Java API for WebSocket

2013 年 9 月
湊 隆行

はじめに

2013 年 6 月 12 日に Java EE 7 が発表され、新機能の 1 つとして Java API for WebSocket が追加されました。

HTTP プロトコルは半二重のステートレスな接続であり、Ajax（XMLHTTPRequest）を利用した通信であっても、通信するたびに HTTP 接続→メッセージ交換→HTTP 切断というステップを踏む必要があるだけでなく、HTTP ヘッダという冗長な情報も送受信します。また、クライアントからの要求に対してサーバが応答する形の pull 通信なので、最新状態に更新するためには、クライアントから定期的にサーバに要求を出し続ける必要があります。

RFC 6455 で定義された WebSocket は HTTP をアップグレードした TCP ベースのプロトコルであり、低レイテンシかつ双方向・全二重通信が可能な軽量プロトコルです。WebSocket では最初に HTTP でハンドシェイク要求を送ってコネクションを張ったら、切断するまでそのコネクションを維持し、双方向・全二重でメッセージ交換を行います。メッセージ交換では HTTP ヘッダが無く、従来の HTTP 通信よりも通信コストを大幅に削減します。HTTP や HTTPS と同じ基盤（ポート番号含む）で動作するため、プロキシやファイアウォールに新たな設定を追加する必要がない点に意義があります。また Internet Explorer 10 を含む多くのブラウザが WebSocket に対応しているため、リアルタイムな情報交換システムを構築できる環境が揃っています。

Java API for WebSocket（以降、WebSocket API）は JSR 356 で策定されたものであり、WebSocket アプリケーションを構築し、クライアント・サーバ間で文字列データおよびバイナリデータを送受信できます。また WebSocket API 実装や依存ライブラリを用意すれば、Java SE 環境でも WebSocket クライアントを開発・実行できます。

現実では WebSocket クライアントを JavaScript で作成しブラウザ上で動作させることが多いですが、本書では WebSocket API について、クライアント・サーバ双方を Java で実装する方法を紹介します。なお、本書で示すサンプルは Java SE 7 の言語仕様に沿ってプログラミングしたものであり、読みやすくするために例外処理を省略しています。

API パッケージ構成

WebSocket API は、次のパッケージで構成しています。

- javax.websocket
- javax.websocket.server

エンドポイント

WebSocket API ではクライアント・サーバともに javax.websocket.Endpoint クラスから派生してエンドポイントのクラスを定義します。そして、onOpen()、onClose()および onError()をオーバーライドし、それぞれ接続時、切断時およびエラー発生時の処理を定義します。また javax.websocket.MessageHandler.Partial または javax.websocket.MessageHandler.Whole インタフェースの実装クラスを Session クラスの addMessageHandler()で登録すると、メッセージを受信できます。Whole インタフェースはメッセージを 1 回で受信するときに使い、Partial インタフェースは巨大なメッセージを何回かに分割して受信するときに使います。なお Endpoint クラスの onOpen()は abstract メソッドなので onOpen()のオーバーライドは必須ですが、onClose()および onError()は必要に応じてオーバーライドします。

このように Endpoint から派生して定義したエンドポイントのことを、WebSocket API 仕様では Programatic endpoint と呼んでいます。

クラス／メソッド	実装方法
エンドポイントのクラス	Endpoint から派生して定義
接続ハンドラ	onOpen()をオーバーライドして定義（必須）
切断ハンドラ	onClose()をオーバーライドして定義（任意）
エラーハンドラ	onError()をオーバーライドして定義（任意）
メッセージ受信ハンドラ	MessageHandler.Partial または MessageHandler.Whole の実装クラスを定義し、Session#addMessageHandler()で登録

表 3.1 Programatic endpoint の実装方法

Endpoint クラスから派生せずに、アノテーションを利用してもエンドポイントを定義できます。この場合、@ServerEndpoint を付けたクラスがサーバエンドポイントになり、@ClientEndpoint を付けたクラスがクライアントエンドポイントになります。両クラスともにメソッドに@OnOpen、@OnClose および @OnError を付加すれば、Programatic endpoint と同様、エンドポイントのライフサイクルで発生する各イベントのハンドラになります。また@OnMessage を付加したメソッドは、メッセージ受信ハンドラになります。

@OnOpen、@OnClose および@OnError は、エンドポイントあたり高々1 個まで付加できます。@OnMessage は、文字列メッセージ受信用、バイナリメッセージ受信用および、pong メッセージ（javax.websocket.PongMessage）受信用それぞれに高々1 個まで付加できます。たとえば、文字列メッセージ受信ハンドラとバイナリメッセージ受信ハンドラを 1 個ずつ定義できますが、文字列メッセージ受信ハンドラを 2 個以上定義するのは不可です。2 個以上定義すると、ランタイムエラーとなります（実行時に例外がスローされる）。

なお@PathParam はサーバエンドポイントでのみ利用可能であり、@ServerEndpoint に URI-Template を指定しておくと、接続時にクライアントから送られてくる URI から値を取得できます。たとえば、サーバエンドポイントの URI が「/chatEndpoint/{id}」のときに、クライアントから「/chatEndpoint/1000」を指定して接続してきた場合、@PathParam("id")で「1000」を取得できます。

WebSocket API の仕様では、アノテーションを付加して作成したエンドポイントのことを、Annotated endpoint と呼んでいます。

アノテーション	レベル	概要
@ServerEndpoint	クラス	クラスをサーバエンドポイントにする
@ClientEndpoint	クラス	クラスをクライアントエンドポイントにする
@OnOpen	メソッド	メソッドを接続ハンドラにする
@OnClose	メソッド	メソッドを切断ハンドラにする
@OnError	メソッド	メソッドをエラーハンドラにする
@OnMessage	メソッド	メソッドをメッセージ受信ハンドラにする
@PathParam	メソッド引数	URI-Templete から値を取得するために使用する

表 3.2 Annotated endpoint の実装方法

WebSocket API ではクライアントからサーバに接続するとユニークなセッションを生成し、クライアント・サーバ双方でエンドポイントのインスタンスも生成します。以降、切断するまでの間がセッションの寿命であり、セッションを介してメッセージを送受信します。エンドポイントのライフサイクルにおける処理の流れを図 3.1 に示します。

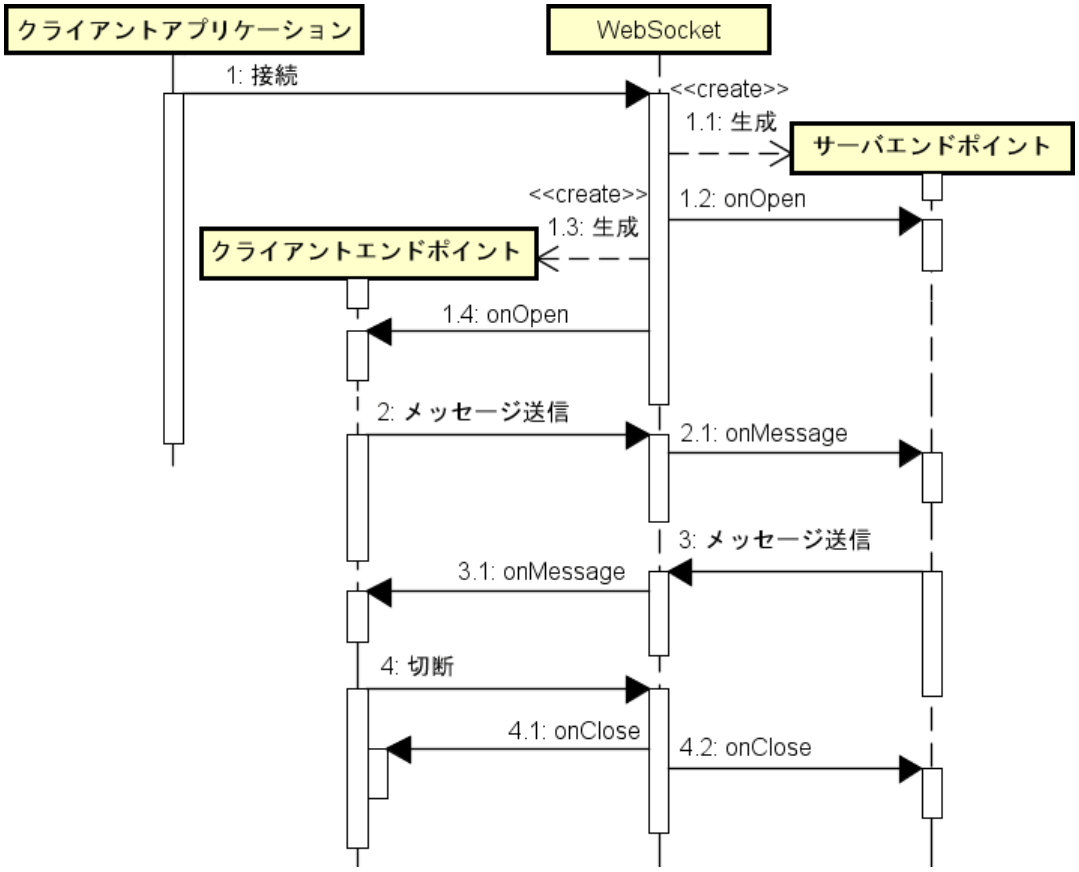

図 3.1 WebSocket ライフサイクルにおける処理の流れ

プレイン文字列の送受信

WebSocket アプリケーション作成の手始めとして、プレイン文字列（平坦な文字列）を送受信する WebSocket サンプルを紹介します。Programatic endpoint よりも Annotated Endpoint を作成するのが簡単です。サーバおよびクライアントの Annotated Endpoint を、それぞれ図 4.1 および図 4.2 に示します。

```java
@ServerEndpoint("/simple")
public class SimpleServer {
    @OnMessage
    public void onMessageText(String message, Session session) throws IOException {
        session.getBasicRemote().sendText(message); //文字列を送信
    }
}
```

図 4.1 プレイン文字列を送受信するサーバエンドポイント

```java
@ClientEndpoint
public class SimpleClient {
    @OnMessage
    public void onTextMessage(String msg, Session session) {
        System.out.println("onTextMessage: " + msg);
    }

    public static void main(String[] args) throws Exception {
        String url = "ws://localhost:8080/WebSocketSample/simple";
        WebSocketContainer container = ContainerProvider.getWebSocketContainer();
        Class<?> clazz = SimpleClient.class;
        try(Session session = container.connectToServer(clazz, URI.create(url))){ //サーバに接続
            session.getBasicRemote().sendText("Hello"); //文字列を送信
            Thread.sleep(3000L); //すぐ切断しないよう、暫定的に3秒待つ
        }
    }
}
```

図 4.2 プレイン文字列を送受信するクライアントエンドポイント

これだけのプログラムで文字列を送受信する WebSocket アプリケーションの完成です。

サーバではクラスに@ServerEndpoint を付加し、クライアントにメッセージをエコーバックするメソッドに@OnMessage を付加しています。クライアントではクラスに@ClientEndpoint を付加し、サーバからのメッセージをコンソールに出力するメソッドに@OnMessage を付加しています。クライアントの main()では、サーバ接続後にメッセージを送信し、try-with-resources 構文により暗黙的に Session クラスの close()を呼び出して接続を切断します。

サーバエンドポイントの URL が"ws://localhost:8080/WebSocketSample/simple"です。先頭の"ws"は RFC 6455 で定義している WebSocket のプロトコルであり、デフォルトのポート番号は 80 です。暗号化したセキュアな通信を行うための"wss"（ポート番号は 443）もあります。"/WebSocketSample"は Java EE コンテナに配備するアプリケーション名であり、最語尾の"/simple"は、@ServerEndpoint の属性で指定したものと同じです。

JSON の送受信

JSON の実体は文字列ですから、前節の方法でも JSON を送受信できます。しかし、開発者としては JSON を解析・生成する処理をエンドポイントから分離し、エンドポイントでは JSON を文字列ではなく Java オブジェクト（以降、カスタムオブジェクトと表記）として受け取りたいものです。WebSocket API には、このような開発者ニーズに対応する仕組みがあります。

具体的にはエンドポイントのクラスに加えて、表 5.1 のクラスを用意します。これらのクラスを、サー

バエンドポイントとクライアントエンドポイント共通のクラスとして作成し、双方のクラスパスに追加します。

クラス	概要
カスタムオブジェクト	JSON データを表現する POJO です。必要に応じて、JSON の各プロパティを取得/設定する getter/setter のメソッドを追加するとよいでしょう。このクラスを用意することにより、エンドポイントでは JSON を解析・生成する処理が不要になります。
デコーダ	javax.websocket.Decoder.Text インタフェースの実装クラスとして定義します。デコーダでは、デコード処理を行うかどうかを決定するための boolean 値を返す willDecode()および、JSON をカスタムオブジェクトに変換するための decode()を実装します。@ServerEndpoint および@ClientEndpoint の decoders 属性で、1 個以上のデコーダのクラスを指定することにより、WebSocket ランタイムがデコードの willDecode()および decode()を呼び出してくれます。 JSON が送信されてきたら、WebSocket ランタイムは、まず、デコーダの willDecode()を呼び出します。willDecode()が true を返した場合は、WebSocket ランタイムはそのデコーダの decode()を呼び出し、他のデコーダの willDecode()および decode()は呼び出しません。逆に、willDecode()が false を返した場合は、そのデコーダの decode()を呼び出さずに、次のデコーダの willDecode()を呼び出します。 なお、WebSocket API の仕様では、デコーダの呼び出し順序を明記していませんが、GlassFish 4 同梱の WebSocket API のリファレンス実装である Tyrus 1.0 の場合、decoders に指定した順に willDecode()および decode()を呼び出します。
エンコーダ	javax.websocket.Encoder.Text インタフェースの実装クラスとして定義し、カスタムオブジェクトを JSON に変換するための encode()を実装します。 @ServerEndpoint および@ClientEndpoint の encoders 属性で、1 個以上のエンコーダのクラスを指定できます。WebSocket ランタイムはエンコーダのクラス宣言で指定するジェネリクスを見て、どのエンコーダの encode()を呼び出すかどうかを決定します。

表 5.1 JSON を送受信するときに作成するクラスの一覧

JSON を交換するときの処理の流れを、図 5.1 に示します。エンドポイントが 2 つありますが、どちらか片方がクライアントエンドポイントで、もう一方はサーバエンドポイントです。その間を WebSocket ランタイムが仲介します。右側のエンドポイントが JSON を受信するときの流れが 1～1.3、JSON を送信するときの流れが 2～2.2 です。

まず 1～1.3 を説明します。左側のエンドポイントが String 型の JSON を送信したら(1)、デコーダの willDecode()を呼び出します(1.1)。willDecode()が true を返したら、そのデコーダの decode()を呼び出し、JSON をカスタムオブジェクトに変換します(1.2)。最後に、エンドポイントの@OnMessage メソッドを呼び出して、カスタムオブジェクトを渡すわけです(1.3)。

次に 2～2.2 を説明します。右側のエンドポイントが sendObject()でカスタムオブジェクトを送信したら(2)、エンコーダの encode()を呼び出し(2.1)、カスタムオブジェクトを String 型の JSON に変換します。最後に、左側のエンドポイントに JSON を送信します(2.2)。

このようにデコーダおよびエンコーダは、エンドポイントがメッセージを送受信するたびに、その間に入ってメッセージの変換を行うわけです。なお、デコーダとエンコーダでは、Java API for JSON Processing などを利用して JSON の生成および解析を行うのがよいでしょう。

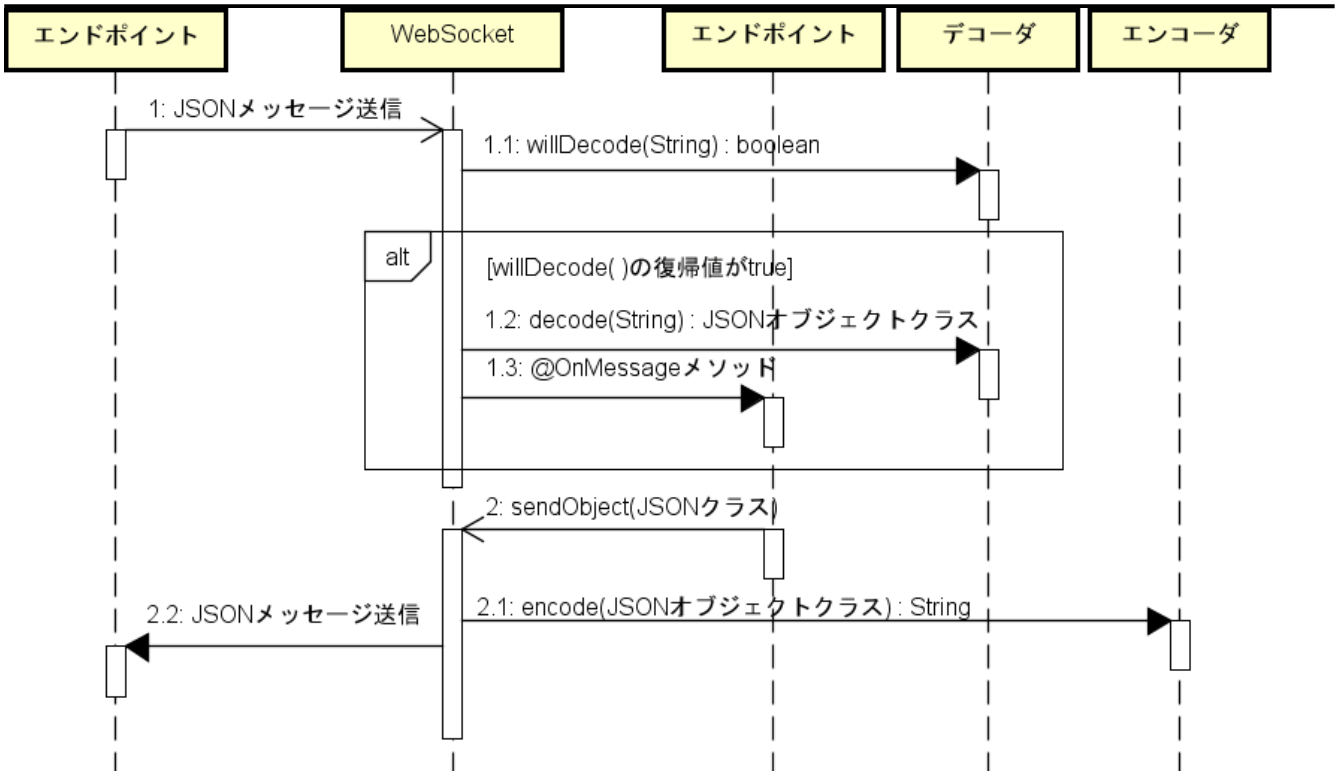

図 5.1 JSON 送受信時の処理の流れ

JSON を交換するチャットシステムを考えます。RFC 1459 などで策定された「Internet Relay Chat」は高度なチャットシステムであり、JOIN（チャネルへの参加）、PRIVMSG（チャネルでの発言）および PART（チャネルから脱退）などのコマンドがあります。

今回はチャネルがない簡単なチャットシステムにし、コマンドも PRIVMSG のみ使うことにします。チャットでやりとりする JSON は図 5.2 のとおり、command、parameter および name プロパティを持ちます。command は PRIVMSG のみとし、parameter は「発言内容」、name は「発言者名」とします。

```
{"command":"PRIVMSG", "parameter":"Hello.", "name":"Taro"}
```

図 5.2 チャットアプリケーションで送受信する JSON

この JSON に対応する「カスタムオブジェクト」「デコーダ」および「エンコーダ」（表 5.1 参照）をまず作成します。それぞれ図 5.3～図 5.5 に示します。

```
public class ChatMessage {
    private final long time;
    private final String command;
    private final String parameter;
    private final String name;

    public ChatMessage(String command, String parameter, String name){
        this.time = System.currentTimeMillis();
        this.command = command;
        this.parameter = parameter;
        this.name = name;
    }

    public long getTime(){
```

```
        return this.time;
    }

    public String getCommand(){
        return this.command;
    }

    public String getParameter(){
        return this.parameter;
    }

    public String getName(){
        return this.name;
    }
}
```

図 5.3 カスタムオブジェクトのプログラム

```
public class ChatMessageDecoder implements Decoder.Text<ChatMessage> {
    @Override public void init(EndpointConfig ec) { }
    @Override public void destroy() { }

    @Override
    public boolean willDecode(String s) {
        //Java API for JSON Processingを使ってJSONを解析
        //JSONに「command」、「parameter」および「name」があれば、デコード対象とする
        JsonObject j = Json.createReader(new StringReader(s)).readObject();
        return (j.containsKey("command") && j.containsKey("parameter") && j.containsKey("name"));
    }

    @Override
    public ChatMessage decode(String s) throws DecodeException {
        //Java API for JSON Processingを使ってJSONを解析
        JsonObject j = Json.createReader(new StringReader(s)).readObject();
        String command = j.getString("command");
        String parameter = j.getString("parameter");
        String name = j.containsKey("name") ? j.getString("name") : null;
        return new ChatMessage(command, parameter, name);
    }
}
```

図 5.4 デコーダのプログラム

```
public class ChatMessageEncoder implements Encoder.Text<ChatMessage> {
    @Override public void init(EndpointConfig ec) { }
    @Override public void destroy() { }

    @Override
    public String encode(ChatMessage chatMessage) throws EncodeException {
        //Java API for JSON Processingを使ってJSONを生成
        StringWriter w = new StringWriter();
        try(JsonGenerator g = Json.createGenerator(w)){
            g.writeStartObject()
                .write("command", chatMessage.getCommand())
                .write("parameter", chatMessage.getParameter())
                .write("name", chatMessage.getName())
            .writeEnd();
        }
        return w.toString();
    }
}
```

図 5.5 エンコーダのプログラム

デコーダおよびエンコーダでは、クラス宣言でカスタムオブジェクトのクラス名をジェネリクス<ChatMessage>で指定しています。これに伴い、デコーダの decode()の復帰値とエンコーダの encode()の引数が ChatMessaage になります。またデコーダおよびエンコーダでは Java API for JSON Processing を利用して、JSON の解析および生成を行っています。これらの 3 つのクラスを用意することにより、エンドポイントでは JSON の解析・生成処理が不要になります。

サーバの Annotated Endpoint を図 5.6 に示します。

```
@ServerEndpoint(value = "/chat",
    encoders = {ChatMessageEncoder.class},
    decoders = {ChatMessageDecoder.class}
)
public class ChatServerEndpoint{
    @OnError
    public void onError(Session session, Throwable cause){
        System.err.println("ChatServerEndpoint#onError: "+cause.getMessage());
    }

    @OnMessage
    public void onChatMessage(ChatMessage chatMessage, Session session) throws Exception{
        Set<Session> sessions = session.getOpenSessions();
        for(Session s : sessions){
//          if(session != s){ //このif文を有効にすると、送信してきたクライアントには配信しない
                s.getBasicRemote().sendObject(chatMessage); //クライアントにJSONを配信
//          }
        }
    }
}
```

図 5.6 カスタムオブジェクトを送受信するサーバエンドポイントのプログラム

@ServerEndpoint の decoders 属性および encoders 属性に、先のデコーダとエンコーダのクラスをそれぞれ指定します。これだけで JSON を送受信するたびに、ChatMessage オブジェクトとの相互変換を行ってくれます。また、onChatMessage()では、Session クラスの getOpenSessions()を利用して、接続中のすべてのクライアントのセッションを取得し、JSON を一斉配信しています。

クライアントの Annotated Endpoint を図 5.7 に示します。

```
@ClientEndpoint(
    encoders = {ChatMessageEncoder.class},
    decoders = {ChatMessageDecoder.class}
)
public class ChatClientEndpoint {
    @OnError
    public void onError(Session session, Throwable cause){
        System.err.println("ChatClientEndpoint#onError: "+cause.getMessage());
    }

    @OnMessage
    public void onChatMessage(ChatMessage chatMessage, Session session) {
        String output = String.format("%1$s: %2$tH:%2$tM (%3$s) %4$s",
            session.getUserProperties().get("name"),
            chatMessage.getTime(), chatMessage.getName(), chatMessage.getParameter());
        System.out.println(output);
    }
}
```

図 5.7 カスタムオブジェクトを送受信するクライアントエンドポイントのプログラム

@ServerEndpoint と同様、@ClientEndpoint の decoders 属性および encoders 属性にも、デコーダおよびエンコーダのクラスを指定します。

図 5.8 は、サーバエンドポイントに接続し、PRIVMSG を送信するクライアントプログラムです。「プレイン文字列の送受信」の図 4.2 の main()と内容が似ていますが、特に sendObject()を使ってカスタムオブジェクトを送信している点に注目してください。sendObject()を利用すれば、サーバエンドポイントに送信する前に、エンコーダがカスタムオブジェクトを JSON に変換してくれるわけです。

```
class ChatClient {
    public static void main(String... args) throws Exception {
        String url = "ws://localhost:8080/WebSocketSample/chat"; //サーバエンドポイントのURL
        WebSocketContainer container = ContainerProvider.getWebSocketContainer();
        Class<?> clazz = ChatClientEndpoint.class;
        try(Session session = container.connectToServer(clazz, URI.create(url))){ //接続
            //JSONを送信
            session.getBasicRemote().sendObject(new ChatMessage("PRIVMSG", "Hello.", "Taro"));
            //すぐ切断しないよう、暫定的に3秒待つ
            Thread.sleep(3000L);
        }
    }
}
```

図 5.8 チャットクライアントのプログラム

JSON とバイナリの送受信

先述したとおり、1 個のエンドポイントは、「文字列メッセージ受信ハンドラ」と「バイナリメッセージ受信ハンドラ」をそれぞれ 1 個まで持てます。そこで、図 6.1 と図 6.2 の onBinaryMessage()を、それぞれ図 5.6 と図 5.7 の各エンドポイントに追加すれば、バイナリの送受信もできます。

```
    @OnMessage
    public void onBinaryMessage(ByteBuffer data, Session session) throws IOException {
        Set<Session> sessions = session.getOpenSessions();
        for(Session s : sessions){
//          if(session != s){ //このif文を有効にすると、送信してきたクライアントには配信しない
                s.getBasicRemote().sendBinary(data); //クライアントにバイナリを配信
//          }
        }
    }
```

図 6.1 サーバエンドポイントのバイナリメッセージ受信ハンドラのプログラム

```
    @OnMessage
    public void onBinaryMessage(ByteBuffer data) throws IOException {
        byte[] array = data.array();
        //受信したバイナリをファイルに保存する
        try(BufferedOutputStream w = new BufferedOutputStream(new FileOutputStream("bin.dat"))){
            w.write(array, 0, array.length);
        }
    }
```

図 6.2 クライアントエンドポイントのバイナリメッセージ受信ハンドラのプログラム

図 6.3 は、sendBinary()を利用してバイナリを送信するプログラム例です。図 5.8 の main()に追加す

れば、サーバエンドポイントにバイナリメッセージを送信できます。

```
byte[] array = "abc".getBytes("UTF-8");
ByteBuffer buffer = ByteBuffer.allocate(array.length);
buffer.put(array);
session.getBasicRemote().sendBinary(buffer);
```

図 6.3 バイナリメッセージを送信するプログラム

図 6.1 と図 6.2 のプログラムではバイナリを変換せずにそのまま受信しますが、表 6.1 のデコーダおよびエンコーダを用意すれば、「JSON の送受信」と同様にエンドポイントでバイナリの代わりにカスタムオブジェクトを受信できます。

クラス	概要
カスタムオブジェクト	バイナリメッセージを表現する POJO です。必要に応じて、バイナリの各プロパティを取得/設定する getter/setter のメソッドを追加するとよいでしょう。このクラスを用意することにより、エンドポイントではバイナリを解析・生成する処理が不要になります。
デコーダ	javax.websocket.Decoder.Binary インタフェースの実装クラスとして定義します。デコーダでは、デコード処理を行うかどうかを決定するための boolean 値を返す willDecode()および、b バイナリをカスタムオブジェクトに変換するための decode()を実装します。@ServerEndpoint および@ClientEndpoint の decoders 属性で、1 個以上のデコーダのクラスを指定することにより、WebSocket ランタイムがデコードの willDecode()および decode()を呼び出してくれます。 バイナリが送信されてきたら、WebSocket ランタイムは、まず、デコーダの willDecode()を呼び出します。willDecode()が true を返した場合は、WebSocket ランタイムはそのデコーダの decode()を呼び出し、他のデコーダの willDecode()および decode()は呼び出しません。逆に、willDecode()が false を返した場合は、そのデコーダの decode()を呼び出さずに、次のデコーダの willDecode()を呼び出します。 なお、WebSocket API の仕様では、デコーダの呼び出し順序を明記していませんが、GlassFish 4 同梱の WebSocket API のリファレンス実装である Tyrus 1.0 の場合、decoders に指定した順に willDecode()および decode()を呼び出します。
エンコーダ	javax.websocket.Encoder.Binary インタフェースの実装クラスとして定義し、カスタムオブジェクトをバイナリに変換するための encode()を実装します。 @ServerEndpoint および@ClientEndpoint の encoders 属性で、1 個以上のエンコーダのクラスを指定できます。WebSocket ランタイムはエンコーダのクラス宣言で指定するジェネリクスを見て、どのエンコーダの encode()を呼び出すかどうかを決定します。

表 6.1 バイナリを送受信するときに作成するクラスの一覧

最後に

以上、WebSocket API について説明しましたが、GlassFish 4 同梱の WebSocket API のリファレンス実装である Tyrus 1.0 には多くのバグが報告されています。最新バージョンの Tyrus を入手して、開発・運用を行うのをお勧めします。

参考文献

- **RFC 6455**

  http://tools.ietf.org/html/rfc6455

  WebSocket プロトコルを策定しています。

- **JSR 356**

  http://jcp.org/en/jsr/detail?id=356

  Web Socket API を策定しています。

- **Java Platform, Enterprise Edition (Java EE) Technical Documentation**

  http://docs.oracle.com/javaee/

  Java EE 7 チュートリアルや Java EE 7 ドキュメントへのリンクがあります。
- **Tyrus**

  http://tyrus.java.net

  GlassFish 4 同梱の WebSocket API のリファレンス実装を開発しているプロジェクトです。

  WebSocket API の仕様書などへのリンクがあります。
- **Tyrus の Issue Tracking**

  https://java.net/jira/browse/TYRUS/

  Tyrus のバグ管理システムです。Tyrus のバグ情報がわかります。

商標について

- Java、Java HotSpot は、Oracle Corporation およびその子会社、関連会社の米国およびその他の国における登録商標です。
- その他の会社名および製品名は、それぞれの会社の登録商標もしくは商標です。

－以上－