcceptor` が自分のスレッドでクライアントとの接続を待ち受ける。

```
public static void main(String[] args) throws Exception {
    SocketAcceptor acceptor = new NioSocketAcceptor();

    // Bind
    acceptor.setHandler(new EchoProtocolHandler());
    acceptor.bind(new InetSocketAddress(PORT));

    System.out.println("Listening on port " + PORT);
}
```

このエコーバックのアプリケーションは次のようになる:

```
public class EchoProtocolHandler extends IoHandlerAdapter {

    /**
     * 着信メッセージ処理をここに書く
     */
}
```

```

public void messageReceived( IoSession session, Object message)
throws Exception {

    // 着信メッセージを通信相手に送り返す
    session.write(((IoBuffer) message).duplicate());
}
}

```

たったこれだけで複数のクライアントに同時対応できるマルチスレッドのサーバが書けてしまうことになる。

MINA は接続のインスタンスである `IoSession` オブジェクトと通信データの `Object` を渡してくれる。このセッションが接続の識別に使われる。このようなイメージは非ブロッキング・ソケット I/O よりは理解しやすいし、サーブレットなどを書いたプログラマには相性が良いものだろう。MINA がサーブレット・エンジンに相当し、`IoHandlerAdapter` がサーブレットに相当する。**サーブレットと同じく `IoHandlerAdapter` には複数のスレッドがアクセスすることになるので、基本的なスレッドの知識が必要である。**

フィルタでは、例えば

```

acceptor.getFilterChain().addLast( "Logger", new LoggingFilter() );

```

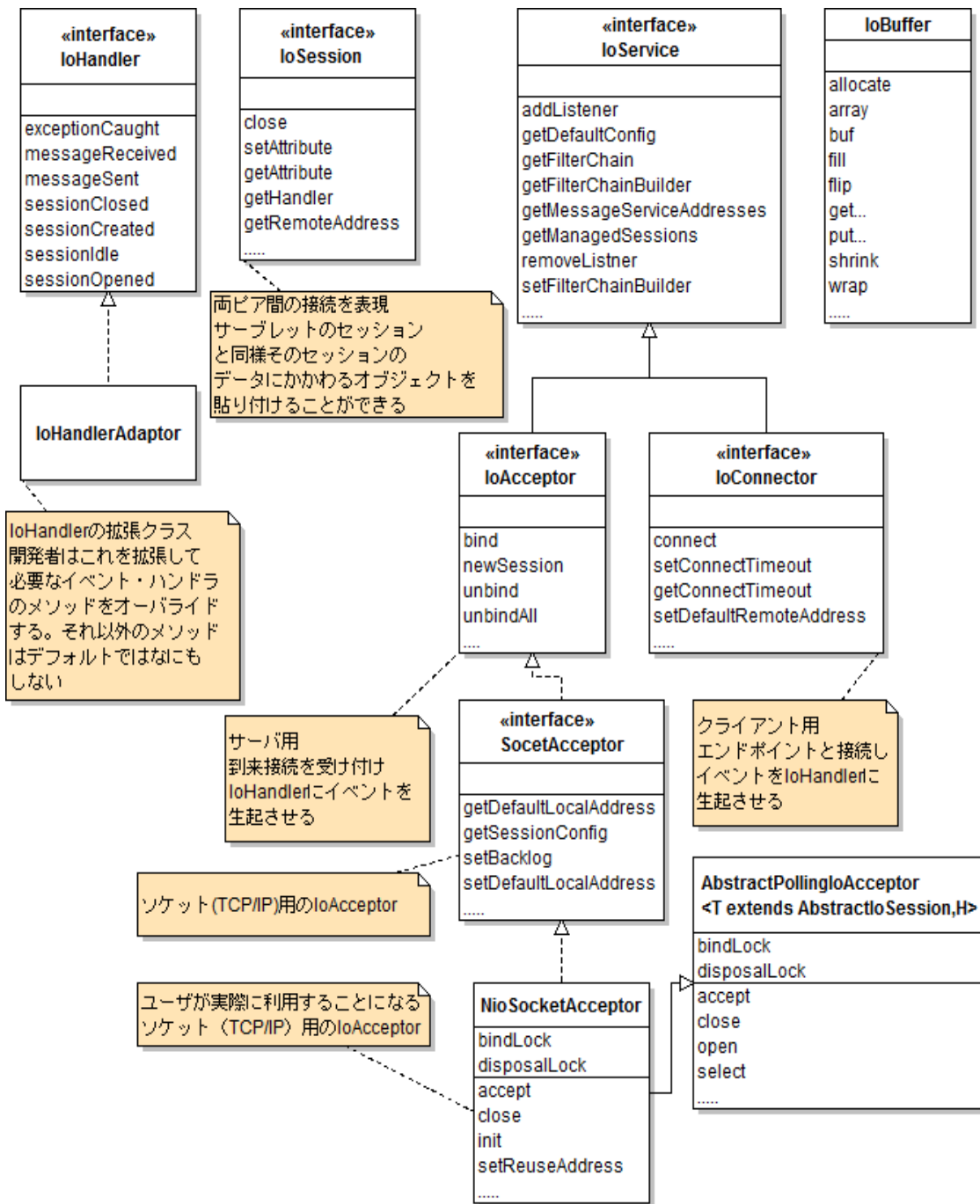
と書くだけで、ログのためのフィルタが付加できる。フィルタとしてはこれ以外にも過負荷防止の帯域制御、プロトコル CODEC (コーデック)、圧縮、マルチ・スレッド (`ExecutorFilter`) などがある。CODEC は TCP がやり取りしているバイト列をアプリケーション層が使いやすいオブジェクトの形に変換 (あるいはその逆) をするものである。現在いろんなプロトコル CODEC が MINA で用意されている。開発者はまた、自分でフィルタを設計することもできる。

## 6.2節 基本的なクラスとインターフェイス

MINA の Javadoc は <http://mina.apache.org/report/trunk/apidocs/> から取得できる。日本語の Javadoc は未完ではあるが <http://sardine.github.com/mina-ja/docs/apidocs/> から取得できる。

我々が使うことが多いと思われる基本的なインターフェイスとクラスを下図にしめす。初期のバージョンからやや複雑になってしまっているのが難点ではある。

図 6-2: 基本的なインターフェイスとクラス



なおソケット (TCP/IP) 用に org.apache.mina.transport.socket.nio.NioSocketAcceptor というクラスが用意されており、これは IoAcceptor を実装した非同期モードのクラスであり、良く使われることになる。このクラスのコンストラクタにはデフォルトの数のスレッドのものと、必要な数のスレッドが用意されるものがある。

## 6.2.1 IoHandlerAdapter

IoHandlerAdapter は MINA で生起されたイベントたちを処理する IoHandler インターフェイスの抽象アダプタ・クラスである。IoHandler はフィルタ・チェーンの終点でもある。用意されているメソッドは総て IoHandler インターフェイスのメソッドである。

主要なメソッドは以下のものがある：

- sessionOpened: 接続が確立した
- messageReceived: 相手からのメッセージを受信した
- sessionClosed: 接続が閉じた

プログラム開発者はこのクラスを継承し、選択的に必要なイベント・ハンドラのメソッドをオーバーライドする。これはサーブレットの開発者たちにはなじみの手法である。総てのメソッドはデフォルトでは何もしないでただ返るだけである。また総てのメソッドが IoSession のオブジェクトを返すので、開発者はそれによりクライアントを識別でき、クライアントごとに一貫したサービスを提供するようにできる。このクラスはサーブレットによく似ているので、ウェブの技術者にはなじみが良いと思われる。サーブレットの場合は(要求、応答)の二つのオブジェクトで渡されるが、IoHandlerAdapter では TCP では接続がベースになっているので、総てのメソッドで session オブジェクトが渡されるが、受信したメッセージはオブジェクトとして、送信するメッセージは session のメソッドを使うという方法をとっている。どうして送信と受信で非対称な取り扱いになっているのかわからない。ここが少しすっきりしない点である。送受信されるメッセージはフィルタで加工され、いろんなクラスのオブジェクトになっている可能性があるため、オブジェクトとして渡されており、受信処理ではオーバーライドするメソッドの中で message を必要なクラスにキャストする。

なお IoHandlerAdapter の各メソッドには複数クライアントに対応したスレッドが到来するのは、サーブレットと同じであり、クラス変数を使わないなど、**スレッド安全なコードとしなければならない**。従って、その**クライアントに固有なデータはセッションに貼り付けることになる**。また**各メソッド内でタイマーや別の事象の待ち受けがあると、そのクライアントの次のデータ処理が待たされることになるので、注意が必要**である。折角非ブロッキングで行われている I/O 処理がそのような事象でブロックされてしまうのは好ましくない。

スレッド・モデルのデザインに関しては「[Apache MINA のスレッド](#)」の節で詳しく説明する。

表 6-1 : IoHandlerAdapter クラスのコンストラクタとメソッド

コンストラクタ	
public IoHandlerAdapter()	
メソッド	
public void sessionCreated (IoSession session) throws Exception	新しい接続が確立されたときに I/O プロセッサのスレッドがこのメソッドを呼び出す。このメソッドは複数のセッションの I/O を処理する同一スレッドで呼ばれることが想定されているので、このメソッドの実装に当たってはソケットのパラメタとユーザが定義したセッション属性の初期化といった最小限の時間を消費するタスクを実行するようにされたい。
public void sessionOpened (IoSession session) throws Exception	ある接続が開いたときに呼び出される。このメソッドは IoHandler.sessionCreated(IoSession)のあとで呼び出される。IoHandler.sessionCreated(IoSession)との最大の相違点は、スレッドのモデルが適正に設定されれば、I/O プロセッサのスレッド以外のスレッドによっても呼び出されることである。
public void sessionClosed (IoSession session) throws Exception	ある接続が閉じたときに呼び出される。
public void sessionIdle (IoSession session,	ある接続がアイドル状態になったときに関連する IdleStatus とともに呼び出される。このメソッドはそのトランスポート・タイプが UDP のときには呼び出されなないという知られた

IdleStatus status) throws Exception	バグがあり、2.0 版ではこれは修正される予定である。
public void <b>exceptionCaught</b> (IoSession session, Throwable cause) throws Exception	ユーザが実装した IoHandler あるいは MINA が何らかの例外をスローしたときに呼び出される。その原因が IOException のときは、MINA はその接続を自動的に閉じる。
public void <b>messageReceived</b> (IoSession session, Object message) throws Exception	あるメッセージを受信したときに呼び出される。
public void <b>messageSent</b> (IoSession session, Object message) throws Exception	IoSession.write(Object)で書かれたあるメッセージが送信されたときに呼び出される。

## 6.2.2 IoSession

これはトランスポートのタイプにかかわらず 2 つのポイント間の接続 (たとえば TCP ではある TCP 接続) を表現したハンドルである。IoSession はユーザが定めた属性を持つことができる。ユーザが定めた属性はあるセッションに関わるアプリケーション固有のデータである。これはしばしばハイレベルのプロトコルの状態を表現するオブジェクトたちを含み、フィルタたちとハンドラたち間のデータ交換のひとつの手段となる。ハンドラにはイベント毎にこのオブジェクトが渡される。

主要なメソッドとしては以下のものがある:

- write: クライアントに送信するデータの書き込み
- close: その接続を閉じる
- get/setAttribute: 属性のセットとゲット

トランスポートの型固有の属性に合わせるために、このセッションを適切なサブクラスにダウンキャストすることができる。

読み書きとクローズは非同期処理であり、その為に ReadFuture、WriteFuture、CloseFuture という通知のためのインターフェイスが用意されている。

このクラスはスレッド安全であるが、同時に複数の write(Object)呼び出しを行うと、

IoFilter.filterWrite(IoFilter.NextFilter, IoSession, WriteRequest)が同時に実行されることになり、自分が使っている IoFilter の実装がスレッド安全であるようになっていなければならないことに注意が必要である。

表 6-2: IoSession クラスのメソッド

メソッド	
long getId()	戻り値: このセッションのための独自の識別子。各セッションは相互の異なる自分の ID を保持している。 注意: 実際の実装では上記の契約が遵守されていることが保証されていない。キーの一意性が保証されていない hashCode() メソッドが使われていることによる。



<b>IoService getService()</b>	戻り値: このセッションに I/O サービスを提供する IoService。
<b>IoHandler getHandler()</b>	戻り値: このセッションを処理する IoHandler。
<b>IoSessionConfig getConfig()</b>	戻り値: このセッションの設定。
<b>IoFilterChain getFilterChain()</b>	戻り値: このセッションに影響を与えるフィルタ・チェーン。
<b>WriteRequestQueue getWriteRequestQueue()</b>	ドキュメント作業中
<b>TransportMetadata getTransportMetadata()</b>	戻り値: このセッションを稼働させる TransportMetadata。
<b>ReadFuture read()</b>	TODO This javadoc is wrong. The return tag should be short.  戻り値: 新しいメッセージの受信、接続の切断、例外のキャッチが発生したときに通知する ReadFuture を返す。この操作はクライアントアプリケーションを実装する際に特に便利である。TODO:この個所にこの機能を有効にする方法を説明する。ただし、この操作はデフォルトでは無効になっており、IllegalStateException をスローする。これは、この操作をサポートするためには受信したイベントをすべてキューに蓄積しなければならない、メモリークを起す可能性があるためである。つまり、この操作を有効にした場合は、read() を呼び続けなければならないということの意味する。この操作を有効にするには、IoSessionConfig.setUseReadOperation(boolean) に true を渡す。 例外: java.lang.IllegalStateException - useReadOperation オプションが有効になっていない場合。
<b>WriteFuture write(Object message)</b>	指定されたメッセージをリモートのピアに書き込む。この操作は非同期である:このメッセージが実際にリモートにピアに送信された際は IoHandler.messageSent(IoSession, Object) が呼び出される。このメッセージが実際に書き込まれたのを待ちたいときには、WriteFuture が戻されるのを待つこともできる。
<b>WriteFuture write(Object message, SocketAddress destination)</b>	(オプション) 指定されたメッセージ message を指定された送信先 destination に書く。この操作は非同期である。メッセージが実際にリモートピアに向けて書き出されると、IoHandler.messageSent(IoSession, Object) が呼び出される。メッセージが実際に書かれるまで待ちたい場合は、WriteFuture が返されるのを待機することもできる。 DHCP サーバなどのサーバからのブロードキャストメッセージを受信するクライアントを実装する場合、クライアントはサーバが送信したブロードキャストメッセージに対して応答メッセージを送信しなければならないことがある。ブロードキャストの場合、セッションのリモートアドレスはサーバのアドレスではないので、応答メッセージを書き出すときに送信先を指定する方法が必要になる。このインタフェースでは送信先を指定できるように write(Object, SocketAddress) メソッドを提供している。 パラメタ: destination - デフォルトのリモート・アドレスにメッセージを送信したい場合は null 例外: java.lang.UnsupportedOperationException - この操作がサポートされていない場合。
<b>CloseFuture close(boolean immediately)</b>	このセッションをすぐに、あるいは、キューに蓄積されている書き出し要求がすべてフラッシュされた後に、クローズする。この操作は非同期である。セッションが実際にクローズされるまで待ちたい場合は CloseFuture が返されるのを待つ。 パラメタ: immediately - このセッションをすぐにクローズしたい場合 (つまり close()) は true。キューに蓄積されている書き出し要求がすべてフラッシュされた後にクローズしたい場合(つまり #closeOnFlush()) は false。
<b>Object getAttribute(Object key)</b>	このセッションのユーザが決めた属性の値を返す。

	<p>パラメタ:  <b>key</b> - この属性のキー</p> <p>戻り値:  指定されたキーを持った属性がないときは <b>null</b> を返す。</p>
Object <b>getAttribute</b> (Object key, Object defaultValue)	<p>指定されたキーに結び付けられたユーザが決めた属性の値を返す。そのような属性がないときは、指定されたデフォルト値は指定されたキーに結び付けられたものであり、そのデフォルト値が返される。このメソッドはその操作がアトミックになされることを除けば以下のコードと同じである:</p> <pre>if (containsAttribute(key)) {     return getAttribute(key); } else {     setAttribute(key, defaultValue);     return defaultValue; }</pre>
Object <b>setAttribute</b> (Object key, Object value)	<p>ユーザが指定した属性をセットする。</p> <p>パラメタ:  <b>key</b> - その属性のキー  <b>value</b> - その属性の値</p> <p>戻り値:  その属性のもとの値、新規の場合は <b>null</b></p>
Object <b>setAttribute</b> (Object key)	<p>ユーザが指定した属性を値なしでセットする。ある「マーク」をしたいときにはこれは有用である。その値は <b>Boolean.TRUE</b> がセットされる。</p> <p>パラメタ:  <b>key</b> - その属性のキー</p> <p>戻り値:  その属性のもとの値、新規の場合は <b>null</b></p>
Object <b>setAttributeIfAbsent</b> (Object key, Object value)	<p>もし指定されたキーがまだセットされていないときにユーザが決めた属性をセットする。このメソッドはその操作がアトミックに実行されることを除いては次のコードと同じである:</p> <pre>if (containsAttribute(key)) {     return getAttribute(key); } else {     return setAttribute(key, value); }</pre>
Object <b>setAttributeIfAbsent</b> (Object key)	<p>もし指定されたキーがまだセットされていないときにユーザが決めた属性を値なしでセットする。ある「マーク」をしたいときにはこれは有用である。その値は <b>Boolean.TRUE</b> がセットされる。このメソッドはその操作がアトミックに実行されることを除いては次のコードと同じである:</p> <pre>if (containsAttribute(key)) {     return getAttribute(key); // might not always be Boolean.TRUE. } else {     return setAttribute(key); }</pre>
Object <b>removeAttribute</b> (Object key)	<p>指定されたキーを有するユーザが決めた属性を削除する。</p> <p>戻り値:  その属性の以前の値。もし見つからないときは <b>null</b></p>
boolean <b>removeAttribute</b> (Object key, Object value)	<p>現行の属性値が指定された値と等しいときは指定されたキーを持ったユーザ定義の属性を削除する。このメソッドは操作がアトミックに行われることを除いては次のコードと等価である:</p> <pre>if (containsAttribute(key) &amp;&amp; getAttribute(key).equals(value)) {</pre>

	<pre> removeAttribute(key); return true; } else { return false; } </pre>
boolean <b>replaceAttribute</b> (Object key, Object oldValue, Object newValue)	<p>現行の属性値が指定された古い値と等しいときは指定されたキーを持ったユーザ定義の属性値を新しい値に置き換える。このメソッドは操作がアトミックに行われることを除いては次のコードと等価である:</p> <pre> if (containsAttribute(key) &amp;&amp; getAttribute(key).equals(oldValue)) { setAttribute(key, newValue); return true; } else { return false; } </pre>
boolean <b>containsAttribute</b> (Object key)	このセッションが指定されたキーをもった属性を含むときは <b>true</b> を返す。
Set<Object> <b>getAttributeKeys</b> ()	ユーザ定義の総ての属性のキーのセットを返す。
boolean <b>isConnected</b> ()	このセッションがリモートのピアと接続しているときは <b>true</b> を返す。
boolean <b>isClosing</b> ()	このセッションがクローズ中であるとき(まだ接続が切断されていない)、あるいはクローズしてしまっているときにのみ <b>true</b> を返す。
CloseFuture <b>getCloseFuture</b> ()	このセッションの <b>CloseFuture</b> を返す。このメソッドはユーザが呼んだときはいつでも同じインスタンスを返す。
SocketAddress <b>getRemoteAddress</b> ()	リモート・ピアのソケット・アドレスを返す。
SocketAddress <b>getLocalAddress</b> ()	今セッションに関わっているローカル・マシンのソケット・アドレスを返す。
SocketAddress <b>getServiceAddress</b> ()	このセッションを管理する為に <b>IoService</b> がリスンしているソケット・アドレスを返す。もしこのセッションが <b>IoAcceptor</b> によって管理されているときには、 <b>IoAcceptor.bind()</b> の絡めたとして指定されている <b>SocketAddress</b> を返す。このセッションが <b>IoConnector</b> によって管理されているときには、 <b>getRemoteAddress()</b> と同じアドレスを返す。
void <b>setCurrentWriteRequest</b> (WriteRequest currentWriteRequest)	<p>TODO setWriteRequestQueue.  パラメタ:  writeRequestQueue -</p>
void <b>suspendRead</b> ()	このセッションの読み出し操作を一時停止する。
void <b>suspendWrite</b> ()	このセッションの書き込み操作を一時停止する。
void <b>resumeRead</b> ()	このセッションの読み出し操作を再開する。
void <b>resumeWrite</b> ()	このセッションの書き込み操作を再開する。
boolean <b>isReadSuspended</b> ()	このセッションの読み出し操作が一時停止中かどうか。I 戻り値: もし一時停止中は <b>true</b>
boolean <b>isWriteSuspended</b> ()	このセッションの書き込み操作が一時停止中かどうか。 戻り値: 一時停止中は <b>true</b>
void <b>updateThroughput</b> (long currentTime, boolean force)	指定された時間が現在の時間だとみなして、スループットに関わる総ての統計値を更新する。デフォルトではこのメソッドは、既に最後の算出インターバルでこれらが算出済みのときは、スループットの値を更新することなくそのまま戻る、そのまま戻る。しかしながら、もし <b>force</b> が <b>true</b> と指定されているときは、このメソッドは即座にスループット値を更新する。

	<p>パラメタ: currentTime - mS による現在時間 t</p>
long <b>getReadBytes()</b>	このセッションで読み出されたバイトの総数を返す。
long <b>getWrittenBytes()</b>	このセッションに書き出されたバイトの総数を返す。
long <b>getReadMessages()</b>	このセッションから読み出されデコードされたメッセージの総数を返す。
long <b>getWrittenMessages()</b>	このセッションで書きこまれたエンコードされたメッセージの総数を返す。
double <b>getReadBytesThroughput()</b>	毎秒あたりの読み出しバイト数を返す。
double <b>getWrittenBytesThroughput()</b>	毎秒あたりの書き込みバイト数を返す。
double <b>getReadMessagesThroughput()</b>	毎秒あたりの読み出しメッセージ数を返す。
double <b>getWrittenMessagesThroughput()</b>	毎秒あたりの書き込みメッセージ数を返す。
int <b>getScheduledWriteMessages()</b>	このセッションで書き込まれることになっているメッセージ数を返す。
Object <b>getCurrentWriteMessage()</b>	IoService によって書き込み中のメッセージを返す。 戻り値: 書き込み中のメッセージがないときに限り null
WriteRequest <b>getCurrentWriteRequest()</b>	IoService によって処理中の WriteRequest を返す。 戻り値: 書き込み中のメッセージがないときに限り null
long <b>getCreationTime()</b>	戻り値: mS でのこのセッションの生成時刻
long <b>getLastIoTime()</b>	mS での I/O が最後に起きた時刻
long <b>getLastReadTime()</b>	mS での読み出し操作が最後に起きた時刻
long <b>getLastWriteTime()</b>	mS での書き込み操作が最後に起きた時刻
boolean <b>isIdle</b> (IdleStatus status)	このセッションが指定された IdleStatus のアイドル状態であるときに true を返す。
boolean <b>isReaderIdle()</b>	このセッションが IdleStatus.READER_IDLE のときに true を返す。 参照: isIdle(IdleStatus)
boolean <b>isWriterIdle()</b>	このセッションが IdleStatus.WRITER_IDLE 状態であるときに true を返す。 参照: isIdle(IdleStatus)
boolean <b>isBothIdle()</b>	このセッションが IdleStatus.BOTH_IDLE のときに true を返す。 参照: isIdle(IdleStatus)
int <b>getIdleCount</b> (IdleStatus status)	指定された IdleStatus で、連続的な sessionIdle イベントが生じた数を返す。もし I/O のあと暫くして最初に sessionIdle イベントが生じたときは、idleCount は 1 になる。何らかの I/O が起きたときは idleCount は 0 にリセットされ、それ以外は 2 つ (あるいはそれ以上) の sessionIdle イベント間で I/O が起きることなく sessionIdle イベントが起きると 2、そしてさらに増加してゆく。
int <b>getReaderIdleCount()</b>	IdleStatus.READER_IDLE の sessionIdle イベントが連続して生じた数を返す。 参照: getIdleCount(IdleStatus)
int <b>getWriterIdleCount()</b>	IdleStatus.WRITER_IDLE の sessionIdle イベントが連続して生じた数を返す。

	参照: getIdleCount(IdleStatus)
int getBothIdleCount()	IdleStatus.BOTH_IDLE の sessionIdle イベントが連続して生じた数を返す。 参照: getIdleCount(IdleStatus)
long getLastIdleTime(IdleStatus status)	指定された IdleStatus での sessionIdle イベントが最後に生じた時刻を mS で返す。
long getLastReaderIdleTime()	IdleStatus.READER_IDLE での sessionIdle イベントが最後に生じた時刻を mS で返す。 参照: getLastIdleTime(IdleStatus)
long getLastWriterIdleTime()	IdleStatus.WRITER_IDLE での sessionIdle イベントが最後に生じた時刻を mS で返す。 参照: getLastIdleTime(IdleStatus)
long getLastBothIdleTime()	IdleStatus.BOTH_IDLE での sessionIdle イベントが最後に生じた時刻を mS で返す。 参照: getLastIdleTime(IdleStatus)

### 6.2.3 IoService

これは I/O サービスを提供し、IoSession を管理している総ての IoAcceptor (サーバ側) と IoConnector (クライアント側) たちのベースとなっているインターフェイスである。このインターフェイスは I/O 関連の操作を実行するのに必要な総ての機能を有している。MINA の[解説](#)にはこのインターフェイスの責任と実装している AbstractIoService を次のように図示している。

IoService の責任としては:

- リスナ管理
- IoHandler
- IoSession 管理
- FilterChain 管理
- 統計量管理

があり、実装している AbstractIoService の機能は:

- DefaultFilterChain の追加
- デフォルトの dispose 実装
- IoHandler のセット
- DefaultIoSessionDataStructureFactory の維持
- Executor の生成
- デフォルトの例外モニタの追加
- セッション設定の管理

などがある。

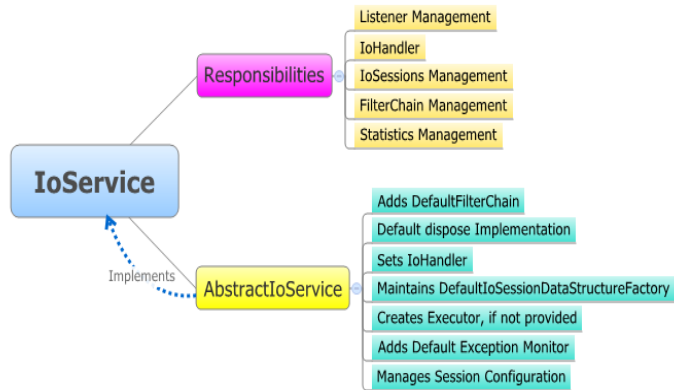


図 6-3: IoService のインターフェイスの責任と実装している AbstractIoService

IoService を実装しているサーバ用のクラスたちのクラス図は次のようである。

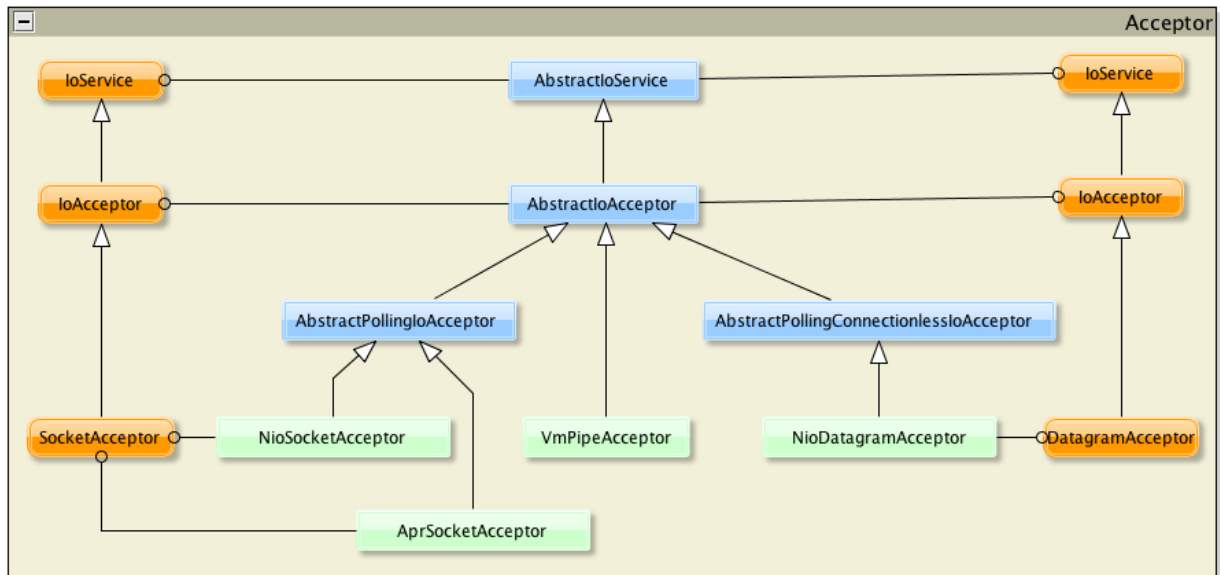


図 6-4: サーバ用の IoService のクラス図

この図では左側が TCP トラフィック用、右側が UDP トラフィック用として示されている。

ここで、

- NioSocketAcceptor : 非ブロッキングのソケット・トラフィックのアクセプタ
- NioDatagramAcceptor : 非ブロッキング UDP トラフィックのアクセプタ
- AprSocketAcceptor : APR(Apache Portable Runtime)に基づくブロッキングのソケット・トラフィックのアクセプタ
- VmPipeSocketAcceptor : Vm 内アクセプタ

で、利用者はその中から自分のアプリケーションに適切なクラスを選択する。

参考までに、クライアント用のクラス図を下図に示す。

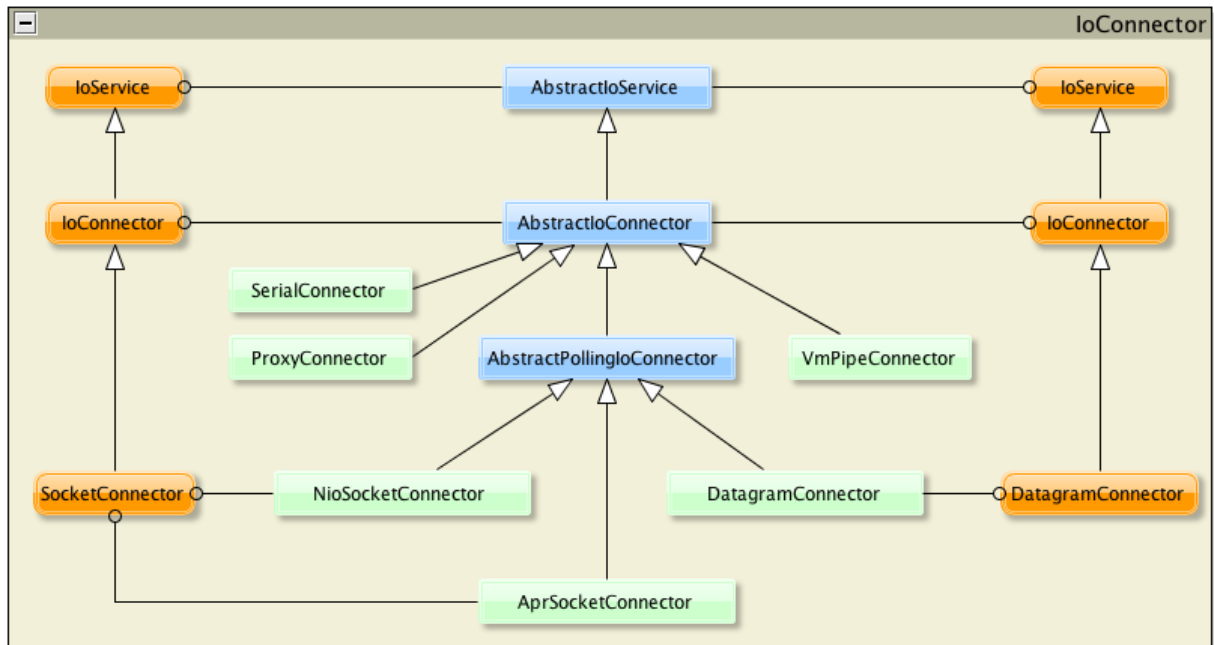


図 6-5: クライアント用の IoService のクラス図

ここで、

- NioSocketConnector : 非ブロッキングのソケット・トランスポートのコネクタ
- NioDatagramConnector : 非ブロッキングの UDP トランスポートのコネクタ
- AprSocketConnector : APR(Apache Portable Runtime)に基づくブロッキングのソケット・トランスポートのコネクタ
- ProxyConnector : プロキシ対応のコネクタ
- SerialConnector : RS-232C などのシリアル通信のためのコネクタ
- VmPipeConnector : VM 内コネクタ

で、利用者はその中から自分のアプリケーションに適切なクラスを選択する。

表 6-3: IoService インターフェイスのメソッド

メソッド	
TransportMetadata <b>getTransportMetadata()</b>	このサービスが走っている TransportMetadata を返す。
void <b>addListener</b> (IoServiceListener listener)	このサービスに関連した何らかのイベントをリスンする IoServiceListener を付加する。
void <b>removeListener</b> (IoServiceListener listener)	このサービスに関連した何らかのイベントをリスンする既存の IoServiceListener を削除する。
boolean <b>isDisposing</b> ()	<b>dispose()</b> メソッドが呼ばれているときに限り true を返す。総ての関連したリソースが開放された後であっても true を返すことに注意されたい。
boolean <b>isDisposed</b> ()	このプロセサの総てのリソースが廃棄されたときに限り true を返す。
void <b>dispose</b> ()	このサービスによって割り当てられているリソースの総てを開放する。このサービスが管理しているセッションがあるとするこのメソッドはブロックする可能性があることに注意されたい。

<code>IoHandler getHandler()</code>	このサービスが管理する総ての接続を取り扱うハンドラを返す。
<code>void setHandler(IoHandler handler)</code>	このサービスが管理する総ての接続を取り扱うハンドラをセットする。
<code>java.util.Map&lt;java.lang.Long, IoSession&gt; getManagedSessions()</code>	このサービスによって現在管理されている総てのセッションのマップを返す。マップのキーはこのセッションの ID である。 戻り値: セッションたち。セッションがなければ空のコレクション
<code>int getManagedSessionCount()</code>	このサービスが現在管理している総てのセッションの数を返す。
<code>IoSessionConfig getSessionConfig()</code>	このサービスが生成した新しい <code>IoSession</code> のデフォルトの設定を返す。
<code>IoFilterChainBuilder getFilterChainBuilder()</code>	このサービスが生成した総ての <code>IoSession</code> たちの <code>IoFilterChain</code> を作る <code>IoFilterChainBuilder</code> を返す。デフォルト値は空の <code>DefaultIoFilterChainBuilder</code> である。
<code>void setFilterChainBuilder(IoFilterChainBuilder builder)</code>	このサービスが生成した総ての <code>IoSession</code> たちの <code>IoFilterChain</code> を作る <code>IoFilterChainBuilder</code> をセットする。この属性に <code>null</code> を指定すると、空の <code>DefaultIoFilterChainBuilder</code> がセットされる。
<code>DefaultIoFilterChainBuilder getFilterChain()</code>	<code>(( DefaultIoFilterChainBuilder ) getFilterChainBuilder() )</code> のショートカット。返されるオブジェクトは実際の <code>IoFilterChain</code> ではなくて、 <code>DefaultIoFilterChainBuilder</code> であることに注意されたい。返されたビルダを修正することは既存の <code>IoSession</code> にはまったく影響を与えないが、これは <code>IoFilterChainBuilder</code> は新規に生成された <code>IoSession</code> にのみ効果を持つからである。 例外: 現在の <code>IoFilterChainBuilder</code> が <code>DefaultIoFilterChainBuilder</code> でないときに <code>java.lang.IllegalStateException</code> をスローする
<code>boolean isActive()</code>	このサービスがアクティブであるかどうかを返す。 戻り値: このサービスがアクティブかどうか
<code>long getActivationTime()</code>	このサービスがアクティブになった時刻を返す。このサービスが現在アクティブでないときはこのサービスがアクティブになった最後の時刻を返す。 戻り値: <code>System.currentTimeMillis()</code> を使った時刻
<code>java.util.Set&lt;WriteFuture&gt; broadcast(java.lang.Object message)</code>	このサービスが管理する総ての <code>IoSession</code> に指定されたメッセージを書く。このメソッドは <code>IoUtil.broadcast(Object, Collection)</code> の為の便利なショートカットである。
<code>void setSessionDataStructureFactory(IoSessionDataStructureFactory sessionDataStructureFactory)</code>	このサービスが生成した新規のセッションの為のデータ構造を提供する <code>IoSessionDataStructureFactory</code> をセットする。
<code>int getScheduledWriteBytes()</code>	書き込み予定のバイト数を返す。 戻り値: 書き込み予定のバイト数
<code>int getScheduledWriteMessages()</code>	書き込み予定のメッセージ数を返す 戻り値: 書き込み予定のメッセージの数
<code>IoServiceStatistics getStatistics()</code>	このサービスの為の <code>IoServiceStatistics</code> のオブジェクトを返す 戻り値: このサービスの為の統計オブジェクト

## 6.2.4 IoAcceptor



このインターフェイスは `IoService` を継承しているサーバ用のもので、到来接続要求を受け付け、クライアントたちと通信し、`IoHandler` たちにイベントを通知する。クライアント用のインターフェイスは `IoConnector` である。

到来接続を受け付けるにはまず受付ソケット・アドレス (IP アドレスと TCP ポート番号) をバインドしなければならない。つぎに到来接続のイベントがデフォルトの `IoHandler` に送信されることになる。`bind()` が呼ばれると到来接続受付の為のスレッドたちが自動的に開始され、`unbound()` が呼ばれると自動的に停止する。

このインターフェイスを実装した最終的なクラスには次のようなものがある：

表 6-4: サーバ用とクライアント用の I/O サービス・クラス

サーバ用	クライアント用	通信プロトコル
<code>NioDatagramAcceptor</code>	<code>NioDatagramConnector</code>	UDP
<code>NioSocketAcceptor</code>	<code>NioSocketConnector</code>	TCP
<code>VmPipeAcceptor</code>	<code>VmPipeConnector</code>	Pipe ベース

表 6-5: `IoAcceptor` クラスのメソッド

メソッド	
<code>SocketAddress getLocalAddress()</code>	現在バインドされているローカル・アドレスを返す。ひとつ以上のアドレスがバインドされているときは、そのうちひとつだけが返されるが、それは必ずしも最初にバインドされたものとは限らない。
<code>Set&lt;SocketAddress&gt; getLocalAddresses()</code>	現在バインドされているローカル・アドレスのセットを返す。
<code>SocketAddress getDefaultLocalAddress()</code>	<code>bind()</code> メソッドでパラメタが指定されていないときに、バインドされるデフォルトのローカル・アドレスを返す。何らかのローカル・アドレスが指定されているときにはこのデフォルトは使われないことに注意されたい。ひとつ以上のアドレスがセットされているときには、そのうちのひとつのみが返されるが、それは必ずしも <code>setDefaultLocalAddresses(List)</code> で最初に指定されているアドレスとは限らない。
<code>List&lt;SocketAddress&gt; getDefaultLocalAddresses()</code>	<code>bind()</code> メソッドでパラメタが指定されていないときにバインドされるデフォルトのローカル・アドレスのリストを返す。何らのローカル・アドレスが指定されてるときには、このデフォルトは使われないことに注意されたい。
<code>void setDefaultLocalAddress(SocketAddress localAddress)</code>	<code>bind()</code> メソッドでパラメタが指定されていないときにバインドされるデフォルトのローカル・アドレスをセットする。何らのローカル・アドレスが指定されてるときには、このデフォルトは使われないことに注意されたい。
<code>void setDefaultLocalAddresses(SocketAddress firstLocalAddress, SocketAddress otherLocalAddresses)</code>	<code>bind()</code> メソッドでパラメタが指定されていないときにバインドされるデフォルトのローカル・アドレスたちをセットする。何らのローカル・アドレスが指定されてるときには、このデフォルトは使われないことに注意されたい。
<code>void setDefaultLocalAddresses(Iterable&lt;? extends SocketAddress&gt; localAddresses)</code>	<code>bind()</code> メソッドでパラメタが指定されていないときにバインドされるデフォルトのローカル・アドレスたちをセットする。何らのローカル・アドレスが指定されてるときには、このデフォルトは使われないことに注意されたい。
<code>void setDefaultLocalAddresses(List&lt;? extends SocketAddress&gt; localAddresses)</code>	<code>bind()</code> メソッドでパラメタが指定されていないときにバインドされるデフォルトのローカル・アドレスたちをセットする。何らのローカル・アドレスが指定されてるときには、このデフォルトは使われないことに注意されたい。
<code>boolean isCloseOnDeactivation()</code>	このアクセプタが総ての関連するローカル・アドレスからバインドを外すとき (言い換えればこのサービスが停止するとき) に総てのクライアントが閉じたときのみ <code>true</code> を返す。
<code>void</code>	このアクセプタが総ての関連するローカル・アドレスからバインドを外すとき (言い換えれば

<b>setCloseOnDeactivation</b> (boolean closeOnDeactivation)	ばこのサービスが停止するとき)に、総てのクライアント・セッションが閉じるかどうかをセットする。デフォルト値は <b>true</b> である。
<b>void bind</b> () throws IOException	デフォルトのローカル・アドレス(ひとつまたは複数)にバインドし、到来接続の受付を開始する。 例外: IOException: バインド失敗時
<b>void bind</b> (SocketAddress localAddress) throws IOException	指定されたローカル・アドレスにバインドし、到来接続の受付を開始する。 例外: IOException: バインド失敗時
<b>void bind</b> (SocketAddress firstLocalAddress, SocketAddress... addresses) throws IOException	指定されたローカル・アドレスたちにバインドし、到来接続の受付を開始する。アドレスが指定されていないときは、デフォルトのローカル・アドレスにバインドする。 例外: IOException: バインド失敗時
<b>void bind</b> (Iterable<? extends SocketAddress> localAddresses) throws IOException	指定されたローカル・アドレスたちにバインドし、到来接続の受付を開始する。 例外: IOException: バインド失敗時
<b>void unbind</b> ()	このサービスがバインドしている総てのローカル・アドレスを外し、到来接続受付を停止する。disconnectOnUnbind 属性が <b>true</b> のときは、管理されている総ての接続は閉じられる。このメソッドはまだローカル・アドレスがバインドされていないときはそのまま戻る。
<b>void unbind</b> (SocketAddress localAddress)	指定されたローカル・アドレスからバインドを外し、到来接続受付を停止する。disconnectOnUnbind 属性が <b>true</b> のときは、管理されている総ての接続は閉じられる。このメソッドはまだローカル・アドレスがバインドされていないときはそのまま戻る。
<b>void unbind</b> (SocketAddress firstLocalAddress, SocketAddress... otherLocalAddresses)	指定されたローカル・アドレスからバインドを外し、到来接続受付を停止する。disconnectOnUnbind 属性が <b>true</b> のときは、管理されている総ての接続は閉じられる。このメソッドはまだローカル・アドレスがバインドされていないときはそのまま戻る。
<b>void unbind</b> (Iterable<? extends SocketAddress> localAddresses)	指定されたローカル・アドレスからバインドを外し、到来接続受付を停止する。disconnectOnUnbind 属性が <b>true</b> のときは、管理されている総ての接続は閉じられる。このメソッドはまだローカル・アドレスがバインドされていないときはそのまま戻る。
<b>IoSession newSession</b> (SocketAddress remoteAddress, SocketAddress localAddress)	(オプション)指定された localAddress と指定された remoteAddress にバインドされている IoSession を返し、これはこのサービスによって既にバインドされているローカル・アドレスを再使用するものである。 この操作はオプションなもので、この操作をサポートしていないトランスポートでは <b>UnsupportedOperationException</b> をスローすること。この操作は通常コネクションレスのトランスポートの形式で実装される。 例外: <b>UnsupportedOperationException</b> - この操作がサポートされていないとき <b>IllegalStateException</b> - このサービスが動作していないとき <b>IllegalArgumentException</b> - 指定された localAddress にこのサービスがバインドしていないとき

## 6.2.5 SocketAcceptor

これはソケット・トランスポート(即ち TCP/IP)用の **IoAcceptor** インターフェイスである。このインターフェイスは TCP/IP ベースの到来ソケット接続を処理する。このインターフェイスでは、下記のメソッド以外に継承している前記の **IoService** と **IoAcceptor** のメソッドたちが利用できる。

表 6-6: SocketAcceptor インターフェイスのメソッド

メソッド	
InetSocketAddress getLocalAddress()	現在バインドされているローカル・アドレスを返す。ひとつ以上のアドレスがバインドされているときは、そのうちひとつだけが返されるが、それは必ずしも最初にバインドされたものとは限らない。 参照: IoAcceptor の getLocalAddress
InetSocketAddress getDefaultLocalAddress()	bind()メソッドでパラメタが指定されていないときに、バインドされるデフォルトのローカル・アドレスを返す。何らかのローカル・アドレスが指定されているときにはこのデフォルトは使われないことに注意されたい。ひとつ以上のアドレスがセットされているときには、そのうちのひとつのみが返されるが、それは必ずしも setDefaultLocalAddresses(List)で最初に指定されているアドレスとは限らない。 参照: IoAcceptor インターフェイスの getDefaultLocalAddress
void setDefaultLocalAddress(InetSocketAddress localAddress)	
boolean isReuseAddress()	参照: ServerSocket.getReuseAddress()
void setReuseAddress(boolean reuseAddress)	参照: ServerSocket.setReuseAddress(boolean)
int getBacklog()	バックログのサイズを返す。バックログとは処理待ち積堆量のことである。
void setBacklog(int backlog)	バックログのサイズをセットする。これはこのクラスがバインドされていないときのみなされる。
SocketSessionConfig getSessionConfig()	このアクセプタのサービスが生成した新しい SocketSession のデフォルトの設定を返す。 参照: IoService インターフェイスの getSessionConfig

## 6.2.6 NioSocketAcceptor

SocketAcceptor はインターフェイスであり、開発者が実際に TCP ソケット通信で使うのはこれを実装したこの NioSocketAcceptor である。これは AbstractPollingIoAcceptor という抽象クラスを継承したものである。

AbstractPollingIoAcceptor は、TCP/IP 網インターフェイスで、あるソケットたちがアクティブなループによってチェックされ(これをポーリングと呼んでいる)、あるソケットで処理が必要となったときに起動(ウェークアップ)される。このクラスはサーバ・ソケットのバインド、アクセプト、廃棄といったロジックを処理する。クライアントからの接続受け付け処理のために、ある Executor (スレッド)が使われ、また読み出し、書き込み、及びクローズといったクライアントの I/O 操作の処理のために、ある AbstractPollingIoProcessor が使われる。(このパラグラフは AbstractPollingIoAcceptor の Javadoc をそのまま訳したもの)

NioSocketAcceptor をインスタンス化して初期化される際は、以下のことが実際に行われている:

1. デフォルトのセッション設定(DefaultSocketSessionConfig)
2. デフォルトとして SimpleIoProcessorPool を設定
3. IoService の初期化

この `NioSocketAcceptor` クラス自身のメソッドは多くはないが、多くのインターフェイスを実装し、また幾つかの抽象クラス継承しているので、それらの極めて多くのローレベルのメソッドたちが利用できる。以下は javadoc で示されているコンストラクタとメソッドである:

表 6-7: `NioSocketAcceptor` クラスのコンストラクタとメソッド

コンストラクタ	
<code>NioSocketAcceptor()</code>	デフォルトのパラメタたち(多重スレッド・モデル)を使った <code>NioSocketAcceptor</code> のコンストラクタ
<code>NioSocketAcceptor(Executor executor, IoProcessor&lt;NioSession&gt; processor)</code>	接続のイベントたちを処理するための指定された <code>Executor</code> と、I/O イベント処理のための指定された <code>IoProcessor</code> をもった <code>NioSocketAcceptor</code> のコンストラクタで、同じタイプの複数の <code>IoService</code> 上で同じ <code>executor</code> とプロセッサを共有する為に有用なコンストラクタである パラメタ: <code>executor</code> - 接続のための <code>executor</code> <code>processor</code> - I/O 操作のためのプロセッサ
<code>NioSocketAcceptor(int processorCount)</code>	デフォルトのパラメタたちを使い、またマルチスレッドの I/O 操作の為に与えられた数の <code>NioProcessor</code> を使う <code>NioSocketAcceptor</code> のコンストラクタ パラメタ: <code>processorCount</code> - 生成し <code>SimpleIoProcessorPool</code> に置くプロセッサの数
<code>NioSocketAcceptor(IoProcessor&lt;NioSession&gt; processor)</code>	デフォルトの設定を使うが、指定された <code>IoProcessor</code> を使う <code>NioSocketAcceptor</code> のコンストラクタで、同じタイプの複数の <code>IoService</code> 上で同じプロセッサを共有するのに有用なコンストラクタである パラメタ: <code>processor</code> - I/O 操作のためのプロセッサ
メソッド	
<code>protected void init()</code> throws <code>Exception</code>	このポーリング・システムを初期化する。このメソッドはコンストラクト時に呼び出される。 規定: クラス <code>AbstractPollingIoAcceptor&lt;NioSession,ServerSocketChannel&gt;</code> の <code>init</code> 例外: <code>Exception</code> - 基になっているシステム呼び出しで何らかの例外がスローされた
<code>protected void destroy()</code> throws <code>Exception</code>	このポーリング・システムをデストロイする。これはこの <code>IoAcceptor</code> 実装物が廃棄される時に呼び出される。 規定: <code>AbstractPollingIoAcceptor&lt;NioSession,ServerSocketChannel&gt;</code> クラスの <code>destroy</code> 例外: <code>Exception</code> - 基になっているシステム呼び出しで何らかの例外がスローされた
<code>public TransportMetadata getTransportMetadata()</code>	このサービスが走っている <code>TransportMetadata</code> を返す 規定: インターフェイス <code>IoService</code> の <code>getTransportMetadata</code>
<code>public SocketSessionConfig getSessionConfig()</code>	このサービスによって作られた新しい <code>IoSession</code> のデフォルト設定を返す 規定: インターフェイス <code>IoService</code> の <code>getSessionConfig</code> 規定: インターフェイス <code>SocketAcceptor</code> の <code>getSessionConfig</code> オーバーライド: クラス <code>AbstractIoService</code> の <code>getSessionConfig</code>
<code>public InetSocketAddress getLocalAddress()</code>	現在バインドされているローカル・アドレスを返す。ひとつ以上のローカル・アドレスがバインドされているときには、そのうちひとつだけが返されるが、それは必ずしも最初にバインドされたアドレスとは限らない。

	<p>規定: インターフェイス <code>IoAcceptor</code> の <code>getLocalAddress</code></p> <p>規定: インターフェイス <code>SocketAcceptor</code> の <code>getLocalAddress</code></p> <p>オーバーライド: クラス <code>AbstractIoAcceptor</code> の <code>getLocalAddress</code></p>
<p><code>public InetAddress getDefaultLocalAddress()</code></p>	<p><code>IoAcceptor.bind()</code>メソッドで引数が指定されていないときにバインドされるデフォルトのローカル・アドレスを返す。このデフォルトはローカル・アドレスが指定されているときに使われないことに注意。ひとつ以上のアドレスがセットされているときには、そのうちひとつだけが返されるが、それは必ずしも <code>IoAcceptor.setDefaultLocalAddresses(List)</code>で最初に指定されたアドレスとは限らない。</p> <p>規定: インターフェイス <code>IoAcceptor</code> の <code>getDefaultLocalAddress</code></p> <p>規定: インターフェイス <code>SocketAcceptor</code> の <code>getDefaultLocalAddress</code></p> <p>オーバーライド: クラス <code>AbstractIoAcceptor</code> の <code>getDefaultLocalAddress</code></p>
<p><code>public void setDefaultLocalAddress(InetAddress localAddress)</code></p>	<p>規定: インターフェイス <code>SocketAcceptor</code> の <code>setDefaultLocalAddress</code></p>
<p><code>public boolean isReuseAddress()</code></p>	<p>規定: インターフェイス <code>SocketAcceptor</code> の <code>isReuseAddress</code></p> <p>参照: <code>ServerSocket.getReuseAddress()</code></p>
<p><code>public void setReuseAddress(boolean reuseAddress)</code></p>	<p>規定: インターフェイス <code>SocketAcceptor</code> の <code>setReuseAddress</code></p> <p>参照: <code>ServerSocket.setReuseAddress(boolean)</code></p>
<p><code>public int getBacklog()</code></p>	<p>バックログのサイズを返す</p> <p>規定: インターフェイス <code>SocketAcceptor</code> の <code>getBacklog</code></p>
<p><code>public void setBacklog(int backlog)</code></p>	<p>バックログのサイズをセット。これはこのクラスがバインドされていないときのみ可能</p> <p>規定: インターフェイス <code>SocketAcceptor</code> の <code>setBacklog</code></p>
<p>protected NioSession <code>accept(IoProcessor&lt;NioSession&gt; processor, ServerSocketChannel handle) throws Exception</code></p>	<p>規定: クラス <code>AbstractPollingIoAcceptor&lt;NioSession,ServerSocketChannel&gt;</code> の <code>accept</code></p> <p>パラメタ: <code>processor</code> – <code>IoSession</code> に結びつける為の <code>IoProcessor</code> <code>handle</code> – サーバ・ハンドル</p> <p>戻り値: 生成された <code>IoSession</code></p> <p>例外: <code>Exception</code> - 基になっているシステム呼び出しで何らかの例外がスローされた</p>
<p>protected ServerSocketChannel <code>open(SocketAddress localAddress) throws Exception</code></p>	<p>与えられたローカル・アドレスのためにあるサーバ・ソケットをオープンする</p> <p>規定: クラス <code>AbstractPollingIoAcceptor&lt;NioSession,ServerSocketChannel&gt;</code> の <code>open</code></p> <p>パラメタ: <code>localAddress</code> – 関連付けられるローカル・アドレス</p> <p>戻り値: オープンされたサーバ・ソケット</p>

	<p>例外: Exception - 基になっているシステム呼び出しで何らかの例外がスローされた</p>
<p>protected SocketAddress <b>localAddress</b>(ServerSocketChannel handle)  throws Exception</p>	<p>与えられたサーバ・ソケットに結び付けられているローカル・アドレスを返す 規定: クラス AbstractPollingIoAcceptor&lt;NioSession,ServerSocketChannel&gt;の localAddress パラメタ: <b>handle</b> - サーバ・ソケット 戻り値: このハンドルに結び付けられたローカルな SocketAddress 例外: Exception - 基になっているシステム呼び出しで何らかの例外がスローされた</p>
<p>protected int <b>select</b>()  throws Exception</p>	<p>対応するチャンネルたちの少なくともひとつが I/O 操作が可能になっているかどうかをチェックする。このメソッドはブロッキング選択操作を行う。このメソッドは少なくともひとつのチャンネルが選択された、このセクタの wakeup メソッドが呼び出された、あるいは現在のスレッドが割り込まれた、のどれかが最初に起きた後で戻る。 規定: クラス AbstractPollingIoAcceptor&lt;NioSession,ServerSocketChannel&gt;の select 戻り値: レディ操作のセットが更新されてるキーの数 例外: IOException – I/O エラーが発生したとき ClosedSelectorException – このセクタがクローズしている Exception – 基になっているシステム呼び出しで何らかの例外がスローされた</p>
<p>protected Iterator&lt;ServerSocketChannel&gt; <b>selectedHandles</b>()</p>	<p>最後の AbstractPollingIoAcceptor.select()呼び出し中に到来接続受け付け可能ということが分かったサーバ・ソケットのセットの為のイテレータ。 規定: クラス AbstractPollingIoAcceptor&lt;NioSession,ServerSocketChannel&gt;の selectHandles 戻り値: レディなサーバ・ハンドラたちのリスト</p>
<p>protected void <b>close</b>(ServerSocketChannel handle)  throws Exception</p>	<p>サーバ・ソケットのクローズ 規定: クラス AbstractPollingIoAcceptor&lt;NioSession,ServerSocketChannel&gt;の close 例外: Exception – 基になっているシステム呼び出しで何らかの例外がスローされた</p>
<p>protected void <b>wakeup</b>()</p>	<p>AbstractPollingIoAcceptor.select()メソッドを割り込みをかける。このポール・セットが修正の必要が出るときに有用。 規定: クラス AbstractPollingIoAcceptor&lt;NioSession,ServerSocketChannel&gt;の wakeup</p>

## 6.2.7 IoBuffer

低レベル API のひとつである org.apache.mina.core.buffer.IoBuffer (2.0 版以前は org.apache.mina.common.ByteBuffer だったが java.nio と名前が同じで紛らわしかった) は java.nio.ByteBuffer の代わりに使用されるが、これは java.nio.ByteBuffer をより機能を拡大させたものである。ただしこのような java.nio.ByteBuffer のラップに関しては、そうしなくても代替手段があるとして次のバージョンでは**大幅に変更(多分廃止化)される**可能性があるなので、注意されたい。

MINA では現在は `java.nio.ByteBuffer` を直接使うことはしていないが、それは次の理由による:

- `java.nio` では `fill`、`get/putString`、及び `get/putAsciiInt()` のような有用なゲッタとプッタが十分用意されていない。
- 容量が固定されているので可変長データを書き込むのが難しい。

次のコードはこのバッファの具体的な使い方である。

```
// Build a string to send.
CharsetEncoder = ...;
IoSession session = ...;
IoBuffer buffer = IoBuffer.allocate(16);
    buffer.setAutoExpand(true)
    .putString("It is ", encoder)
    .putString(new Date().toString(), encoder)
    .putString(" now.\r\n", encoder).flip();

// Asynchronous write requested data.
session.write(buffer);
```

バッファの生成には以下のような方法がある:

- 新しいヒープ・バッファ(ガベージ・コレクトされたメモリ領域上で作ったバッファ)の割当ては  
`IoBuffer buf = IoBuffer.allocate(1024, false);`
- 新しい直接バッファ(ガベージ・コレクトされたメモリ領域外で作ったバッファ)の割当ては  
`IoBuffer buf = IoBuffer.allocate(1024, true);`
- あるいはデフォルトのヒープ・バッファの割当ては  
`IoBuffer.setUseDirectBuffer(false);`  
そして新しいヒープ・バッファを戻す:  
`IoBuffer buf = IoBuffer.allocate(1024);`

で実現される。一般にはヒープ・バッファを使ったほうが効率が良い。またこのクラスには `java.io` のバッファとバイト・アレーをラップする為の幾つかの `wrap` メソッドが用意されている。

このバッファは豊富なプッタとゲッタが用意されていることに加えて、そのサイズが自動伸縮するように設定できるところが `java.nio` の `ByteBuffer` と違うところである。

`java.nio` の `ByteBuffer` を使って可変長のデータをかきこむのは、このバッファのサイズが固定であるので、あまり簡単なことではない。`IoBuffer` では `autoExpand` という属性がある。この属性が真のときは、`BufferOverflowException` あるいは `IndexOutOfBoundsException` を生じることがなくなる(インデックスがマイナスの場合を除いて)。この場合は自動的に容量とリミットが増加する。たとえば:

```
String greeting = messageBundle.getMessage( "hello" );
IoBuffer buf = IoBuffer.allocate( 16 );
// autoExpandをオンにする (デフォルトではオフになっている)
buf.setAutoExpand( true );
buf.putString( greeting, utf8encoder );
```

このバッファのもとになっている `ByteBuffer` が `IoBuffer` によりエンコードされたバイト・データが 16 バイトを超えた場合には内部で再アロケートされる。もし容量が 2 倍になれば、その文字列が書き込まれた最後の位置にまでリミットが殖える。

割り当てたメモリ領域のほとんどが使われないときは、そのバッファの容量を減らすことも可能である。その為に `IoBuffer` は `autoShrink` という属性を持っている。この `autoShrink` がオンになっていると、`IoBuffer` は `compact()` が呼ばれ、現在の容量の 1/4 以下しか使われていないときに、その容量を半分にする。また `shrink()` メソッドでそのバッファの容量をマニュアルで縮小させることも可能である。このオブジェクトのもとになっている `ByteBuffer` が `IoBuffer` により再割り当てされ、その容量に変更が起きたときは異なった `ByteBuffer` インスタンスを戻す。新しい容量がそのバッファの `minimumCapacity()` 以下の時には、`compact()` あるいは `shrink()` を呼んでもその容量は変化しない。

派生バッファは `duplicate()`、`slice()`、あるいは `asReadOnlyBuffer()` によって生成されたバッファのことであるが、これらは同じメッセージを複数の `IoSession` にブロードキャストする時には特に有用である。派生されたバッファとそのもとのバッファの双方とも自動伸張でも自動縮小でもないことに注意する必要がある。これらのバッファで `AutoExpand(boolean)` あるいは `setAutoShrink(boolean)` を呼ぶと `IllegalStateException` が生起される。

このクラスのコンストラクタとメソッド一覧は [Javadoc](#) を見て頂きたい。非常に多くのメソッドが用意されているが、かなりの部分 `java.nio` の `Buffer`、`IoBuffer`、及び `ByteBuffer` などと重複しているので、そちらのドキュメントも参照されたい。

## 6.3節 Apache MINA が使っているロギング・システム

Apache MINA ではアプリケーションの開発者たちが自分のロギング・システムを使い易いように、`Simple Logging Facade for Java (SLF4J)` というファサードを採用している。ファサードは、デザイン・パタンでなじみのもので、その名の通り窓口的なユーティリティで、このファサードにログを書き込むことでいろんなロギング・システムに対応できる。Java では `java.util.logging` が一般的なロギング・システムであるが、SLF4J では `log4j` というパッケージが用意されており、開発の途中で `java.util.logging` から `log4j` にソース・コードを全く変更しないで行ける点が魅力的である。`log4j` は `Struts` でも使われているので、なじみの人も多いだろう。

### 6.3.1 基本的な使い方

次のコード(温度管理)を見てみよう:

```
001 import org.slf4j.Logger;
002 import org.slf4j.LoggerFactory;
003
004 public class Wombat {
005
006     final Logger logger = LoggerFactory.getLogger(Wombat.class);
007     Integer t;
008     Integer oldT;
009     public void setTemperature(Integer temperature) {
010
011         oldT = t;
012         t = temperature;
013         logger.debug("Temperature set to {}. Old temperature was {}. ", t, oldT);
014         if(temperature.intValue() > 50) {
015             logger.info("Temperature has risen above 50 degrees.");
016         }
017     }
018 }
```



```
017  }  
018  }
```

6行目で `logger` を生成し、13行と15行目でそのローガーに情報を書き込んでいる。13行目の文字列の中の `{}` がデータの置き場所になっている。

ログ情報は以下のレベルで区分されている:

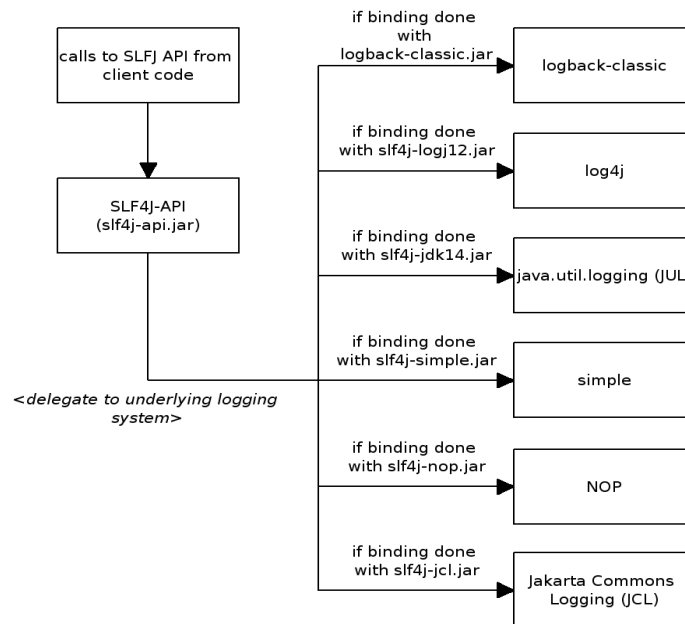
表 6-7: SLF4J のログ・レベル

ログ・レベル	内容
NONE	設定にかかわらず何らログ・イベントが生成されない
TRACE	そのロギング・システムのなかに TRACE イベントを生成
DEBUG	そのロギング・システムのなかにデバッグ・メッセージを生成
INFO	そのロギング・システムのなかに情報のメッセージを生成
WARN	そのロギング・システムのなかに警告メッセージを生成
ERROR	そのロギング・システムのなかにエラー・メッセージを生成

### 6.3.2 ロギング・システムのバインド

SLF4J は下図のように、ユーザが利用するロギング・システムに委譲する形式をとっている。その為に、各ロギング・システム用の `jar` ファイルが同梱されているので、それを選択してバインドすることになる。ロギング・システムを切り替えるときは、単に自分のクラス・パス上の SLF4J のバインド物を置き換えればよい。例えば `java.util.logging` から `log4j` に変更する為には、単に `slf4j-jdk14-1.6.1.jar` を `slf4j-log4j12-1.6.1.jar` で置き換えれば良い。

図 6-6: ロギング・システムのバインド



SLF4J バインド物としては以下のものが用意されている:

- slf4j-log4j12-1.6.1.jar  
広く利用されているロギング・フレームワークである log4j version 1.2 用のバインド物。この場合は自分のクラス・パス上に log4j.jar も置かねばならないことに注意
- slf4j-jdk14-1.6.1.jar  
JDK 1.4 ロギングとも呼ばれている java.util.logging 用のバインド物
- slf4j-nop-1.6.1.jar  
総てのロギングを無視する NOP 用のバインド物
- slf4j-simple-1.6.1.jar  
Simple 実装のためのバインド物で、総てのイベントを System.err に送り出す。INFO レベル以上の総てのメッセージが印刷される。このバインド物は小規模のアプリケーションには有用かもしれない。
- slf4j-jcl-1.6.1.jar
- akarta Commons Logging 向けのバインド物。このバインディングでは SLF4J ロギングの総てを JCL に委譲する

## 6.4節 エコーバック・サーバのプログラム例

先ず簡単なエコーバックのサーバで、MINA の基本的なコンセプトを Eclipse 上で把握することにする。

### 6.4.1 SLF4J のパッケージのダウンロード

1. SLF4J のダウンロードの [サイト](#) から適当な圧縮形式 (例えば ZIP の slf4j-1.6.1.zip) のファイルを適当なディレクトリにダウンロードする。

2. 次に展開したもののメインのフォルダ(例えば SLF4J-1.6.1)を C:SLF4J と C ディレクトリに移す。
3. Eclipse を起動して、「ウィンドウ」→「設定」→「java」→「ビルド・パス」→「ユーザー・ライブラリ」→「新規」で新しいライブラリ名を入力(例えば slf4j\_lib など)
4. 次にこのライブラリにダウンロードした jar ファイルたち、即ち c:slf4j の中の以下の jar を選択してインポートする:
  1. slf4j-api-1.6.1
  2. slf4j-simple-1.6.1(ここでは System.Err 出力を利用する)
5. 次にパッケージ・エクスプローラ上で自分のパッケージを右クリック→「ビルドパス」→「ライブラリの追加」→「ユーザ・ライブラリ」で自分が名付けた SLF4J のライブラリをチェックする

#### 6.4.2 MINA のパッケージのダウンロード

1. まず[ダウンロードのサイト](#)から適当な圧縮形式(例えば ZIP)のファイルを適当なディレクトリにダウンロードする。ここではまだ安定化していないが最新の「Apache MINA 2.0.0-RC1 unstable (Java 5+)」の zip アーカイブ・ファイルをダウンロードする。
2. 次に展開したもののメインのフォルダ(例えば MINA 2.0.0-RC1 )を C:MINA と C ディレクトリに移す。
3. Eclipse を起動して、「ウィンドウ」→「設定」→「java」→「ビルド・パス」→「ユーザー・ライブラリ」→「新規」で新しいライブラリ名を入力(例えば mina\_lib など)
4. 次にこのライブラリにダウンロードした jar ファイルたち、即ち **c:mina.dist** の中の総ての jar を選択してインポートする
5. 次にパッケージ・エクスプローラ上で自分のパッケージを右クリック→「ビルドパス」→「ライブラリの追加」→「ユーザ・ライブラリ」で自分が名付けた MINA のライブラリをチェックする

#### 6.4.3 簡単なエコー・サーバのプログラム

次のコードは Apache MINA のサンプル・プログラムのひとつに少し手を加えたものである。このプログラムは SimpleMinaServer というメインのクラスと, IoHandlerAdapter を継承してそのイベント処理メソッドをオーバーライドした EchoProtocolHandler というクラスで構成されている。最初にメインの SimpleMinaServer を見てみよう。

##### SimpleMinaServer

```
package TCPIP_Programming;

/*
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more contributor license agreements. See the NOTICE file
 * distributed with this work for additional information
 * regarding copyright ownership. The ASF licenses this file
 * to you under the Apache License, Version 2.0 (the
 * "License"); you may not use this file except in compliance
 * with the License. You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing,
 * software distributed under the License is distributed on an
 * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
 * KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations
 * under the License.
 */
```

```

*
*/

import java.net.InetSocketAddress;
import org.apache.mina.transport.socket.SocketAcceptor;
import org.apache.mina.transport.socket.nio.NioSocketAcceptor;

/**
 * (<b>Entry point</b>) Echo server
 *
 * @author The Apache MINA Project (dev@mina.apache.org)
 * @version $Rev: 600461 $, $Date: 2007-12-03 10:55:52 +0100 (Mon, 03 Dec 2007)
$
 */
public class SimpleMinaServer {
    /** Choose your favorite port number. */
    private static final int PORT = 1000;

    public static void main(String[] args) throws Exception {
        SocketAcceptor acceptor = new NioSocketAcceptor();

        // Bind
        acceptor.setReuseAddress(true);
        acceptor.setHandler(new EchoProtocolHandler());
        acceptor.bind(new InetSocketAddress(PORT));

        System.out.println("Listening on port " + PORT);

        for (;;) {
            System.out.println("R: " +
acceptor.getStatistics().getReadBytesThroughput() + ", W: " +
acceptor.getStatistics().getWrittenBytesThroughput());
            Thread.sleep(3000);
        }
    }
}

```

ここでは最初に TCP 受付ポート番号(ここでは 1000)を設定している。次に main()メソッドを見ると最初にアクセプタとして TCP 用の NioSocketAcceptor のインスタンスを生成し、これに対し次に示す EchoProtocolHandler というハンドラをセットし、更に受付ポート番号と IP アドレスをバインドしている。このバインド操作によってこのサーバが稼働開始する。メインのスレッドは本質的にこれだけで用がなくなる。あとは MINA のスレッドが TCP 接続受け付けを行い、接続(ここではセッションと称している)ごとにスレッドが生成されて、ユーザ毎にサービスを提供する。このプログラムではメインのスレッドは 3 秒ごとに読み書きのスループット統計値を出力している。不要ならばこの部分は削除すれば良い。

このプログラムでは、acceptor.setReuseAddress(true);が使われているが、これは「[setReuseAddress メソッドについて](#)」の節で説明したように、あまり意味を持たない。

それでは I/O ハンドラのほうはどうだろうか。以下のコードを見て頂きたい。

#### EchoProtocolHandler

```

package TCPIP_Programming;

/*
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more contributor license agreements. See the NOTICE file

```

```

* distributed with this work for additional information
* regarding copyright ownership. The ASF licenses this file
* to you under the Apache License, Version 2.0 (the
* "License"); you may not use this file except in compliance
* with the License. You may obtain a copy of the License at
*
* http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing,
* software distributed under the License is distributed on an
* "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
* KIND, either express or implied. See the License for the
* specific language governing permissions and limitations
* under the License.
*
*/

import org.apache.mina.core.buffer.IoBuffer;
import org.apache.mina.core.service.IoHandler;
import org.apache.mina.core.service.IoHandlerAdapter;
import org.apache.mina.core.session.IdleStatus;
import org.apache.mina.core.session.IoSession;
import org.apache.mina.filter.ssl.SslFilter;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
 * {@link IoHandler} implementation for echo server.
 *
 * @author <a href="http://mina.apache.org">Apache MINA Project</a>
 */
public class EchoProtocolHandler extends IoHandlerAdapter {
    private final static Logger LOGGER =
LoggerFactory.getLogger(EchoProtocolHandler.class);

    @Override
    public void sessionCreated(IoSession session) {
        session.getConfig().setIdleTime(IdleStatus.BOTH_IDLE, 10);

        // We're going to use SSL negotiation notification.
        session.setAttribute(SslFilter.USE_NOTIFICATION);
    }

    @Override
    public void sessionClosed(IoSession session) throws Exception {
        LOGGER.info("CLOSED");
    }

    @Override
    public void sessionOpened(IoSession session) throws Exception {

```

```

        LOGGER.info("OPENED");
    }

    @Override
    public void sessionIdle(io.Session session, IdleStatus status) {
        LOGGER.info("*** IDLE #" + session.getIdleCount(IdleStatus.BOTH_IDLE) + " ***");
    }

    @Override
    public void exceptionCaught(io.Session session, Throwable cause) {
        session.close(true);
    }

    @Override
    public void messageReceived(io.Session session, Object message)
        throws Exception {
        LOGGER.info("Received : " + message );
        // Write the received data back to remote peer
        session.write(((io.Buffer) message).duplicate());
    }
}

```

この場合では、受理した `message` オブジェクトをそのまま `io.Buffer` にキャストしてそのコピーをクライアントに返しているの、特に相手がどの文字エンコーディング使っているかを心配しなくて良い。インターネットの ECHO というアプリケーションはそうなっていて、送られてきたものをそのまま送り返すテスト用のものである。しかしながら、一般のアプリケーションでは、文字エンコーディング処理が必ず必要である。

#### 6.4.4 Eclipse での動作確認

Eclipse 上で `SimpleMinaServer` をたちあげ、`Tera Term` を 2 つ起動してこのサーバにアクセスしてみよう。下図のように正しくクライアントが識別され、エコー応答していることが確認される。サーバのスクリーンにはプログラムで読み書きのスループットだけでなく、ログ出力が表示されている。

ここで `sessionIdle(io.Session session, IdleStatus status)` というメソッドは、そのセッションがアイドルである状態を返している。アイドルであることの検出は、`session.getConfig().setIdleTime(IdleStatus.BOTH_IDLE, 10);` で、10 秒間を設定されている。アイドル状態には次の 3 つが存在する:

- `READER_IDLE` - リモート・ピアからデータが到来していない
- `WRITER_IDLE` - このセッションがデータを書いていない
- `BOTH_IDLE` - `READER_IDLE` と `WRITER_IDLE` の双方

図 6-7: 二つのクライアントからのアクセス

```

<終了> SimpleMinaServer [Java アプリケーション] C:\Program Files\Java\jre6\bin\javaw.exe (2010/08/11 8:50:15)
Listening on port 1000
R: 0.0, W: 0.0
7473 [NioProcessor-1] INFO TCPIP_Programming.EchoProtocolHandler - OPENED
R: 0.0, W: 0.0
17815 [NioProcessor-1] INFO TCPIP_Programming.EchoProtocolHandler - *** IDLE #1 ***
R: 0.0, W: 0.0
27757 [NioProcessor-1] INFO TCPIP_Programming.EchoProtocolHandler - *** IDLE #2 ***
27887 [NioProcessor-1] INFO TCPIP_Programming.EchoProtocolHandler - Received : HeapBuffer[pos=0 lim=23 cap=2048: 63 60 69 65 6E 74 31 3A 20 74 65 73 74 20 64 61...]
R: 0.0, W: 0.0
35637 [NioProcessor-2] INFO TCPIP_Programming.EchoProtocolHandler - OPENED
38009 [NioProcessor-1] INFO TCPIP_Programming.EchoProtocolHandler - *** IDLE #1 ***
R: 0.0, W: 0.0
45780 [NioProcessor-2] INFO TCPIP_Programming.EchoProtocolHandler - *** IDLE #1 ***
48152 [NioProcessor-1] INFO TCPIP_Programming.EchoProtocolHandler - *** IDLE #2 ***
50135 [NioProcessor-2] INFO TCPIP_Programming.EchoProtocolHandler - Received : HeapBuffer[pos=0 lim=23 cap=2048: 63 60 69 65 6E 74 20 32 3A 20 74 65 73 74 20 64...]
R: 0.0, W: 0.0
68295 [NioProcessor-1] INFO TCPIP_Programming.EchoProtocolHandler - *** IDLE #3 ***
60282 [NioProcessor-2] INFO TCPIP_Programming.EchoProtocolHandler - *** IDLE #1 ***
R: 0.0, W: 0.0
68017 [NioProcessor-2] INFO TCPIP_Programming.EchoProtocolHandler - Received : HeapBuffer[pos=0 lim=23 cap=2048: 63 60 69 65 6E 74 20 32 3A 20 74 65 73 74 20 64...]
68431 [NioProcessor-1] INFO TCPIP_Programming.EchoProtocolHandler - *** IDLE #4 ***
R: 0.0, W: 0.0
78150 [NioProcessor-2] INFO TCPIP_Programming.EchoProtocolHandler - *** IDLE #1 ***

```

## 6.5節 Apache MINA での文字エンコーディング

### 6.5.1 バイト・バッファでの変換

MINA の `IoHandlerAdapter` クラスの `messageReceived(IoSession session, Object message)` というメソッドは、受信メッセージである `message` というオブジェクトを渡してくれる。これはヒープ・バイト・バッファ(本質的に `IoBuffer`) である。従って「文字セット変換」の節の [NIO のクラスを使った変換](#) で示した方法が使える。

`EchoProtocolHandler.java` を次のように変更してみよう:

```

package TCPIP_Programming;

import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.charset.Charset;
import java.nio.charset.CharsetDecoder;
import org.apache.mina.core.buffer.IoBuffer;
import org.apache.mina.core.service.IoHandler;
import org.apache.mina.core.service.IoHandlerAdapter;
import org.apache.mina.core.session.IdleStatus;
import org.apache.mina.core.session.IoSession;
import org.apache.mina.filter.ssl.SslFilter;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
 * {@link IoHandler} implementation for echo server.
 *
 * @author <a href="http://mina.apache.org">Apache MINA Project</a>
 */
public class EchoProtocolHandler extends IoHandlerAdapter {

```

```

private final static Logger LOGGER = LoggerFactory.getLogger(EchoProtocolHandler.class);

@Override
public void sessionCreated(ioSession session) {
    session.getConfig().setIdleTime(IdleStatus.BOTH_IDLE, 10);

    // We're going to use SSL negotiation notification.
    session.setAttribute(SslFilter.USE_NOTIFICATION);
}

@Override
public void sessionClosed(ioSession session) throws Exception {
    LOGGER.info("CLOSED");
}

@Override
public void sessionOpened(ioSession session) throws Exception {
    LOGGER.info("OPENED");
}

@Override
public void sessionIdle(ioSession session, IdleStatus status) {
    LOGGER.info("*** IDLE #" + session.getIdleCount(IdleStatus.BOTH_IDLE) + " ***");
}

@Override
public void exceptionCaught(ioSession session, Throwable cause) {
    session.close(true);
}

@Override
public void messageReceived(ioSession session, Object message)
    throws Exception {
    LOGGER.info( "Received bytes : " + message );
    // 文字コードの選択
    Charset cs = Charset.forName("Windows-31J");
    IoBuffer mes = (IoBuffer)message;
    // バイトから文字へのデコーダの生成
    CharsetDecoder dec = cs.newDecoder();
    // データ読み出しの為に byte[] から char[] への実質的な変換
    CharBuffer cb = dec.decode( mes.buf() );
    LOGGER.info( "Received message : " + cb.toString() );
    // Write the received data back to remote peer
    session.write((mes.flip()).duplicate());
}
}

```

このコードの赤で示した部分が `IoBuffer` 及びそのもとになっている `ByteBuffer` を使って受信したデータを `Windows-31J` だとしてデコードしてログ出力している。

## 6.5.2 エンコーダ・フィルタによる方法



HTTPのように、クライアントと更新するデータが総てテキスト・データの場合は、フィルタで変換を受け持たせる方法がより便利であろう。ハンドラの場合は文字エンコーディングのことを一切気にしなくても良くなる。

フィルタに関しては次の節で説明するが、org.apache.mina.filter.codec.textline パッケージには、TextLineEncoder、TextLineDecoder、他のクラスが用意されている。これらはテキスト行単位でのエンコードとデコードをしてくれる。これらのクラスはフィルタ・チェーンに追加されることを前提にしている。

最初に次の MinaEchoServer と AnotherEchoProtocolHandler サーバのクラスを見て頂きたい。MinaEchoServer では TextLineCodec のフィルタが追加されている。それにより AnotherEchoProtocolHandler の messageReceived( IoSession session, Object message ) というメソッドから渡される message オブジェクトは String となっており、Session に書き込むメッセージも String で済んでしまう。従ってハンドラのコードはなじみの String ベースで極めてシンプルなものとなる。

```
package TCPIP_Programming;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.charset.Charset;

import org.apache.mina.filter.codec.ProtocolCodecFilter;
import org.apache.mina.filter.codec.textline.TextLineCodecFactory;
import org.apache.mina.filter.logging.LoggingFilter;
import org.apache.mina.transport.socket.SocketAcceptor;
import org.apache.mina.transport.socket.nio.NioSocketAcceptor;

public class MinaEchoServer {
    // 受付ポート番号
    private static final int PORT = 1000;

    public static void main(String[] args) throws IOException {
        // アクセプタを用意
        SocketAcceptor acceptor = new NioSocketAcceptor();
        acceptor.getFilterChain().addLast("logger", new LoggingFilter()); // デフォルトのロ
ガー追加
        acceptor.getFilterChain().addLast("codec",
            new ProtocolCodecFilter(
                new TextLineCodecFactory( Charset.forName("Windows-
31J")))); // 文字コード変換を追加
        acceptor.setReuseAddress( false ); // 必要があればここを true にする
        acceptor.setHandler(new AnotherEchoProtocolHandler());
        // ソケット・アドレスをバインドして、サーバを開始
        acceptor.bind(new InetSocketAddress(PORT));
        System.out.println("MINA Echo サーバ開始");
        // 1分ごとにスループットを報告
        for (;;) {
            System.out.println("R: " +
acceptor.getStatistics().getReadBytesThroughput() + ", W: " +
acceptor.getStatistics().getWrittenBytesThroughput());
            try {
                Thread.sleep(60 * 1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
package TCPIP_Programming;
```

```

import org.apache.mina.core.service.IoHandlerAdapter;
import org.apache.mina.core.session.IdleStatus;
import org.apache.mina.core.session.IoSession;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class AnotherEchoProtocolHandler extends IoHandlerAdapter {
    private final static Logger LOGGER =
    LoggerFactory.getLogger(EchoProtocolHandler.class);

    @Override
    public void sessionCreated(IoSession session) {
        session.getConfig().setIdleTime(IdleStatus.BOTH_IDLE, 10);
    }

    @Override
    public void sessionClosed(IoSession session) throws Exception {
        LOGGER.info("CLOSED");
    }

    @Override
    public void sessionOpened(IoSession session) throws Exception {
        LOGGER.info("OPENED");
    }

    @Override
    public void sessionIdle(IoSession session, IdleStatus status) {
        LOGGER.info("*** IDLE #" + session.getIdleCount(IdleStatus.BOTH_IDLE) + " ***");
    }

    @Override
    public void exceptionCaught(IoSession session, Throwable cause) {
        session.close(true);
    }

    @Override
    public void messageReceived(IoSession session, Object message)
        throws Exception {
        // 受信したオブジェクトはStringとして渡される
        String rmes = (String)message;
        LOGGER.info("Received : " + rmes);
        // 日本語のヘッダを付けて送り返す
        String smes = "MINA サーバ受信 : " + rmes;
        // クライアントがendを送信してきたら
        if( rmes.trim().startsWith("end") ) {
            session.write("endを受信しました");
            session.close(false);
            return;
        }
        session.write(smes);
    }
}

```

Eclipse で MinaEchoServer をたちあげ、SimpleClient.java をはしらせるか、Tera Term を起動するかしてこのサーバにアクセスするとその動作が確認できる。

SimpleClient.java でアクセスしたときのコンソールは次のようになる:

**SimpleClient.java:**

```

本日は晴天なり
MINA サーバ受信 : 本日は晴天なり ... Time: 16:21:49S
end
endを受信しました

```

**MinaEchoServer:**

```

MINA Echo サーバ開始
R: 0.0, W: 0.0
9098 [NioProcessor-1] INFO org.apache.mina.filter.logging.LoggingFilter - CREATED
9098 [NioProcessor-1] INFO org.apache.mina.filter.logging.LoggingFilter - OPENED
9098 [NioProcessor-1] INFO TCPIP_Programming.EchoProtocolHandler - OPENED
19098 [NioProcessor-1] INFO org.apache.mina.filter.logging.LoggingFilter - IDLE
19098 [NioProcessor-1] INFO TCPIP_Programming.EchoProtocolHandler - *** IDLE #1 ***
22710 [NioProcessor-1] INFO org.apache.mina.filter.logging.LoggingFilter - RECEIVED:
HeapBuffer[pos=0 lim=14 cap=2048: 96 7B 93 FA 82 CD 90 B0 93 56 82 C8 82 E8]
22728 [NioProcessor-1] INFO org.apache.mina.filter.logging.LoggingFilter - RECEIVED:
HeapBuffer[pos=0 lim=11 cap=2048: 20 2E 2E 2E 20 54 69 6D 65 3A 20]
22729 [NioProcessor-1] INFO org.apache.mina.filter.logging.LoggingFilter - RECEIVED:
HeapBuffer[pos=0 lim=11 cap=1024: 31 36 3A 32 31 3A 34 39 53 20 0A]
22729 [NioProcessor-1] INFO TCPIP_Programming.EchoProtocolHandler - Received : 本日は晴天なり ...
Time: 16:21:49S
22735 [NioProcessor-1] INFO org.apache.mina.filter.logging.LoggingFilter - SENT: HeapBuffer[pos=0
lim=53 cap=64: 4D 49 4E 41 83 54 81 5B 83 6F 8E F3 90 4D 20 3A...]
22736 [NioProcessor-1] INFO org.apache.mina.filter.logging.LoggingFilter - SENT: HeapBuffer[pos=0
lim=0 cap=0: empty]
27666 [NioProcessor-1] INFO org.apache.mina.filter.logging.LoggingFilter - RECEIVED:
HeapBuffer[pos=0 lim=14 cap=1024: 65 6E 64 20 2E 2E 2E 20 54 69 6D 65 3A 20]
27666 [NioProcessor-1] INFO org.apache.mina.filter.logging.LoggingFilter - RECEIVED:
HeapBuffer[pos=0 lim=11 cap=512: 31 36 3A 32 31 3A 35 34 53 20 0A]
27666 [NioProcessor-1] INFO TCPIP_Programming.EchoProtocolHandler - Received : end ... Time:
16:21:54S
27667 [NioProcessor-1] INFO org.apache.mina.filter.logging.LoggingFilter - SENT: HeapBuffer[pos=0
lim=18 cap=32: 65 6E 64 82 F0 8E F3 90 4D 82 B5 82 DC 82 B5 82...]
27667 [NioProcessor-1] INFO org.apache.mina.filter.logging.LoggingFilter - SENT: HeapBuffer[pos=0
lim=0 cap=0: empty]
27767 [NioProcessor-1] INFO org.apache.mina.filter.logging.LoggingFilter - CLOSED
27767 [NioProcessor-1] INFO TCPIP_Programming.EchoProtocolHandler - CLOSED

```

## 6.6節 Apache MINA のフィルタ

Apache MINA は、その[コンセプト図](#)で示したように、幾つかのフィルタで構成されるフィルタ・チェーンをハンドラとの間に介入できるようになっているいわゆる「インターセプタ・モデル」である。前節の `MinaEchoServer` のコードを見れば、そのコンセプトを理解することができる。

MINA ではフィルタをダイナミックな追加と削除(ホット・スワップ)が可能である。また `IoConnector`/`IoAcceptor` ごとにフィルタ・チェーンを構成できる。

各フィルタが持っている有用なメソッドを直接使いたいときは、`IoFilterChain` の `get` メソッドでそのフィルタを選択する(例えば `acceptor.getFilterChain().get("MyFilter").myFilterMethod(...)`) ことで、そのメソッドを呼ぶ。

MINA の `IoFilter` は、サーブレットの `ServletFilter` と似ている。`IoFilter` はアプリケーションであるハンドラとコンテナである `IoService` との間にチェーンとして存在し、アプリケーションに適した前後処理をしてくれる。`MinaEchoServer` ではロギングとテキスト入出力のためのフィルタが使われていた。`LoggingFilter` は総てのイベントとクライアント要求のログをとる。`ProtocolCodecFilter` は到来または送信される `ByteBuffer` をアプリケーションに適したオブジェクトの変換する。それ以外にも通信速度改善のための圧縮(`CompressionFilter`)、通信データの保護のための SSL 暗号(`SSLFilter`)、ブラックリストの載っているクライアントとの接続を阻止するブラックリスト・フィルタ(`BlackListFilter`)、接続確認のためのキープアライブ(`KeepAliveFilter`)などが用意されて

いるし、開発者が自分でフィルタを作成することも可能である。用意されているフィルタは <https://cwiki.apache.org/MINA/iofilter.html> で見ることが出来る。以下これらのいくつかを簡単に紹介する。

### 6.6.1 圧縮のためのフィルタ(CompressionFilter)

このフィルタは JZlib を使って総てのデータを圧縮/解凍する `IoFilter` である。JZlib は Deflate アルゴリズムにもとづく Zlib 圧縮 (zlib/gzip) を純粋 Java で実装したもので、BSD ライセンスのもとで配布されていて、`java.util.zip` の後継物である。LZW (DLCZ) アルゴリズム対応も予定されている。このフィルタはストリーム・レベルの圧縮対応の PARTIAL FLUSH 手法のみに対応している。

圧縮と解凍は上りと下りで選択的に実施できる。またアプリケーションが稼動中であってもオン・オフができる。

圧縮と解凍に使われる zlib 辞書の積上げのために、このフィルタでは zlib オブジェクトはこのフィルタが存続中は維持される。繰り返しデータが送信されるにつれ圧縮効率が向上する。

[Javadoc](#) で分かるように、このフィルタのコンストラクタは圧縮と解凍で圧縮レベルを指定することができる。通常は何も指定しなければデフォルトのレベルでインスタンスが生成される。

このクラスには以下のようなメソッドが含まれている:

- `doFilterWrite(IoFilter.NextFilter nextFilter, IoSession session, WriteRequest writeRequest)` : これはハンドラでセッションに送信データが書き込まれたときに呼び出される
- `isCompressInbound()` : 受信で解凍が指定されているか
- `isCompressOutbound()` : 発信で圧縮が指定されているか
- `messageReceived(IoFilter.NextFilter nextFilter, IoSession session, Object message)` : `IoHandler.messageReceived(IoSession, Object)` イベントのフィルタでここで解凍する
- `setCompressInbound(boolean compressInbound)` : 受信解凍を指定
- `setCompressOutbound(boolean compressOutbound)` : 発信で圧縮を指定

### 6.6.2 ブラック・リストのフィルタ(BlacklistFilter)

これはブラックリストされた IP アドレスまたはサブネットとの接続を阻止するフィルタである。その為に `InetAddress` または `Subnet` の配列あるいは反復子からなるブラックリストを、`setBlackList` または `setSubnetBlackList` メソッドで最初にセットする必要がある。ブラックリストは一括のセットと削除になっている。個別にオンまたはオフにするには、

- `block(InetAddress address)`、または
- `block(Subnet subnet)`

を呼び、オフにするには、

- `unblock(InetAddress address)`、または
- `unblock(Subnet subnet)`

を呼ぶ。

このフィルタによって影響を受けるハンドラのメソッドは、`messageReceived`、`messageSent`、`sessionClosed`、`sessionCreated`、`sessionIdle`、及び `sessionOpened` である。

### 6.6.3 送信データをバッファするフィルタ(BufferedWriteFilter)

送信データ(WriteRequest)をバッファする為のフィルタである。バッファすることで送信データグラムを大きくして、転送速度を実質改善することが出来る(IPレベルでの送信回数が減ることでネットワーク遅延の依存性が下がる)。これは非常に小さなメッセージを頻繁に発生していて、不必要にトラフィックの輻輳を起こすようなセッションには有効である。ただしバッファすることで影響が出るようなアプリケーション(エコーやチャット、即時性が必要なテレメトリなど)には使ってはいけない。

**このフィルタは ProtocolCodecFilter の前に配置**しなければならないことに注意のこと。

ハンドラの IoSession.write(Object)呼び出しでこのフィルタの filterWrite メソッドが呼び出され、バッファへの書き込みが行われる。バッファの容量を超えたらネットワークへの吐き出しが行われるので通常はハンドラ側ではこのフィルタの存在を気にしなくてもよい。強制的なバッファの内容のネットワークへの吐き出しは flush(IoSession session)メソッドで行う。

### 6.6.4 接続帯域制限フィルタ(ConnectionThrottleFilter)

あるピアとの接続間隔が指定した時間(デフォルトでは1秒間)よりも早く起きるのを制限する。あるクライアント間で指定した時間間隔(ミリ秒で指定する)よりも早く発生した TCP 接続はクローズされる。これは一部のヘビーなクライアントにより負荷が増加するのを制限するのに効果がある。**クライアント側では接続が一旦確立してもすぐにサーバが解放したことを知って、ある時間をおいて接続を再試行するようなプログラムにする。**

### 6.6.5 スレッド処理のフィルタ(ExecutorFilter)

このフィルタでは I/O のイベントを Executor に渡して、あるスレッド・モデルを実行し、セッションあたりのイベントたちを並行処理できるようになる。このフィルタを IoFilterChain に挿入することでいろんなスレッド・モデルに適用できる。デフォルトでは単一スレッド・モデルとなっている。挿入法は、例えば次のようになる:

```
acceptor.getFilterChain().addLast("executor", new ExecutorFilter(new  
OrderedThreadPoolExecutor(16)));
```

このフィルタは、圧縮やコーデックなどのフィルタの後に追加することが推奨されている。

#### 6.6.5.1 ライフ・サイクル管理

このフィルタはこの Executor のライフ・サイクルを管理しないことに注意を要する。自分がインスタンス化したある Executor を受けつける ExecutorFilter(Executor)または類似のコンストラクタでこのフィルタを作った場合は、プログラム開発者がそのライフ・サイクルを管理する(例えば ExecutorService.shutdown()を呼ぶ)コントロールと責任を持つことになる。ExecutorFilter(int)のような簡易コンストラクタ(Convenience constructor)でこのフィルタを生成したときは、明示的に destroy()メソッドを呼んでこの Executor をシャット・ダウンできる。

#### 6.6.5.2 イベントの順序付け(Event Ordering)

このフィルタの総ての簡易コンストラクタが `org.apache.mina.filter.executor.OrderedThreadPoolExecutor` の新しいインスタンスを生成している。従って、イベントの順序は以下のようにになっている:

- イベント・ハンドラの総てのメソッドは排他的に呼び出される(例えば `messageReceived` と `messageSent` は同時には呼び出されない)。
- イベントの順序は乱れない(例えば `messageReceived` は常に `sessionClosed` あるいは `messageSent` の前に呼び出される)。

しかしながら、コンストラクタの中で他の `Executor` のインスタンスを指定した場合は、イベントの順序は全く保たれなくなる。このことは、複数のイベント・ハンドラのメソッドが異なった順序で呼び出され得ることを意味する。例えば、`messageReceived`、`messageSent`、及び `sessionClosed` のイベントが生起したとすると、

- 総てのイベント・ハンドラ・メソッドたちが同時に呼び出され得る例えば `messageReceived` と `messageSent` は同時には呼び出され得る)。
- イベントの順序は乱れる(例えば `messageReceived` が呼ばれる前に `sessionClosed` あるいは `messageSent` の前に呼び出され得る)。

もしセッションあたりでイベントに順序付けをする必要がある場合は、`OrderedThreadPoolExecutor` のインスタンスを指定する。そうでなければ複数の簡易コンストラクタを使用する。

### 6.6.5.3 選択的フィルタリング(Selective Filtering)

デフォルトでは、`sessionCreated`、`filterWrite`、`filterClose`、及び `filterSetTrafficMask` を除く総てのイベントのタイプが `executor` に渡されるが、これは最も一般的な設定である。あるイベントのタイプたちのセットに限りて渡したいときは、それらのイベントのセットをコンストラクタのなかで指定することが出来る。例えば次のように、性能を最大化する為に書き込み操作の為にスレッドのプールを設定することが出来る。

```
IoService service = ...;
DefaultIoFilterChainBuilder chain = service.getFilterChain();
chain.addLast("codec", new ProtocolCodecFilter(...));
// 殆どのイベントにひとつのスレッド・プールを使用する
chain.addLast("executor1", new ExecutorFilter());
// 別のスレッド・プールを'filterWrite'のイベントのみに割り当てる
chain.addLast("executor2", new ExecutorFilter(IoEventType.WRITE));
```

### 6.6.5.4 OutOfMemoryError エラーの防止

簡易コンストラクタのパラメタとして規定されている [IoEventQueueThrottle](#) を参照のこと。

### 6.6.6 キープ・アライブのフィルタ(KeepAliveFilter)

キープアライブによるピア間の接続確認は、ネットワークにおける障害検出と対処に重要であることは[以前述べた](#)。 `IoEventType.SESSION_IDLE` のイベントが起きるとキープ・アライブの要求をピアに送信し、また送信されてきたキープ・アライブ要求にたいし応答を返信する。応答がしてされた時間内に戻ってこなかった場合は `KeepAliveRequestTimeoutHandler` というハンドラが呼び出されるので、これを適正に設定しておく必要がある。

#### 6.6.6.1 IoSessionConfig.setIdleTime(IdleStatus, int)とのインターフェイス

このフィルタは自分が関心を持っている `IdleStatus` たちの `idleTime` を自動的に調整している(例えば `IdleStatus.READER_IDLE` 及び `IdleStatus.WRITER_IDLE`)。 `IdleStatus` たちの `idleTime` を変更すると、このフィルタは予期せぬ振る舞いをする可能性がある。また、 `KeepAliveFilter` のあとにある何らかの `IoFilter` と `IoHandler` は何の `IoEventType.SESSION_IDLE` イベントも受けなくなることに注意。内部の `IoEventType.SESSION_IDLE` イベントを受けたいときは、 `setForwardEvent(boolean)` を引数を `true` にして呼ぶ。

### 6.6.6.2 KeepAliveMessageFactory の実装

このフィルタを使うには、受信したまたは送信したメッセージがキープ・アライブかどうかを判断し、また新しいキープ・アライブのメッセージを作るかどうかを判断する `KeepAliveMessageFactory` というインターフェイスを実装しなければならない。

表 6-8: キープ・アライブのモードとファクトリの実装

モードの名前	記述	実装
能動(Active)	そのリーダがアイドルになったときにキープ・アライブ要求を送信する。その要求が送られたらその要求に対する応答が <code>keepAliveRequestTimeout</code> 秒内に戻ってくるはずである。そうでないときは、指定した <code>KeepAliveRequestTimeoutHandler</code> が呼び出される。キープ・アライブ要求を受信したときはそれに対する応答を送り返す。	<code>KeepAliveMessageFactory.getRequest(IoSession)</code> と <code>KeepAliveMessageFactory.getResponse(IoSession, Object)</code> の双方が非ヌルで戻らねばならない。
順能動(Semi-active)	そのリーダがアイドルになったときにキープ・アライブ要求を送信する。しかし応答が受かったかどうかを気にしない。キープ・アライブ要求を受信したときはそれに対する応答を送り返す。	<code>KeepAliveMessageFactory.getRequest(IoSession)</code> と <code>KeepAliveMessageFactory.getResponse(IoSession, Object)</code> の双方が非ヌルで戻らねばならず、またその <code>timeoutHandler</code> が適正 <code>KeepAliveRequestTimeoutHandler.NOOP</code> 、 <code>KeepAliveRequestTimeoutHandler.LOG</code> あるいはカスタムの <code>KeepAliveRequestTimeoutHandler</code> の実装にセットされていて、セッション状態に影響を与えず、また例外をスローしないように、なっていなければならない。
受動(Passive)	自分ではキープ・アライブ要求を送信しないが、キープ・アライブ要求を受信したときはそれに対する応答を送り返す。	<code>KeepAliveMessageFactory.getRequest(IoSession)</code> が <code>null</code> を返さねばならず、また <code>KeepAliveMessageFactory.getResponse(IoSession, Object)</code> は非 <code>null</code> を返さねばならない。
聞くことをしない話して(Deaf Speaker)	そのリーダがアイドル状態になったときにキープ・アライブ要求を送信するが、	<code>KeepAliveMessageFactory.getRequest(IoSession)</code> が非 <code>null</code> を返さねばならず、また <code>KeepAliveMessageFactory.getResponse(Io</code>

	キープ・アライブ要求を受信したときはそれに対する応答を送り返さない。	Session, Object)が null を返さねばならず、また timeoutHandler は KeepAliveRequestTimeoutHandler.DEAF_SPEAKER にセットされていなければならない。
話すことをしない聞き手(Silent Listener)	自分ではキープ・アライブ要求を送信しないし、キープ・アライブ要求を受信したときはそれに対する応答を送り返さない。	KeepAliveMessageFactory.getRequest( IoSession)と KeepAliveMessageFactory.getResponse( IoSession, Object)の双方が null を返さねばならない。

### 6.6.6.3 タイムアウト処理

KeepAliveFilter は送信したキープ・アライブのメッセージに対する応答メッセージを受けなかったときには、それを KeepAliveRequestTimeoutHandler に通知をする。デフォルトのハンドラは KeepAliveRequestTimeoutHandler.CLOSE (セッションをクローズする) であるが、KeepAliveRequestTimeoutHandler.NOOP (何もしない)、KeepAliveRequestTimeoutHandler.LOG (ログ出力)、あるいは KeepAliveRequestTimeoutHandler.EXCEPTION (例外のスロー) のような他の設定にプリセットできる。また自分のハンドラを用意することも可能である。

特別のハンドラとして、KeepAliveRequestTimeoutHandler.DEAF\_SPEAKER があるが、これは上記の'deaf speaker'モード専用のものである。timeoutHandler の属性を KeepAliveRequestTimeoutHandler.DEAF\_SPEAKER にセットすることで、このフィルタが応答メッセージを待って、それによって応答のタイムアウト検出が出来なくなるのが防止される。

### 6.6.6.4 このフィルタの追加法

例えば次のようにフィルタ・チェーンに追加する:

```
KeepAliveFilter keepAlives = new KeepAliveFilter(new KeepAliveMessageFactory());
acceptor.getFilterChain().addLast("keepAlives", keepAlives);
```

### 6.6.7 ロギング・フィルタ(LoggingFilter)

これは既に「[Apache MINA が使っているロギング・システム](#)」の節で説明してある。

org.apache.mina.filter.logging.LoggingFilter は、総ての MINA プロトコル・イベントをログする。ユーザは各イベント毎にゲットまたはセットのメソッドを使い、ログ・レベルを設定できる。またイベント・タイプ毎に存在するメソッドをオーバーライドして、ログ・レベル、ログをする/しない、ログ・メッセージなどを指定できる。デフォルトでは、IoFilterAdapter.exceptionCaught(IoFilter.NextFilter, IoSession, Throwable)が LogLevel.WARN ことを除いて、総てのイベントが LogLevel.INFO のレベルでログされる。

### 6.6.8 処理時間計測のフィルタ(ProfilerTimerFilter)



このフィルタはあるイベントの発生時刻、`IoFilterAdapter` クラスのなかのあるメソッドの処理時間、及び呼び出し回数を計測する。つまり、そのメソッドの開始時の現在時刻を取得し、次に `nextFilter` のメソッドを呼び出し、再度現在時刻を取得し、両者の差を処理時間としている。一般的な使い方は、次のようである:

```
ProfilerTimerFilter profiler = new ProfilerTimerFilter( TimeUnit.MILLISECOND,
IoEventType.MESSAGE_RECEIVED);
chain.addFirst("Profiler", profiler);
```

計測できる `IoEventType` は、次のようである:

- `IoEventType.MESSAGE_RECEIVED`
- `IoEventType.MESSAGE_SENT`
- `IoEventType.SESSION_CREATED`
- `IoEventType.SESSION_OPENED`
- `IoEventType.SESSION_IDLE`
- `IoEventType.SESSION_CLOSED`

#### 6.6.8.1 テスト・プログラム

`IoEventType.MESSAGE_RECEIVED` イベント処理にかかる時間を計測してログ出力する簡単なフィルタ (`MesRecEventProfilerTestFilter`) を以下に示す:

```
package TCPIP_Programming;
import java.util.concurrent.TimeUnit;

import org.apache.mina.core.session.IoEventType;
import org.apache.mina.core.session.IoSession;
import org.apache.mina.filter.statistic.ProfilerTimerFilter;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class MesRecEventProfilerTestFilter extends ProfilerTimerFilter{
    // コンストラクタ
    MesRecEventProfilerTestFilter(){
        super();
    }
    // 時間単位とイベント指定可能なコンストラクタ
    MesRecEventProfilerTestFilter(TimeUnit tu, IoEventType et){
        super(tu, et);
    }
    private final static Logger LOGGER =
LoggerFactory.getLogger(MesRecEventProfilerTestFilter.class);
    // messageReceivedメソッドをオーバーライドする
    public void messageReceived(NextFilter nextFilter, IoSession session, Object
message)
        throws Exception {
        long start = System.currentTimeMillis();
        // スーパー・クラスのこのメソッドを呼ぶ
        super.messageReceived(nextFilter, session, message);
        long end = System.currentTimeMillis();
        // 処理時間をミリ秒でログする
        LOGGER.info("Received message processed in (mS) : " +
            (end - start));
        // 次に平均処理時間をミリ秒でログする
        LOGGER.info("Averaged process time in (mS) : " +
            getAverageTime(IoEventType.MESSAGE_RECEIVED));
    }
}
```

```
}
```

オーバーライドされた `messageReceived` メソッドはスーパー・クラスの同じメソッドを呼ぶ前後の時刻を計測し、そのあとでこれにかかった処理時間、及び `ProfilerTimerFilter` で統計している平均処理時間をログに出力している。

以下はこのフィルタのテスト用のサーバ(`MesRecEventProfilerTestServer`)のコードである:

```
package TCPIP_Programming;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.charset.Charset;
import java.util.concurrent.TimeUnit;

import org.apache.mina.core.service.IoHandlerAdapter;
import org.apache.mina.core.session.IoEventType;
import org.apache.mina.core.session.IoSession;
import org.apache.mina.filter.codec.ProtocolCodecFilter;
import org.apache.mina.filter.codec.textline.TextLineCodecFactory;
import org.apache.mina.filter.logging.LoggingFilter;
import org.apache.mina.transport.socket.SocketAcceptor;
import org.apache.mina.transport.socket.nio.NioSocketAcceptor;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class MesRecEventProfilerTestServer extends IoHandlerAdapter {
    // ロガー指定
    private final static Logger LOGGER =
LoggerFactory.getLogger(MesRecEventProfilerTestServer.class);
    // 受付ポート番号
    private static final int PORT = 1000;

    public static void main(String[] args) throws IOException {
        // アクセプタを用意
        SocketAcceptor acceptor = new NioSocketAcceptor();
        // デフォルトのロガー追加するときは次の行のコメント・アウトを外す
// acceptor.getFilterChain().addLast("logger", new LoggingFilter());
        // テキスト行のコーデックを追加
        acceptor.getFilterChain().addLast("codec",
            new ProtocolCodecFilter(
                new
TextLineCodecFactory( Charset.forName("Windows-31J"))));
        // 文字コード変換を追加

        // プロファイラ・フィルタを追加
        acceptor.getFilterChain().addLast("profiler", new
MesRecEventProfilerTestFilter());
// acceptor.getFilterChain().addLast("profiler", new
MesRecEventProfilerTestFilter
// (TimeUnit.MICROSECONDS, IoEventType.MESSAGE_RECEIVED));
        acceptor.setReuseAddress( false ); // 必要があればここを true にする
// セットするハンドラはこのクラスのオブジェクト
        acceptor.setHandler(new MesRecEventProfilerTestServer());
        // ソケット・アドレスをバインドして、サーバを開始
        acceptor.bind(new InetSocketAddress(PORT));
        System.out.println("MINA Echo サーバ開始");
    }
}
```

```

public void messageReceived(IOException session, Object message)
throws Exception {
    // 受信したオブジェクトはStringとして渡される
    String rmes = (String)message;
    LOGGER.info( "Received : " + rmes);
    // 日本語のヘッダを付けて送り返す
    String smes = "MINA サーバ受信 : " + rmes;
    // クライアントがendを送信してきたら
    if( rmes.trim().startsWith("end") ) {
        session.write("endを受信しました");
        session.close(false);
        return;
    }
    session.write(smes);
}
}

```

### 6.6.8.2 テスト例

このサーバを立ち上げ、Tera Term や SimpleClient から「テスト1」から「テスト4」、そして「end」という行を送信すると、以下のようなログ出力となる。この場合は、初回はインスタンス化等のために16mSかかっているが、その後は1mS以内で処理されていることが分かる。

```

MINA Echo サーバ開始
27388 [NioProcessor-1] INFO TCPIP_Programming.EchoProtocolHandler - Received : テスト1
27404 [NioProcessor-1] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message
processed in (mS) : 16
27412 [NioProcessor-1] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Averaged process
time in (mS) : 16.0
41577 [NioProcessor-1] INFO TCPIP_Programming.EchoProtocolHandler - Received : テスト2
41577 [NioProcessor-1] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message
processed in (mS) : 0
41577 [NioProcessor-1] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Averaged process
time in (mS) : 8.0
58232 [NioProcessor-1] INFO TCPIP_Programming.EchoProtocolHandler - Received : テスト3
58232 [NioProcessor-1] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message
processed in (mS) : 0
58232 [NioProcessor-1] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Averaged process
time in (mS) : 5.0
74151 [NioProcessor-1] INFO TCPIP_Programming.EchoProtocolHandler - Received : テスト4
74151 [NioProcessor-1] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message
processed in (mS) : 0
74151 [NioProcessor-1] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Averaged process
time in (mS) : 4.0
107477 [NioProcessor-1] INFO TCPIP_Programming.EchoProtocolHandler - Received : end
107477 [NioProcessor-1] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message
processed in (mS) : 0
107477 [NioProcessor-1] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Averaged process
time in (mS) : 3.0

```

### 6.6.9 メッセージのエンコードとデコードのフィルタ(ProtocolCodecFilter)

これは既に「[Apache MINA での文字エンコーディング](#)」の節で紹介してある。MINA のコーデック関連パッケージは以下のようなものである:

- org.apache.mina.filter.codec : 基本となるパッケージで ProtocolCodecFilter クラスを含む

- `org.apache.mina.filter.codec.demux` :ひとつのコーデックを幾つかのサブ・コーデックに分割して、より複雑なプロトコルに対応できるようにする
- `org.apache.mina.filter.codec.prefixedstring` :Java の `String` をもとにしたプレフィックスがついたメッセージの通信用
- `org.apache.mina.filter.codec.serialization` :Java のオブジェクト直列化を使ったプロトコル・コーデック
- `org.apache.mina.filter.codec.statemachine` :バイト文字列処理状態を使った処理用
- `org.apache.mina.filter.codec.textline` :テキスト行ベースのプロトコル用

多分最も良く利用されるのは `TextLineCodecFactory` であろう。

### 6.6.10 プロキシのためのフィルタ(`ProxyFilter`)

MINA にはプロキシのためのパッケージが多く用意されている。プロキシとはクライアントと別のサーバ間の間に介在する代理サーバであり、おもにセキュリティや負荷配分などに利用されている。Javadoc によれば、`ProxyFilter.java` というフィルタは `ProxyConnector` によって自動的にフィルタ・チェーンに挿入される。初期のハンドシェークのメッセージをこのプロキシに送信し、このプロキシからの何らかの応答を処理する。一旦このハンドシェークが確立され、プロキシされた接続が確立されたら、このフィルタはこの接続を介して流れるデータに対しては透過となる。

ドキュメンテーションが不十分なので、詳細は省略する。簡単なトンネル・プロキシの[サンプル](#)が使い方の参考になる。これは通過するメッセージの総てをログするだけのものである。この[パッケージ](#)をダウンロードしてインポートし、`org.apache.mina.example.proxy.Main` のクラスを引数 `12345 www.google.com 80` で実行し、ブラウザから `http://localhost:12345` でこのプロキシをアクセスする。

### 6.6.11 SSL フィルタ(`SslFilter`)

Secure Sockets Layer (セキュア・ソケット・レイヤ、SSL) は、セキュリティーを要求される通信のためのプロトコルで、TCP 層プロトコルの上位に位置し、通常は TCP をラッピングする形で利用される。UDP では使えないことに注意を要する。HTTP (HTTPS) なのではユーザ情報や電子商取引データのセキュリティ向上に頻繁に使われている。

この SSL フィルタが追加されると、即座に SSL 'hello' メッセージを送信することでハンドシェークの手続きが開始される。従って下記の `StartTLS` を実装していない限りマニュアルで `startSsl( IoSession )` を呼び出す必要はない。即座のハンドシェーク開始を望まない場合は、コンストラクタのなかで `autoStart` パラメタを `false` にする。

また接続開始前に `SSLFilter.setUseClientMode(true または false)` を使って接続時のモードを指定する必要がある。またサーバでは `IoSession.readerIdleTime` を設定し、`IoHandler.sessionIdle()` の際にそのセッションをクローズすることで、SSL ハンドシェークを開始しないユーザを切り離す必要がある。

このフィルタが Java 5 で導入されている `SSLEngine` を使用しているので、JDK 1.5 以上のバージョンが必須である。

`StartTLS` の実装:

StartTLS を実装するには、`DISABLE_ENCRYPTION_ONCE` 属性を使用する。

```
public void messageReceived( IoSession session, Object message ) {
    if ( message instanceof MyStartTLSRequest ) {
        // Insert SSLFilter to get ready for handshaking
        session.getFilterChain().addFirst( sslFilter );

        // Disable encryption temporarily.
        // This attribute will be removed by SSLFilter
        // inside the Session.write() call below.
        session.setAttribute( SSLFilter.DISABLE_ENCRYPTION_ONCE, Boolean.TRUE );

        // Write StartTLSResponse which won't be encrypted.
        session.write( new MyStartTLSResponse( OK ) );

        // Now DISABLE_ENCRYPTION_ONCE attribute is cleared.
        assert session.getAttribute( SSLFilter.DISABLE_ENCRYPTION_ONCE ) == null;
    }
}
```

MINA の [エコー・サーバのサンプル](#) は SSL が使われているので、参考になる。Main というクラスで、`USE_SSL = true;` として実験してみると良い。

## 6.7節 Apache MINA のスレッド

Java.net のプログラミング・モデルでは、`socket.read()` メソッドでそのスレッドが待たされ、同時に接続しているクライアント数分のスレッドが必要であった。ある時間をとってみると、実際にうごいているスレッドの数は接続数よりは遥かに少なく、殆どのスレッドが受信待ちの状態ではまっている、即ちブロッキングの状態に留まっている。スレッドの数が多ということはシステムのオーバヘッドも、その為のメモリ領域もが大きくなってしまふ。Java NIO の特徴は「[非ブロッキング・ソケット I/O](#)」の節で示したように、非ブロッキングのネットワーク・プログラミングのモデルであった。しかし NIO でのチャンネルとセレクトタの使用の場合は、単一のスレッドがイタレータを介してソケット接続数分の繰り返しを行っていた。

MINA では NIO ベースではあるが、これをより使いやすくしている。ローレベルの I/O ソケット処理は隠されており、イベント・ドリブンのインターフェイスを使って、非常に簡単に、且つ問題が起きにくい高い品質のアプリケーションが書けるようになっている。

MINA の特徴として、スレッド・モデルの高度なカスタマイズがある。即ち、単一スレッド、ひとつのスレッド・プール、そして複数のスレッド・プール (即ち SEDA : Staged Event-Driven Architecture モデル) によるスレッド処理が可能になっている。

MINA でのスレッドは、接続と送受信のための IO スレッド、及び IO ハンドラ・アダプタの為の IO プロセッサ・スレッドのためのスレッドたちが存在する。更に既に説明した [スレッド処理のフィルタ](#) では、ある I/O イベントの処理が時間がかかってしまうような場合にも有効な `ExecutorFilter` のスレッド環境を提供していた。

### 6.7.1 Apache MINA のデフォルトのスレッド・モデル

(注意: Apache MINA の[説明のページ](#)は現在一部はバージョン 1.0 のもので、今回のように 2.0 を使用するときは注意が必要である。)

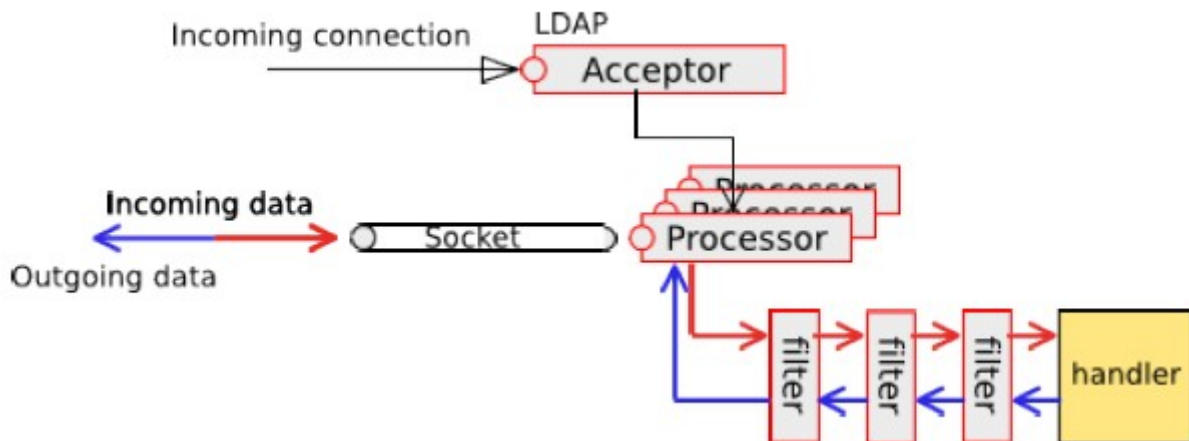
Apache MINA には IO プロセッサ・スレッド (IoProcessor スレッドまたはワーカ・スレッド(worker threads)ともいう) と、Acceptor スレッド (サーバの場合) と Connector スレッド (クライアントの場合) という、ソケットを常に管理し IO プロセッサ・スレッドにその接続を渡しているスレッドたち、の 3 つのスレッドのタイプが存在している。

SocketAcceptor あるいは SocketConnector が自らの IO プロセッサ・スレッドたちをスレッド・プールから生成している。デフォルトの設定では、IO スレッドはあるスレッド・プール (SimpleIoProcessorPool) から割り当てられる:

- コア・プール・サイズは `Runtime.getRuntime().availableProcessors() + 1`;

下図は MINA の[プレゼンテーション資料](#)にある動作図である。

図 6-8: MINA の動作



接続を処理するアクセプタ (通常は `NioSocketAcceptor`) がプロセッサ (通常は `NioProcessor`) のスレッド・プール (`SimpleIoProcessorPool`) を用意している。プロセッサはスレッドであり、アクセプタはコンストラクト時に `IoProcessor` スレッドたちを走らせる。アクセプタ自身も前述のようにスレッドであり、`bind()` が呼ばれたときに開始し、`unbind()` が呼ばれたときに停止する。IO プロセッサのスレッドはアクセプタからの接続 (`IoSession`)、あるいは送受信等のイベントの発生に応じ、これをフィルタ・チェーン経由でハンドラにわたす。IO プロセッサのスレッドは複数の `IoSession` を受け持っていて、そのセッションの発生から終了までを監視する。

### 6.7.2 MINA のスレッド・モデルの設定

Apache MINA を使って最適な効率を得るためには (特に大量の同時クライアント接続を扱うような場合に)、そのスレッド・モデルを注意して設定する必要がある。スレッド・モデルの設定はそのアプリケーションに強く依

存する。ユーザがそのスレッド・モデルを実現するためには、IO ハンドラ・スレッドと `ExecutorFilter` スレッドの、2つのタイプのスレッドを設定することになる。

#### 6.7.2.1 IO ハンドラ・スレッド(`NioProcessor`)数の設定

スレッド・モデルの設定は、まず適正な `IoService` あたりの IO プロセッサ・スレッドの数 (`SimpleIoProcessorPool` のサイズ)を決めることから始まる。上記のように、IO プロセッサ・スレッドはソケットと通信している。デフォルトでは IO プロセッサ・スレッドは複数となっていて、クライアント接続したソケットたちはアクセプタによってこれらのスレッドたちに順番に分配される。

データ・ベースへのアクセスなどのために、クライアント間との IO 操作にかかる時間が非常に長いアプリケーションの場合には、`ExecutorFilter` スレッドをつかわないときは、このスレッドが IO ハンドラ・アダプタ (`IoHandlerAdapter`)のメソッドを通過する為、そのアプリケーションに適した IO プロセッサ・スレッド数を選択する。スレッドの設定は表 6-6 で示したようアクセプタのコンストラクタで指定する。実際は、`NioProcessor` は `IoProcessor` インターフェイスを実装したもので、そのコンストラクタは `NioProcessor(Executor executor)`と `Executor` を指定していて、その `Executor` が所定の数のスレッドを用意している。

#### 6.7.2.2 `ExecutorFilter` スレッドの設定

Apache MINA では、受信したクライアントからのデータ(読み出しメッセージ)は前加工のためにフィルタ・チェーンに渡される。そのメッセージがフィルタ・チェーンを通過したあとで、そのメッセージは登録されている IO ハンドラ・アダプタの `messageReceived` メソッドに渡される。`messageReceived` メソッドで時間がかかるようなアプリケーションでは、それが隘路になってクライアントとの IO が待たされ、全体性能を落とすので、[前述の `ExecutorFilter`](#) を使用することになる。このフィルタは、IO プロセッサ・スレッドからメッセージを引き継いで、新たなスレッド・プールからのスレッドで IO ハンドラ・アダプタの `messageReceived` メソッドなどのメソッドを呼ぶ。従って、IO プロセッサ・スレッドは何もしないで即座に戻る事が出来る。[`ExecutorFilter` の効果は大きい](#)。そのことは次の節で具体的に実験で確認できる。

### 6.7.3 簡単なテスト・プログラムによる確認

#### 6.7.3.1 テスト用クライアント(`MultiClientSimulator`)

最初に、次のような指定した数のクライアントを一斉に発生させる簡単なシミュレータ・プログラム (`MultiClientSimulator`)を用意する。

```
package TCPIP_Programming;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintStream;
import java.net.Socket;
import java.util.Calendar;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

```

public class MultiClientSimulator implements Runnable{
    // 番号付きコンストラクタ
    int client_number;
    MultiClientSimulator(int i){
        client_number = i;
    }

    // ロガー設定
    private final static Logger LOGGER = LoggerFactory.getLogger(MultiClientSimulator.class);

    // メイン・メソッド
    public static void main(String[] args) throws IOException {
        int clients = 1;
        if (args.length == 1){clients = Integer.parseInt(args[0].trim());}
        // Executorの作成
        ExecutorService threadExecutor = Executors.newFixedThreadPool( clients );
        int i;
        for (i = 1; i <= clients; i++){
            // start threads and place in runnable state
            threadExecutor.execute( new MultiClientSimulator(i) ); // タスクたちの開始
        }
        threadExecutor.shutdown(); // ワーカー・スレッドたちのシャットダウン
        LOGGER.info( "スレッドたちの開始, mainは終了" );
    }

    // スレッド・タスク
    public void run(){
        try{
            // サーバと接続
            Socket client_socket = new Socket("localhost", 1000);
            // ストリームの生成
            BufferedReader downward_stream = new BufferedReader(
                new InputStreamReader(client_socket.getInputStream()),
                "Windows-31J");
            PrintStream upward_stream = new
            PrintStream(client_socket.getOutputStream(), true, "Windows-31J");
            // タイムスタンプつきテスト行送信
            upward_stream.printf("クライアント:%d   %tTS \n", client_number,
            Calendar.getInstance());
            // エコーバック受信
            String received_data = downward_stream.readLine();
            String arrived_time = String.format(" %tTS", Calendar.getInstance());
            LOGGER.info( received_data + arrived_time);
            // 接続の切断
            client_socket.close();
        }
        catch (Exception e){
            e.printStackTrace();
            System.exit(-1);
            return;
        }
    }
}

```

このプログラムは **main** の引数として発生させるクライアントの数を指定する(指定しないとひとつだけのクライアントが発生される)。クライアントはサーバに接続したら自分のクライアント番号をタイムスタンプ付きでサーバに送信する。次にサーバからテキスト・メッセージが返されてきたらそれを受けた時刻とともにログ出力して、サーバとの接続を切断する。サーバ側では接続が切断されたことを知ってそのクライアントへのサービスを終了する。



Eclipse のデバッグ・パースペクト上で最初に `SimpleServerThread` を実行させ、更にこの `MultiClientSimulator` を引数を 20 として実行させると、例えば次のような結果が得られる。

```
7 [main] INFO TCPIP_Programming.MultiClientSimulator - スレッドたちの開始, mainは終了
561 [pool-1-thread-14] INFO TCPIP_Programming.MultiClientSimulator - Server received(21 chraracters): クライアント: 14 10:49:20S 10:49:20S
562 [pool-1-thread-17] INFO TCPIP_Programming.MultiClientSimulator - Server received(21 chraracters): クライアント: 17 10:49:20S 10:49:20S
562 [pool-1-thread-15] INFO TCPIP_Programming.MultiClientSimulator - Server received(21 chraracters): クライアント: 15 10:49:20S 10:49:20S
562 [pool-1-thread-1] INFO TCPIP_Programming.MultiClientSimulator - Server received(20 chraracters): クライアント: 1 10:49:20S 10:49:20S
562 [pool-1-thread-19] INFO TCPIP_Programming.MultiClientSimulator - Server received(21 chraracters): クライアント: 19 10:49:20S 10:49:20S
564 [pool-1-thread-5] INFO TCPIP_Programming.MultiClientSimulator - Server received(20 chraracters): クライアント: 5 10:49:20S 10:49:20S
563 [pool-1-thread-3] INFO TCPIP_Programming.MultiClientSimulator - Server received(20 chraracters): クライアント: 3 10:49:20S 10:49:20S
563 [pool-1-thread-2] INFO TCPIP_Programming.MultiClientSimulator - Server received(20 chraracters): クライアント: 2 10:49:20S 10:49:20S
564 [pool-1-thread-7] INFO TCPIP_Programming.MultiClientSimulator - Server received(20 chraracters): クライアント: 7 10:49:20S 10:49:20S
562 [pool-1-thread-4] INFO TCPIP_Programming.MultiClientSimulator - Server received(20 chraracters): クライアント: 4 10:49:20S 10:49:20S
563 [pool-1-thread-8] INFO TCPIP_Programming.MultiClientSimulator - Server received(20 chraracters): クライアント: 8 10:49:20S 10:49:20S
563 [pool-1-thread-11] INFO TCPIP_Programming.MultiClientSimulator - Server received(21 chraracters): クライアント: 11 10:49:20S 10:49:20S
566 [pool-1-thread-18] INFO TCPIP_Programming.MultiClientSimulator - Server received(21 chraracters): クライアント: 18 10:49:20S 10:49:20S
567 [pool-1-thread-12] INFO TCPIP_Programming.MultiClientSimulator - Server received(21 chraracters): クライアント: 12 10:49:20S 10:49:20S
568 [pool-1-thread-9] INFO TCPIP_Programming.MultiClientSimulator - Server received(20 chraracters): クライアント: 9 10:49:20S 10:49:20S
563 [pool-1-thread-13] INFO TCPIP_Programming.MultiClientSimulator - Server received(21 chraracters): クライアント: 13 10:49:20S 10:49:20S
565 [pool-1-thread-6] INFO TCPIP_Programming.MultiClientSimulator - Server received(20 chraracters): クライアント: 6 10:49:20S 10:49:20S
569 [pool-1-thread-10] INFO TCPIP_Programming.MultiClientSimulator - Server received(21 chraracters): クライアント: 10 10:49:20S 10:49:20S
564 [pool-1-thread-16] INFO TCPIP_Programming.MultiClientSimulator - Server received(21 chraracters): クライアント: 16 10:49:20S 10:49:20S
564 [pool-1-thread-20] INFO TCPIP_Programming.MultiClientSimulator - Server received(21 chraracters): クライアント: 20 10:49:20S 10:49:20S
```

これを見ると、1 秒以内に 20 個のクライアントのセッションが終了している。またログのタイムスタンプを見ると、最小が 561、最高が 569 と、8 ミリ秒の間に 20 個の応答が一斉に返ってきていることが判る。プログラム上はクライアント 1 からクライアント 20 まで順番に開始させているが、応答してきた順番はそうはなっていない。クライアント 10 が最後になっている。これは NIO プロセサは順番に生成されているものの、受信がほぼ同時な為、取り出しのソケットの順番が揃わないことによるもので気にしなくても良い。このことは `MultiClientSimulator` の `threadExecutor.execute( new MultiClientSimulator(i) );` の行の後に例えば `Thread.sleep(1000);` という行を置いてみれば、正しい順番になることから確認される (但し `main` のメソッドがスローする例外を `IOException` から `Exception` に変更のこと)。

### 6.7.3.2 MINA のデフォルト設定のサーバのテスト

次にこのクライアント・シミュレータを使って、MINA のデフォルトのスレッド設定のサーバにアクセスしてみよう。この場合は「[処理時間計測のフィルタ\(ProfilerTimerFilter\)](#)」の節で紹介した MesRecEventProfilerTestServer を使ってみる。このプログラムでは、メッセージ受信イベント処理時間が短いので、以下のように 5 秒間スレッドをこのメソッド内で待たせることにする。

```
public void messageReceived( IoSession session, Object message)
throws Exception {
    // 受信したオブジェクトはStringとして渡される
    String rmes = (String)message;
    LOGGER.info( "Received : " + rmes);
    // 日本語のヘッダを付けて送り返す
    String smes = "MINA サーバ受信 : " + rmes;
    // クライアントがendを送信してきたら
    if( rmes.trim().startsWith("end") ) {
        session.write("endを受信しました");
        session.close(false);
        return;
    }
    Thread.sleep(5000);
    session.write(smes);
}
}
```

Eclipse のデバッグ・パースペクト上で最初にこのように変更した MesRecEventProfilerTestServer を実行させ、更にこの MultiClientSimulator を引数を 20 として実行させると、例えば次のような結果が得られる。

```
MINA Echo サーバ開始
41533 [NioProcessor-1] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 10 16:29:46S
41533 [NioProcessor-4] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 7 16:29:46S
41533 [NioProcessor-2] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 1 16:29:46S
41533 [NioProcessor-5] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 9 16:29:46S
41533 [NioProcessor-3] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 8 16:29:46S
46538 [NioProcessor-4] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message processed in (mS) : 5005
46538 [NioProcessor-2] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message processed in (mS) : 5005
46538 [NioProcessor-3] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message processed in (mS) : 5005
46538 [NioProcessor-1] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message processed in (mS) : 5005
46538 [NioProcessor-5] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message processed in (mS) : 5005
46539 [NioProcessor-3] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Averaged process time in (mS) : 5005.0
46539 [NioProcessor-2] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Averaged process time in (mS) : 5005.0
46539 [NioProcessor-5] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Averaged process time in (mS) : 5005.0
46539 [NioProcessor-1] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Averaged process time in (mS) : 5005.0
46539 [NioProcessor-4] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Averaged process time in (mS) : 5005.0
46540 [NioProcessor-1] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 19 16:29:46S
46542 [NioProcessor-3] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 5 16:29:46S
46543 [NioProcessor-4] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 20 16:29:46S (以下省略)
```

このように、NIO プロセサは 5 個のみが用意されている。従って 20 のクライアント処理には 20 秒を要している。MultiClientSimulator のログを見ると、クライアント 20 では 20 秒経過してサーバの応答を受信している。

これは NIO プロセサのスレッド・プールのデフォルトのサイズが DEFAULT\_SIZE = Runtime.getRuntime().availableProcessors() + 1 となっていて、筆者の環境(クワッド・コア)の場合は Runtime.getRuntime().availableProcessors() が 4 の為である。

### 6.7.3.3 スレッド枯渇時の返信の遅れ

問題は、このようなスレッド枯渇状況のときのクライアント側のログでわかるように、MesRecEventProfilerTestServer のログ出力では正常に 5 秒ごとに 5 個ずつ受信メッセージが処理され、クライアントへの送信メッセージも session に渡されているにも関わらず、クライアントには 5 秒ごとに 5 つずつ返送

メッセージが届いていないということである。これでは平均サービス時間が 10 秒を超えてしまう。これはどうしてであろうか？

以下は、MesRecEventProfilerTestServer のコードに messageSent というメソッドを置いて、“message sent”というメッセージをログ出力するようにしたときの実験例である：

```
MINA Echo サーバ開始
16210 [NioProcessor-4] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 18 10:11:28S
16210 [NioProcessor-5] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 10 10:11:28S
16210 [NioProcessor-1] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 1 10:11:28S
16210 [NioProcessor-2] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 7 10:11:28S
16210 [NioProcessor-3] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 14 10:11:28S
21216 [NioProcessor-3] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message processed in (mS) : 5006
21216 [NioProcessor-5] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message processed in (mS) : 5006
21216 [NioProcessor-2] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message processed in (mS) : 5006
21217 [NioProcessor-3] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 5 10:11:28S
21217 [NioProcessor-5] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 19 10:11:28S
21216 [NioProcessor-4] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message processed in (mS) : 5006
21216 [NioProcessor-1] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message processed in (mS) : 5006
21217 [NioProcessor-2] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 12 10:11:28S
21218 [NioProcessor-1] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 4 10:11:28S
21218 [NioProcessor-4] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 13 10:11:28S
26218 [NioProcessor-4] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message processed in (mS) : 5000
26218 [NioProcessor-1] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message processed in (mS) : 5000
26218 [NioProcessor-3] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message processed in (mS) : 5001
26218 [NioProcessor-4] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 8 10:11:28S
26218 [NioProcessor-5] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message processed in (mS) : 5001
26218 [NioProcessor-2] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message processed in (mS) : 5001
26219 [NioProcessor-1] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 16 10:11:28S
26219 [NioProcessor-3] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 6 10:11:28S
26219 [NioProcessor-2] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 2 10:11:28S
26219 [NioProcessor-5] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 17 10:11:28S
31218 [NioProcessor-4] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message processed in (mS) : 5000
31218 [NioProcessor-4] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 20 10:11:28S
31219 [NioProcessor-3] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message processed in (mS) : 5001
31219 [NioProcessor-1] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message processed in (mS) : 5000
31219 [NioProcessor-3] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 3 10:11:28S
31219 [NioProcessor-2] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message processed in (mS) : 5000
31219 [NioProcessor-5] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message processed in (mS) : 5000
31219 [NioProcessor-2] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 11 10:11:28S
31220 [NioProcessor-1] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 9 10:11:28S
31220 [NioProcessor-5] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 15 10:11:28S
36218 [NioProcessor-4] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message processed in (mS) : 5000
36219 [NioProcessor-3] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message processed in (mS) : 5000
36220 [NioProcessor-2] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message processed in (mS) : 5001
36220 [NioProcessor-4] INFO TCPIP_Programming.MesRecEventProfilerTestServer - message sent
36221 [NioProcessor-4] INFO TCPIP_Programming.MesRecEventProfilerTestServer - message sent
36221 [NioProcessor-4] INFO TCPIP_Programming.MesRecEventProfilerTestServer - message sent
36221 [NioProcessor-4] INFO TCPIP_Programming.MesRecEventProfilerTestServer - message sent
36222 [NioProcessor-3] INFO TCPIP_Programming.MesRecEventProfilerTestServer - message sent
36222 [NioProcessor-3] INFO TCPIP_Programming.MesRecEventProfilerTestServer - message sent
36222 [NioProcessor-3] INFO TCPIP_Programming.MesRecEventProfilerTestServer - message sent
36222 [NioProcessor-3] INFO TCPIP_Programming.MesRecEventProfilerTestServer - message sent
36223 [NioProcessor-5] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message processed in (mS) : 5003
36223 [NioProcessor-2] INFO TCPIP_Programming.MesRecEventProfilerTestServer - message sent
36223 [NioProcessor-2] INFO TCPIP_Programming.MesRecEventProfilerTestServer - message sent
36223 [NioProcessor-2] INFO TCPIP_Programming.MesRecEventProfilerTestServer - message sent
36223 [NioProcessor-1] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message processed in (mS) : 5003
36224 [NioProcessor-1] INFO TCPIP_Programming.MesRecEventProfilerTestServer - message sent
36224 [NioProcessor-1] INFO TCPIP_Programming.MesRecEventProfilerTestServer - message sent
36224 [NioProcessor-1] INFO TCPIP_Programming.MesRecEventProfilerTestServer - message sent
36225 [NioProcessor-5] INFO TCPIP_Programming.MesRecEventProfilerTestServer - message sent
36225 [NioProcessor-5] INFO TCPIP_Programming.MesRecEventProfilerTestServer - message sent
36225 [NioProcessor-5] INFO TCPIP_Programming.MesRecEventProfilerTestServer - message sent
```

このことから、実際のネットワーク送信のための処理への割当てが遅れていることが分かる。これは messageReceived のイベント(IoEventType.WRITE)の優先度が高くなっていることによる。

イベントの優先度設定は出来ないので、この問題の手っ取り早い対策はやはり

- IO プロセサ・スレッドのプール・サイズを大きくする

- 後述の `ExecutorFilter` を使用して、IO プロセサ・スレッドを早く戻すの 2 つとなる。

#### 6.7.3.4 クライアント接続継続との関係

ちなみにこの状態で更に `MultiClientSimulator` 側でも接続を切るのに 5 秒間のタイマを下記のように組み込んでみよう。

```
public void run(){
    try{
        // サーバと接続
        Socket client_socket = new Socket("localhost", 1000);
        // ストリームの生成
        BufferedReader downward_stream = new BufferedReader(
            new InputStreamReader(client_socket.getInputStream(),
"Windows-31J"));
        PrintStream upward_stream = new
PrintStream(client_socket.getOutputStream(), true, "Windows-31J");
        // タイムスタンプつきテスト行送信
        upward_stream.printf("クライアント:%d   %tTS \n", client_number,
Calendar.getInstance());
        // エコーバック受信
        String received_data = downward_stream.readLine();
        String arrived_time = String.format(" %tTS", Calendar.getInstance());
        LOGGER.info( received_data + arrived_time);
        // 接続の切断
        Thread.sleep(5000);
        client_socket.close();
    }
    catch (Exception e){
        e.printStackTrace();
        System.exit(-1);
        return;
    }
}
```

こうしても実際のクライアント処理能力には全く影響が無いことが実験で確認されよう。NIO プロセサは NIO でいう非ブロッキング動作をしていて、クライアント接続と開放を待ってはいないからである。

#### 6.7.3.5 NIO プロセサ・スレッド・プールのサイズ増加による処理能力向上

スレッド枯渇により処理能力低下への対策の第 1 は NIO プロセサ・スレッドのプール・サイズを大きくすることである。この数を変更したい場合には、以下のようにサイズを指定する:

```
SocketAcceptor acceptor = new SocketAcceptor(10);
```

これにより、どのように処理能力が改善されるかは、各自試して見られたい。

#### 6.7.3.6 スレッド処理のためのフィルタ追加による処理能力向上

それでは `MesRecEventProfilerTestServer` には 5 秒間のタイマ付き、`MultiClientSimulator` では 5 秒間のタイマなし、そして NIO プロセサ・スレッド・プールのサイズはデフォルトの状態に戻してみよう。

この状態では、5 つの NIO プロセサのスレッドが `messageReceived` メソッドを抜けるのに 5 秒間かかっているの  
で、結局クライアントからのメッセージ受信処理能力は平均毎秒 1 クライアントということになる。[「スレッド処理](#)

[のためのフィルタ\(ExecutorFilter\)の節](#)で説明したように、このフィルタがそのような状況での有効な解決策になる。

ExecutorFilter というのは到来した IO プロセサ・スレッドが持ってきた IO イベントたちを `ava.util.concurrent.Executor` 実装物に引き渡す為の `IoFilter` である。これらのイベントはこの `Executor` (通常はスレッド・プール)を介して次の `IoFilter` に引き渡される。従って NIO プロセサ・スレッドはこの `Executor` にイベントを引き渡した後は直ちに自分のスレッド・プールに戻る事が出来、NIO プロセサ・スレッドは少なくとも効率的に使用できることになる。`ExecutorFilter` は IO フィルタ・チェーンのどこにでもいくつでも配置できるので、いろんなスレッド・モデルを構成出来る。`ExecutorFilter` を使わないときは、実際のサービス(ビジネス・ロジック)処理である `IoHandler` アダプタのメソッドを IO プロセサ・スレッドが実行することになる。このような使い方を「単一スレッド・モデル(single thread model)」と称している。データベースやリモートのウェブ・サービスからの応答を待つようなアプリケーションでは、そのような待機状態がサービス品質を低下させてしまう。

`ExecutorFilter` の追加は簡単である。

```
acceptor.getFilterChain().addLast("threadPool", new ExecutorFilter());
```

豊富なコンストラクタが用意されており、`ExecutorFilter()`と何も指定しないと、サイズは 16 となる。

`ExecutorFilter(20)`などとサイズを指定したり、`ExecutorFilter(Executor executor)`と `Executor` を指定したりできる。またイベント毎にプールを用意することもできる。

`new ExecutorFilter(Executors.newCachedThreadPool());`という具合に、指定するスレッド・プールとしては `Executors.newCachedThreadPool()`が推奨されているので、これを使用したほうが良い。他の種類のスレッド・プールを使うと副作用で予期せぬ結果をもたらす可能性があるとして MINA が説明している。またこのフィルタをコーデック・フィルタの後に追加したほうが好ましいとも説明している。

それでは、`ExecutorFilter` の効果をこれまでの `MultiClientSimulator`、`MesRecEventProfilerTestServer`、及び `MesRecEventProfilerTestFilter` の組み合わせで確認してみよう。この実験の変更事項は次のようである：

- `MesRecEventProfilerTestServer.java`  
main のメソッドの中で、10 個のスレッド・プールからなる `ExecutorFilter` を次のように最後の場所に追加する：

```
public static void main(String[] args) throws IOException {
    // アクセプタを用意
    SocketAcceptor acceptor = new NioSocketAcceptor();
    // テキスト行のコーデックを追加
    acceptor.getFilterChain().addLast("codec",
        new ProtocolCodecFilter(
            new TextLineCodecFactory( Charset.forName("Windows-
31J"))));
    // 文字コード変換を追加
    // プロファイラ・フィルタを追加
    acceptor.getFilterChain().addLast("profiler", new
MesRecEventProfilerTestFilter());
    acceptor.getFilterChain().addLast("threadPool", new ExecutorFilter(10));
    .....以下省略
```

また `messageReceived` のメソッドのなかで、クライアントに送り返す前に 5 秒間のスリープを追加したままとし、更にログ出力するだけの `messageSent` のメソッドを追加する：

```
public void messageReceived(IOException session, Object message)
throws Exception {
    // 受信したオブジェクトはStringとして渡される
    String rmes = (String)message;
```

```

        LOGGER.info( "Received : " + rmes);
        // 日本語のヘッダを付けて送り返す
        String smes = "MINA サーバ受信 : " + rmes;
        // クライアントがendを送信してきたら
        if( rmes.trim().startsWith("end") ) {
            session.write("endを受信しました");
            session.close(false);
            return;
        }

        Thread.sleep(5000); // 処理時間がかかることのシミュレーション
        session.write(smes);
    }

    //messageSent イベントの発生時間を調べるテスト用コード
    public void messageSent( IoSession session, Object message)
        throws Exception {
        LOGGER.info("message sent");
        // テスト用、なにもしない
    }
}

```

- MesRecEventProfilerTestFilter.java

平均処理時間のログ出力の個所をコメント・アウトする:

```

        // 処理時間をミリ秒でログする
        LOGGER.info("Received message processed in (mS) : " +
            (end - start));
        // 次に平均処理時間をミリ秒でログする
// LOGGER.info("Averaged process time (mS) : " +
//     getAverageTime( IoEventType.MESSAGE_RECEIVED));

```

この状態で、Eclipse のデバッグのパスペクト上で:

1. 最初に MesRecEventProfilerTestServer を実行する
2. 次に MultiClientSimulator を引数を 20、つまりクライアント数は 20 として実行する

そうすると、サーバのログ出力は次のようになる:

```

MINA Echo サーバ開始
11445 [NioProcessor-4] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message processed in
(mS) : 0
11445 [NioProcessor-2] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message processed in
(mS) : 0
11445 [pool-3-thread-4] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 17
16:05:28S
11445 [pool-3-thread-1] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 2
16:05:28S
11445 [pool-3-thread-2] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 6
16:05:28S
11445 [NioProcessor-5] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message processed in
(mS) : 1
11446 [NioProcessor-5] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message processed in
(mS) : 0
11445 [pool-3-thread-5] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 11
16:05:28S
11445 [NioProcessor-1] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message processed in
(mS) : 1
11445 [NioProcessor-3] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message processed in
(mS) : 0
11446 [pool-3-thread-3] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 18
16:05:28S
11447 [NioProcessor-3] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message processed in
(mS) : 0
11447 [NioProcessor-1] INFO TCPIP_Programming.MesRecEventProfilerTestFilter - Received message processed in
(mS) : 0
11447 [pool-3-thread-7] INFO TCPIP_Programming.MesRecEventProfilerTestServer - Received : クライアント : 4
16:05:28S

```



この出力結果を見れば、次のことがわかる:

1. NIOProcessor のスレッドは処理時間ゼロで戻っている。即ち ExecutorFilter で 10 個のスレッドのプールを持っている Executor にそのイベントを引き継いだら、後はすることが無くなっている。従って、20 個の NIOProcessor のスレッドはサーバが開始して 11.445 秒目付近で総て処理を終了している。
2. またこの時間に ExecutorFilter の 10 個のスレッドが messageReceived メソッドを通過している。
3. その 5 秒後に、再度 ExecutorFilter の 10 個のスレッドが messageReceived メソッドを通過している。
4. そのまた 5 秒後に、ExecutorFilter の 10 個のスレッドが 2 回 messageSent メソッドを通過している。これは NIOProcessor のスレッドがこのイベントを Executor に引き継いだ為である。

従ってこのサーバのスループットは、NIOProcessor のスレッド・プールではなくて、ExecutorFilter のスレッド・プールによって決まることになる。そのことは次に示すクライアント側のログを見れば確認できる。

```
8 [main] INFO TCPIP_Programming.MultiClientSimulator - スレッドたちの開始, mainは終了
5572 [pool-1-thread-9] INFO TCPIP_Programming.MultiClientSimulator - MINAサーバ受信 : クライアント : 9 16:05:28S
16:05:33S
5572 [pool-1-thread-13] INFO TCPIP_Programming.MultiClientSimulator - MINAサーバ受信 : クライアント : 13 16:05:28S
16:05:33S
5572 [pool-1-thread-3] INFO TCPIP_Programming.MultiClientSimulator - MINAサーバ受信 : クライアント : 3 16:05:28S
16:05:33S
5572 [pool-1-thread-17] INFO TCPIP_Programming.MultiClientSimulator - MINAサーバ受信 : クライアント : 17 16:05:28S
16:05:33S
5572 [pool-1-thread-11] INFO TCPIP_Programming.MultiClientSimulator - MINAサーバ受信 : クライアント : 11 16:05:28S
16:05:33S
5573 [pool-1-thread-4] INFO TCPIP_Programming.MultiClientSimulator - MINAサーバ受信 : クライアント : 4 16:05:28S
16:05:33S
5573 [pool-1-thread-12] INFO TCPIP_Programming.MultiClientSimulator - MINAサーバ受信 : クライアント : 12 16:05:28S
16:05:33S
5573 [pool-1-thread-18] INFO TCPIP_Programming.MultiClientSimulator - MINAサーバ受信 : クライアント : 18 16:05:28S
16:05:33S
5572 [pool-1-thread-6] INFO TCPIP_Programming.MultiClientSimulator - MINAサーバ受信 : クライアント : 6 16:05:28S
16:05:33S
5573 [pool-1-thread-2] INFO TCPIP_Programming.MultiClientSimulator - MINAサーバ受信 : クライアント : 2 16:05:28S
16:05:33S
10571 [pool-1-thread-1] INFO TCPIP_Programming.MultiClientSimulator - MINAサーバ受信 : クライアント : 1 16:05:28S
16:05:38S
10571 [pool-1-thread-16] INFO TCPIP_Programming.MultiClientSimulator - MINAサーバ受信 : クライアント : 16
16:05:28S 16:05:38S
10572 [pool-1-thread-14] INFO TCPIP_Programming.MultiClientSimulator - MINAサーバ受信 : クライアント : 14
16:05:28S 16:05:38S
10573 [pool-1-thread-15] INFO TCPIP_Programming.MultiClientSimulator - MINAサーバ受信 : クライアント : 15
16:05:28S 16:05:38S
10573 [pool-1-thread-10] INFO TCPIP_Programming.MultiClientSimulator - MINAサーバ受信 : クライアント : 10
16:05:28S 16:05:38S
10573 [pool-1-thread-19] INFO TCPIP_Programming.MultiClientSimulator - MINAサーバ受信 : クライアント : 19
16:05:28S 16:05:38S
10573 [pool-1-thread-20] INFO TCPIP_Programming.MultiClientSimulator - MINAサーバ受信 : クライアント : 20
16:05:28S 16:05:38S
10574 [pool-1-thread-7] INFO TCPIP_Programming.MultiClientSimulator - MINAサーバ受信 : クライアント : 7 16:05:28S
16:05:38S
10574 [pool-1-thread-8] INFO TCPIP_Programming.MultiClientSimulator - MINAサーバ受信 : クライアント : 8 16:05:28S
16:05:38S
10574 [pool-1-thread-5] INFO TCPIP_Programming.MultiClientSimulator - MINAサーバ受信 : クライアント : 5 16:05:28S
16:05:38S
```

即ち、クライアントのプログラムが開始して 5.572 秒で、10 個のエコーバックが受信されており、更にその 5 秒後の 10.571 秒目から残りの 10 個のエコーバックが受信されている。

#### 6.7.4 スレッド・モデルの選択



これまでの実験から、MINA では柔軟なスレッド・モデルが選択できることが理解されたであろう。実際の選択に当たっては次の事柄が重要であろう:

- マシンの能力
- ヒープ領域のサイズ
- IO ハンドラ・アダプタでのブロッキングを生じる要素とその待機時間の平均と最大値
- 対象とするクライアント数
- 必要なスループット、あるいは最大待ち時間

これらの総合的検討から、必要なスレッド・モデルを選択したら、商用配備に先行して、シミュレータによる負荷試験(社内 LAN だけでなく、いろんな状態が起き得るインターネット上でも)を入念に行うことが、特に高信頼のアプリケーションでは欠かせない。

## 6.8節 複数のサーバの実装

MINA では、複数のアクセプタやコネクタを実装できる。ひとつの例として、いままでの受付ポート番号 1000 のエコー・サーバに加えて、受付ポート番号 9123 番の TIME という現在時間を返すサーバを実装した MINADualModeServer というクラスを試してみよう。エコー・サーバは Window-31J(SJIS と呼ばれることが多いが、一部追加されている)を使い、タイム・サーバは EUC-JP(EUC と呼ばれている)を使っている。以下に MINADualModeServer と TimeProtocolHandler のコードを示す。

### MINADualModeServer:

```
package TCPIP_Programming;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.charset.Charset;

import org.apache.mina.filter.codec.ProtocolCodecFilter;
import org.apache.mina.filter.codec.textline.TextLineCodecFactory;
import org.apache.mina.filter.logging.LoggingFilter;
import org.apache.mina.transport.socket.SocketAcceptor;
import org.apache.mina.transport.socket.nio.NioSocketAcceptor;

public class MINADualModeServer {
    // 受付ポート番号
    private static final int ECHO_PORT = 1000;
    private static final int TIME_PORT = 9123;

    public static void main(String[] args) throws IOException {
        // アクセプタを用意
        SocketAcceptor acceptor1 = new NioSocketAcceptor();
        acceptor1.getFilterChain().addLast("logger", new LoggingFilter()); // デフォルトのロ
ガー追加
        acceptor1.getFilterChain().addLast("codec",
            new ProtocolCodecFilter(
                new TextLineCodecFactory( Charset.forName("Windows-
31J")))); // 文字コード変換を追加
        acceptor1.setReuseAddress( false ); // 必要があればここを true にする
        acceptor1.setHandler(new AnotherEchoProtocolHandler());
        // ソケット・アドレスをバインドして、サーバを開始
        acceptor1.bind(new InetSocketAddress( ECHO_PORT ));
    }
}
```

```

System.out.println("MINA Echo サーバ開始");
// アクセプタを用意
SocketAcceptor acceptor2 = new NioSocketAcceptor();
acceptor2.getFilterChain().addLast("logger", new LoggingFilter()); // デフォルトのロ
ガー追加
acceptor2.getFilterChain().addLast("codec",
    new ProtocolCodecFilter(
        new
TextLineCodecFactory( Charset.forName("EUC_JP")))); // 文字コード変換を追加
acceptor2.setReuseAddress( false ); // 必要があればここを true にする
acceptor2.setHandler(new TimeProtocolHandler());
// ソケット・アドレスをバインドして、サーバを開始
acceptor2.bind(new InetSocketAddress(TIME_PORT));
System.out.println("MINA Time サーバ開始");
// 1分ごとにスループットを報告
for (;;) {
    System.out.println("Echo throughput, R: " +
acceptor1.getStatistics().getReadBytesThroughput() + ", W: " +
acceptor1.getStatistics().getWrittenBytesThroughput());
    System.out.println("Time throughput, R: " +
acceptor2.getStatistics().getReadBytesThroughput() + ", W: " +
acceptor2.getStatistics().getWrittenBytesThroughput());
    try {
        Thread.sleep(60 * 1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
}

```

このサーバアクセプタのフィルタはともにロガーとテキスト行コーデックを使用しているが、文字コードが異なっていることに注意されたい。

#### TimeProtocolHandler:

```

package TCPIP_Programming;

import java.util.Date;

import org.apache.mina.core.service.IoHandlerAdapter;
import org.apache.mina.core.session.IdleStatus;
import org.apache.mina.core.session.IoSession;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class TimeProtocolHandler extends IoHandlerAdapter{
    private final static Logger LOGGER = LoggerFactory.getLogger(EchoProtocolHandler.class);

    @Override
    public void sessionCreated(IoSession session) {
        session.getConfig().setIdleTime(IdleStatus.BOTH_IDLE, 10);
    }

    @Override
    public void sessionClosed(IoSession session) throws Exception {
        LOGGER.info("TIME_CLOSED");
    }

    @Override
    public void sessionOpened(IoSession session) throws Exception {
        LOGGER.info("TIME_OPENED");
        Date date = new Date();
        session.write( "現在の時間は : " + date.toString());
    }
}

```

```

        System.out.println("Time Message written...");
    }

    @Override
    public void sessionIdle( IoSession session, IdleStatus status) {
        LOGGER.info("*** TIME_IDLE # " + session.getIdleCount(IdleStatus.BOTH_IDLE) + " ***");
        Date date = new Date();
        session.write( "現在の時間は : " + date.toString());
    }

    @Override
    public void exceptionCaught( IoSession session, Throwable cause) {
        session.close(true);
    }

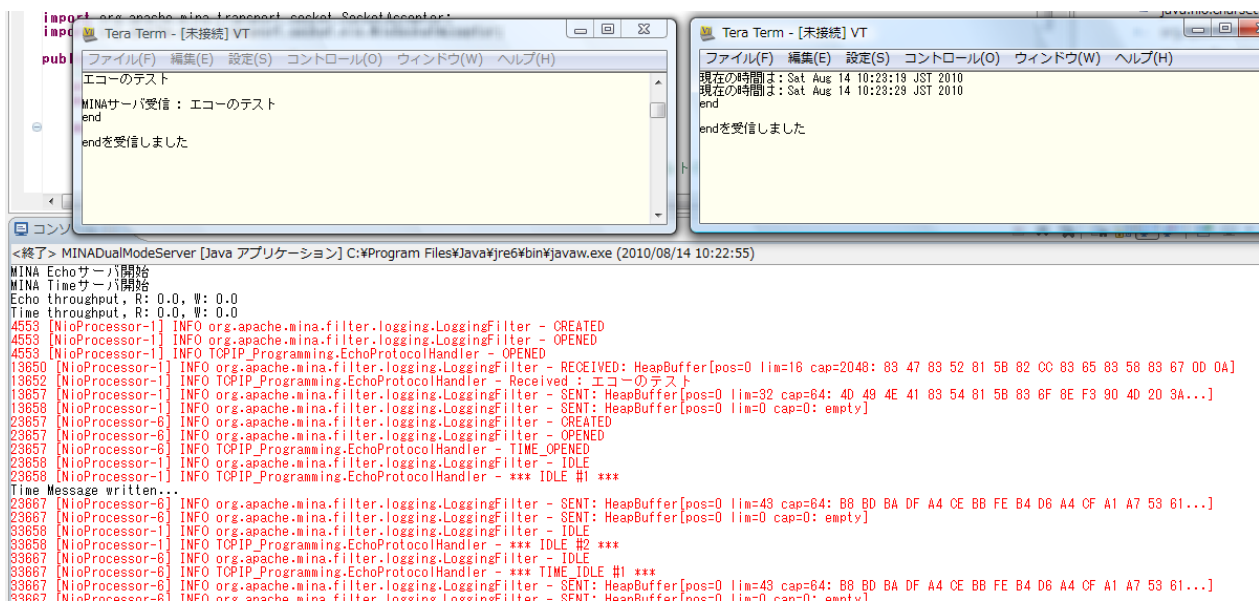
    @Override
    public void messageReceived( IoSession session, Object message)
        throws Exception {
        // 受信したオブジェクトはStringとして渡される
        String rmes = (String)message;
        // クライアントがendを送信してきたら
        if( rmes.trim().startsWith("end") ) {
            session.write("endを受信しました");
            session.close(false);
            return;
        }
        session.write("終了するときはendを入力してください");
    }
}

```

このハンドラは、接続時と10秒ごとのデフォルトのアイドル検出時間に現在時間をクライアントに返している。クライアントが **end** を送ってくると接続を解放する。

下図は2つの Tera Term でこのサーバをアクセスした例である。2番目の Tera Term は使用コードは送受信とも EUC にし、ポート番号を9123と設定すること。

図 6-9: MINADualModeServer のアクセス例



## 6.9節 Apache MINA での UDP アプリケーション

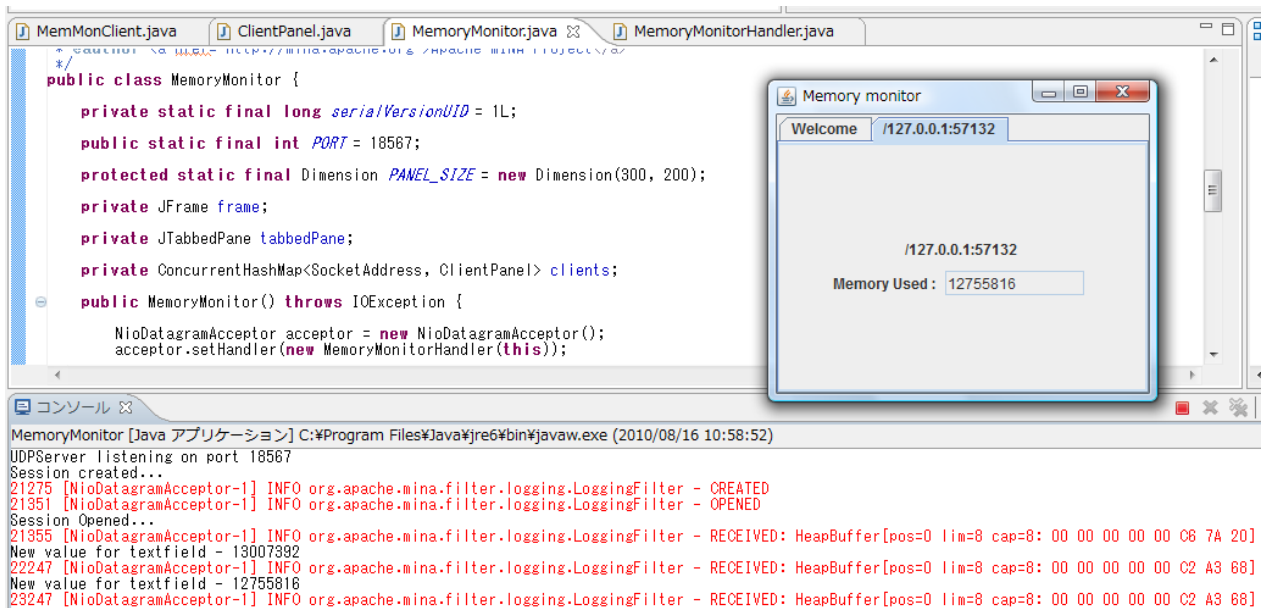
UDP では、最大ヘッダ部分を含めて 4096 バイト長の UDP データグラムを相手に送信し、相手からの受信応答の ACK を待つだけである。発信側は相手からの ACK を受けて送った UDP データグラムが相手に届いたことを知るだけである。UDP は接続や再送の手続きを持っていないので、非常に軽く、高速の処理が可能(例えば 10 バイトのデータを送信するのに TCP では合計 7 回の IP データグラムのやりとりが必要だが、UDP では ACK を含めて 2 回で済む)であり、マルチメディア・データの通信や、定期的に放送するようなテレメトリに適している。最近では通信の世界で使われている SIP で UDP が主に使われている。SIP は HTTP と似たデータグラム構造なので(但しサーバが起動も行う)、SIP サーブレットが使われている。

[サーバ用の IoService のクラス図](#)や[クライアント用の IoService のクラス図](#)で、接続という概念の無い UDP も TCP と同じようにコネクタとアクセプタがあるのは些か奇異に感じられるかもしれない。UDP メッセージでサーバと通信する為に `DatagramConnector`、クライアントと通信する為に `DatagramAcceptor` が用意されているというだけである。サーバでは待ち受けポートはあるが、クライアントとの接続は存在しない。セッションは相手から UDP メッセージを受信した、あるいは相手に送信した UDP メッセージの ACK が返った、ということで生成される。クライアントとの接続/解放処理はアプリケーションのレベルで持つことになる。具体的な例として [MINA の UDP チュートリアル](#)の `MemoryMonitor` を示す。

このアプリケーションはクライアントたちが UDP で報告してくるメモリ使用量を Java のパネルに表示するものである。これは一種の監視アプリケーションともいえよう。

このアプリケーションはクライアント用の `MemMonClient`、サーバ用の `MemoryMonitor`、`MemoryMonitorHandler`、及びサーバでの表示用の `ClientPanel` の 4 つのクラスで構成されている。下図はこれらのクラスを Eclipse 上で走らせた例である。

図 6-10: UDP ベースのメモリ・モニタの実行例



## 6.9.1 サーバのコード

サーバのメインのクラスである `MemoryMonitor.java` は次のようになっている:

```

001 package TCPIP_Programming;
002
003 /*
004  * Licensed to the Apache Software Foundation (ASF) under one
005  * or more contributor license agreements. See the NOTICE file
006  * distributed with this work for additional information
007  * regarding copyright ownership. The ASF licenses this file
008  * to you under the Apache License, Version 2.0 (the
009  * "License"); you may not use this file except in compliance
010  * with the License. You may obtain a copy of the License at
011  *
012  * http://www.apache.org/licenses/LICENSE-2.0
013  *
014  * Unless required by applicable law or agreed to in writing,
015  * software distributed under the License is distributed on an
016  * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
017  * KIND, either express or implied. See the License for the
018  * specific language governing permissions and limitations
019  * under the License.
020  *
021  */
022
023 import java.awt.BorderLayout;
024 import java.awt.Dimension;
025 import java.io.IOException;
026 import java.net.InetSocketAddress;
027 import java.net.SocketAddress;
028 import java.util.concurrent.ConcurrentHashMap;
029
030 import javax.swing.JFrame;
031 import javax.swing.JLabel;
032 import javax.swing.JPanel;

```

```

033 import javax.swing.JTabbedPane;
034
035 import org.apache.mina.core.filterchain.DefaultIoFilterChainBuilder;
036 import org.apache.mina.filter.logging.LoggingFilter;
037 import org.apache.mina.transport.socket.DatagramSessionConfig;
038 import org.apache.mina.transport.socket.nio.NioDatagramAcceptor;
039
040 /**
041  * The class that will accept and process clients in order to properly
042  * track the memory usage.
043  *
044  * @author <a href="http://mina.apache.org">Apache MINA Project</a>
045  */
046 public class MemoryMonitor {
047
048     private static final long serialVersionUID = 1L;
049
050     public static final int PORT = 18567;
051
052     protected static final Dimension PANEL_SIZE = new Dimension(300, 200);
053
054     private JFrame frame;
055
056     private JTabbedPane tabbedPane;
057
058     private ConcurrentHashMap<SocketAddress, ClientPanel> clients;
059
060     public MemoryMonitor() throws IOException {
061
062         NioDatagramAcceptor acceptor = new NioDatagramAcceptor();
063         acceptor.setHandler(new MemoryMonitorHandler(this));
064
065         DefaultIoFilterChainBuilder chain = acceptor.getFilterChain();
066         chain.addLast("logger", new LoggingFilter());
067
068         DatagramSessionConfig dcfg = acceptor.getSessionConfig();
069         dcfg.setReuseAddress(true);
070
071         frame = new JFrame("Memory monitor");
072         tabbedPane = new JTabbedPane();
073         tabbedPane.add("Welcome", createWelcomePanel());
074         frame.add(tabbedPane, BorderLayout.CENTER);
075         clients = new ConcurrentHashMap<SocketAddress, ClientPanel>();
076         frame.pack();
077         frame.setLocation(300, 300);
078         frame.setVisible(true);
079
080         acceptor.bind(new InetSocketAddress(PORT));
081         System.out.println("UDPServer listening on port " + PORT);
082     }
083
084     private JPanel createWelcomePanel() {
085         JPanel panel = new JPanel();
086         panel.setPreferredSize(PANEL_SIZE);
087         panel.add(new JLabel("Welcome to the Memory Monitor"));
088         return panel;
089     }
090
091     protected void recvUpdate(SocketAddress clientAddr, long update) {
092         ClientPanel clientPanel = clients.get(clientAddr);
093         if (clientPanel != null) {
094             clientPanel.updateTextField(update);
095         } else {
096             System.err.println("Received update from unknown client");
097         }
098     }
099
100     protected void addClient(SocketAddress clientAddr) {
101         if (!containsClient(clientAddr)) {

```

```

102         ClientPanel clientPanel = new ClientPanel(clientAddr.toString());
103         tabbedPane.add(clientAddr.toString(), clientPanel);
104         clients.put(clientAddr, clientPanel);
105     }
106 }
107
108 protected boolean containsClient(SocketAddress clientAddr) {
109     return clients.contains(clientAddr);
110 }
111
112 protected void removeClient(SocketAddress clientAddr) {
113     clients.remove(clientAddr);
114 }
115
116 public static void main(String[] args) throws IOException {
117     new MemoryMonitor();
118 }
119 }

```

UDP サーバの起動は以下の行でなされている:

```

062     NioDatagramAcceptor acceptor = new NioDatagramAcceptor();
063     acceptor.setHandler(new MemoryMonitorHandler(this));
065     DefaultIoFilterChainBuilder chain = acceptor.getFilterChain();
066     chain.addLast("logger", new LoggingFilter());
068     DatagramSessionConfig dcfg = acceptor.getSessionConfig();
069     dcfg.setReuseAddress(true);
080     acceptor.bind(new InetSocketAddress(PORT));

```

最初に 62 行で `NioDatagramAcceptor` のインスタンスを `acceptor` として生成し、第 62 行でこれに `MemoryMonitorHandler` というハンドラをセットしている。`MemoryMonitorHandler` のコンストラクタには自分のオブジェクトを知らせているが、これはハンドラのほうでこのオブジェクトが必要な為である。65 行と 66 行でロガーのフィルタを追加している。第 68、69 行で `setReuseAddress(true)` としているが、これは「[setReuseAddress メソッドについて](#)」の節で説明したように、必要ならば `true` とする。第 80 行で受け付けポート番号と IP アドレスをバインドすれば、このサーバはクライアントからの UDP データ待ちの状態になる。このクラスのメインの部分はこれだけで、この後はアクセプタのスレッドが残る。このクラスの残りは受信/標示に必要なメソッドたちである。

ハンドラのコードは特に説明するまでもなかろう。

## 6.9.2 クライアントのコード

クライアントのコードは次のようになっている:

```

001 package TCPIP_Programming;
002
003 /*
004  * Licensed to the Apache Software Foundation (ASF) under one
005  * or more contributor license agreements. See the NOTICE file
006  * distributed with this work for additional information
007  * regarding copyright ownership. The ASF licenses this file
008  * to you under the Apache License, Version 2.0 (the
009  * "License"); you may not use this file except in compliance
010  * with the License. You may obtain a copy of the License at
011  *
012  * http://www.apache.org/licenses/LICENSE-2.0
013  *
014  * Unless required by applicable law or agreed to in writing,
015  * software distributed under the License is distributed on an
016  * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
017  * KIND, either express or implied. See the License for the
018  * specific language governing permissions and limitations
019  * under the License.

```

```

020 *
021 */
022
023 import java.net.InetSocketAddress;
024
025
026 import org.apache.mina.core.buffer.IoBuffer;
027 import org.apache.mina.core.future.ConnectFuture;
028 import org.apache.mina.core.future.IoFutureListener;
029 import org.apache.mina.core.service.IoConnector;
030 import org.apache.mina.core.service.IoHandlerAdapter;
031 import org.apache.mina.core.session.IdleStatus;
032 import org.apache.mina.core.session.IoSession;
033 import org.apache.mina.transport.socket.nio.NioDatagramConnector;
034 import org.slf4j.Logger;
035 import org.slf4j.LoggerFactory;
036
037 /**
038  * Sends its memory usage to the MemoryMonitor server.
039  *
040  * @author <a href="mailto:dev@directory.apache.org">Apache Directory Project</a>
041  * @version $Rev$, $Date$
042  */
043 public class MemMonClient extends IoHandlerAdapter {
044
045     private Logger log = LoggerFactory.getLogger(MemMonClient.class);
046
047     private IoSession session;
048
049     private IoConnector connector;
050
051     /**
052      * Default constructor.
053      */
054     public MemMonClient() {
055
056         log.debug("UDPCClient::UDPCClient");
057         log.debug("Created a datagram connector");
058         connector = new NioDatagramConnector();
059
060         log.debug("Setting the handler");
061         connector.setHandler(this );
062
063         log.debug("About to connect to the server...");
064         ConnectFuture connFuture = connector
065             .connect(new InetSocketAddress("localhost",
066                 MemoryMonitor.PORT));
067
068         log.debug("About to wait.");
069         connFuture.awaitUninterruptibly();
070
071         log.debug("Adding a future listener.");
072         connFuture.addListener(new IoFutureListener<ConnectFuture>() {
073             public void operationComplete(ConnectFuture future) {
074                 if (future.isConnected()) {
075                     log.debug("...connected");
076                     session = future.getSession();
077                     try {
078                         sendData();
079                     } catch (InterruptedException e) {
080                         e.printStackTrace();
081                     }
082                 } else {
083                     log.error("Not connected...exiting");
084                 }
085             }
086         });
087     }
088

```



```

089     private void sendData() throws InterruptedException {
090         for (int i = 0; i < 30; i++) {
091             long free = Runtime.getRuntime().freeMemory();
092             IoBuffer buffer = IoBuffer.allocate(8);
093             buffer.putLong(free);
094             buffer.flip();
095             session.write(buffer);
096
097             try {
098                 Thread.sleep(1000);
099             } catch (InterruptedException e) {
100                 e.printStackTrace();
101                 throw new InterruptedException(e.getMessage());
102             }
103         }
104     }
105
106     @Override
107     public void exceptionCaught(IoSession session, Throwable cause)
108         throws Exception {
109         cause.printStackTrace();
110     }
111
112     @Override
113     public void messageReceived(IoSession session, Object message)
114         throws Exception {
115         log.debug("Session recv...");
116     }
117
118     @Override
119     public void messageSent(IoSession session, Object message)
120         throws Exception {
121         log.debug("Message sent...");
122     }
123
124     @Override
125     public void sessionClosed(IoSession session) throws Exception {
126         log.debug("Session closed...");
127     }
128
129
130     @Override
131     public void sessionIdle(IoSession session, IdleStatus status) throws Exception {
132         log.debug("Session idle...");
133     }
134
135     @Override
136     public void sessionOpened(IoSession session) throws Exception {
137         log.debug("Session opened...");
138     }
139
140     public static void main(String[] args) {
141         new MemMonClient();
142     }
143 }

```

MemMonClient というクラスは、アダプタと一体となっているので、第 43 行に見るように `IoHandlerAdapter` を継承していることに注意されたい。またロガーはフィルタではなくて `slf4j` のロガーをそのまま使っている(第 45 行)。

コンストラクタ `MemMonClient()` の内部は、少し分かりづらいかもしれない。

```

058         connector = new NioDatagramConnector();
061         connector.setHandler(this);
064         ConnectFuture connFuture = connector
065             .connect(new InetSocketAddress("localhost",
066                 MemoryMonitor.PORT));
069         connFuture.awaitUninterruptibly();

```

```

072         connFuture.addListener(new IoFutureListener<ConnectFuture>() {
073             public void operationComplete(ConnectFuture future) {
074                 if (future.isConnected()) {
075                     log.debug("...connected");
076                     session = future.getSession();
077                     try {
078                         sendData();
079                     } catch (InterruptedException e) {
080                         e.printStackTrace();
081                     }
082                 } else {
083                     log.error("Not connected...exiting");
084                 }
085             }
086         });

```

第 58 行で `NioDatagramConnector` のインスタンス `connector` を生成している。第 61 行ではこのコネクタにハンドラ(この場合は自分自身)をセットしている。

64-66 行で `MemoryMonitor` のポート番号と IP アドレス(今回はループバック・アドレス)で接続を開始している。このメソッドは `ConnectFuture` を返すが、これは非同期接続要求の `IoFuture` のひとつである。`IoFuture` は、ある I/O セッションの非同期 I/O 操作の完了を意味するインターフェイスである。完了は `IoFutureListener` によって知ることが出来る。`ConnectFuture` の一般的な使い方は次のようである:

```

IoConnector connector = ...;
ConnectFuture future = connector.connect(...);
future.join(); // 接続完了を待つ
IoSession session = future.getSession();
session.write(...);

```

しかしながら、UDP ではこれは何を意味するのであろうか?

第 69 行で、`ConnectFuture.awaitUninterruptibly()` を呼んでいて、これは非同期操作の完了を待っている。このメソッドはリスナが添付されておれば、そのリスナに通知がされる。第 72 行以降はリスナの付加のためのメソッド `addListener` で `IoFutureListener` を付加しているが、このコンストラクタでは `operationComplete` というメソッドでオーバーライドされている。このメソッドは `sendData()` という毎秒自分の空きメモリ容量をセッションに書き込むことを 30 回繰り返している。

ハンドラのメソッドたちは何もオーバーライドされていない。つまりこのクライアントの処理はイベントをもとにする必要が無い。`if (future.isConnected())` ということセッションに `IoBuffer` の内容を書き込んでいる。

クライアントのコードは UDP としてはやや複雑になっていて、UDP に `Session` というものを導入したことが原因になっている気がする。UDP を使ったアプリケーションを開発する場合は、このサンプルを参照されたい。