



.NET SDK ガイド

Sybase Event Stream Processor

5.0

ドキュメント ID：DC01753-01-0500-01

改訂：2011 年 12 月

Copyright © 2011 by Sybase, Inc. All rights reserved.

このマニュアルは Sybase ソフトウェアの付属マニュアルであり、新しいマニュアルまたはテクニカル・ノートで特に示されないかぎり、後続のリリースにも付属します。このマニュアルの内容は予告なしに変更されることがあります。このマニュアルに記載されているソフトウェアはライセンス契約に基づいて提供されるものであり、無断で使用することはできません。

このマニュアルの内容を弊社の書面による事前許可を得ずに、電子的、機械的、手作業、光学的、またはその他のいかなる手段によっても、複製、転載、翻訳することを禁じます。

Sybase の商標は、Sybase の商標リスト (<http://www.sybase.com/detail?id=1011207>) で確認できます。Sybase およびこのリストに掲載されている商標は、米国法人 Sybase, Inc. の商標です。® は、米国における登録商標であることを示します。

このマニュアルに記載されている SAP、その他の SAP 製品、サービス、および関連するロゴは、ドイツおよびその他の国における SAP AG の商標または登録商標です。

Java および Java 関連の商標は、米国およびその他の国における Sun Microsystems, Inc. の商標または登録商標です。

Unicode と Unicode のロゴは、Unicode, Inc. の登録商標です。

このマニュアルに記載されている上記以外の社名および製品名は、当該各社の商標または登録商標の場合があります。

Use, duplication, or disclosure by the government is subject to the restrictions set forth in subparagraph (c)(1)(ii) of DFARS 52.227-7013 for the DOD and as set forth in FAR 52.227-19(a)-(d) for civilian agencies.

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

目次

Aleri Streaming Platform からの移行	1
エンティティのライフサイクルとアクセス・モード	3
SDK の起動	4
サーバへの接続	5
プロジェクトの取得と接続	5
パブリッシュ	7
直接アクセス・モードでのパブリッシュ	8
サブスクライブ	11
コールバック・モードでのストリームのサブスク ライブ	11
SDK の停止	15
フェールオーバーの処理	17
API リファレンス	19
索引	21

目次

Aleri Streaming Platform からの移行

Sybase® Event Stream Processor (ESP) の SDK インターフェイスは、Aleri Streaming Platform (ASP) の SDK インターフェイスとは異なります。Event Stream Processor の SDK が改善され、柔軟性およびパフォーマンスが向上し、クラスタ環境で実行しているプロジェクトに対応できるようになりました。

クラスタとプロジェクト

プロジェクトをクラスタで実行できるようになったため、プロジェクトにコマンドとコントロール・ホストおよびコントロール・ポートを使用してアクセスする必要がなくなりました。プロジェクトには、一意の ID が付けられており、その ID は、通常、クラスタ情報、ワークスペース名、プロジェクト名で構成される URI で示されます。SDK によって、内部的に URI が物理アドレスに解決されます。ESP のプロジェクト・オブジェクトは、ASP のプラットフォーム・オブジェクトにほぼ対応しています。Pub/Sub API は ESP サーバの API とは異なります。

注意： スタンドアロン・プロジェクトに接続する方法はありますが、今後のリリースで削除される予定の機能であるため、使用しないでください。

ESP SDK には、クラスタの構成および監視を行う新しい機能があります。ASP Pub/Sub API には、これらの機能に相当するものではありません。

アクセス・モード

ASP Pub/Sub では、プラットフォーム・オブジェクトとパブリッシャ・オブジェクトには、同期メソッド呼び出しを使用してアクセスしていました。サブスクライバ・オブジェクトには、コールバック・ハンドラが必要でしたが ESP では、この点が変更されました。すべてのエンティティ (サーバ、プロジェクト、パブリッシャ、サブスクライバ) には、直接メソッドを呼び出すか、コールバックハンドラを使用してアクセスできます。また、ESP では、セレクト・アクセスという 3 つ目のメソッドが導入されました。

直接アクセス・モードは、以前のプラットフォーム・オブジェクトとパブリッシャ・オブジェクトが ASP で呼び出される方法と似ています。各オブジェクトの呼び出しは、タスクが完了するか、エラーが発生するまでブロックします。ESP では、サブスクライバの操作でこのモードも使用できます。

コールバック・モードでは、ハンドラ関数を登録すると、SDK でユーザ指定のイベントが発生した場合に、指定の関数が呼び出されます。ASP では、これはサブスクライバを操作する唯一の方法でした。ESP では、他のエンティティにもこのメソッドを使用できます。

セレクト・アクセス・モードでは、セレクトに複数のエンティティを登録することができます。また、単一のスレッドが、それらのエンティティのいずれかに関するイベントを待つように設定できます。この機能は、単一のスレッドで複数のファイル記述子の監視を行う `select/poll` メカニズムに似ています。

自動再接続とモニタリング

ASP では、Pub/Sub API によってホット/アクティブ・モードで作業している場合のピアへの自動再接続がサポートされていました。ESP でも自動再接続がサポートされていますが、コールバック・アクセス・モードまたはセレクト・アクセス・モードの場合、その他の機能が利用できるようになりました。追加された機能には、クラスタまたはプロジェクトが停止したかどうかを確認する機能、オプションでバックエンドを監視して再起動する機能があります。

パブリッシュ

直接アクセス・モードでは、SDK でパブリッシュのスループットを向上するためにバックグラウンド・スレッドが早く進むように設定できるようになりました。ASP では、上記のようなタスクは、Pub/Sub ユーザが行う必要がありました。

ASP では、メッセージは一時的な記憶領域 (ベクトル) でフォーマットされており、その記憶域には、Pub/Sub API を呼び出してバッファを作成する前にデータを入力する必要がありました。ESP では、直接バッファにデータが書き込まれるため、この操作は必要ありません。ESP SDK でメッセージを作成するとき、ユーザがブロックまたはローの開始を指定し、その後順にデータが入力されます。フィールドはスキーマに表示されるのと同じ順序で入力する必要があります。

サブスクライブ

ASP では、メッセージのデータをオブジェクトのコレクションとして利用できました。ESP SDK では、この手順は利用できず、標準のデータ型またはヘルパ・オブジェクト (`Money`、`BigDatetime`、`Binary`) として直接バッファを読み取るメソッドを利用できます。データ・フィールドには、ランダムにアクセスできます。

エンティティのライフサイクルとアクセス・モード

Sybase® Event Stream Processor の .NET SDK は、C SDK と同じ機能を提供し、同じ概念を使用しています。SDK で公開されているすべてのエンティティで共通のライフサイクルが採用されています。

ユーザによる Event Stream Processor (ESP) SDK の操作は、SDK で使用されるエンティティを介して処理されます。主なエンティティには、サーバ、プロジェクト、パブリッシャ、サブスクリバがあります。これらのエンティティは SDK の機能に対応しています。たとえば、サーバ・オブジェクトは、クラスタの実行インスタンスを表します。プロジェクトは、クラスタに配備される単一のプロジェクトを、またパブリッシャ・オブジェクトは、実行プロジェクトへのデータのパブリッシュを扱います。

最初に取得するとき、エンティティはオープン状態であると見なされます。エンティティがオープン状態であるとき、そのエンティティに関する特定の静的情報を取得できます。割り当てられたタスクを行うには、エンティティをクラスタ内の該当のコンポーネントに接続する必要があります。サーバはクラスタの実行インスタンスに接続し、NetEspProject、NetEspPublisher、NetEspSubscriber はすべてクラスタ内のプロジェクトの実行インスタンスに接続します。

エンティティが接続されると、クラスタ・コンポーネントで操作できます。エンティティの接続が解除されると、クラスタで操作できなくなりますが、SDK 内ではアクティブなままで、クラスタに再度接続できます。エンティティをクローズすると、操作できなくなり、SDK で再度要求されます。クローズされたエンティティを再度利用するには、エンティティのコピーを新たに取得します。

たとえば、プロジェクト・オブジェクトを取得して、クラスタ内のプロジェクトに接続できます。バックエンドのプロジェクトが停止すると、SDK のプロジェクトは切断イベントを受け取ります。この場合、手動で再接続するか、コールバック・モードを使用していて、再接続がサポートされる構成の場合、SDK で自動の再接続が試行されます。再接続できた場合、接続イベントが生成されます。ユーザがエンティティをクローズすると、エンティティはバックエンドのプロジェクトから切断され、SDK でプロジェクト・オブジェクトが再度要求されます。再接続するには、新規のプロジェクト・オブジェクトを取得する必要があります。

SDK には、API を介して、さまざまなエンティティにアクセスできる柔軟な方法があります。エンティティには、直接モード、コールバック・モード、セレクト・モードでアクセスできます。デフォルトのアクセス・モードは、直接モードです。エンティティの取得時は常にこのモードに設定されています。直接モード

では、エンティティのすべての操作は同期されます。呼び出しが返されると、オペレーションが完了またはエラーが発生したと認識されます。

コールバック・アクセスでは、エンティティにハンドラ関数を登録します。エンティティに対するほとんどの呼び出しは、即座に返されます。リクエストが完了すると、該当のイベントが生成されます。SDK では、コールバック・メカニズムを実装するための内部スレッドが2つあります。更新スレッドは、現在登録されているすべてのエンティティを監視し、該当の更新のコールバックを確認します。更新がある場合、該当のイベントが生成され、ディスパッチ・スレッドのキューに追加されます。ディスパッチ・スレッドは、登録されているハンドラを呼び出し、ユーザ・コードによってそれら进行处理します。

セレクト・アクセス・モードでは、単一のユーザ・スレッドのさまざまなエンティティを多重化し、ファイル記述子を監視できます。この機能は、多くのシステムで使用されている select/poll メカニズムに似ています。エンティティを登録するには、監視するエンティティ (NetEspServer、NetEspPublisher、NetEspSubscriber、または NetEspProject) で `select_with(...)` メソッドを呼び出し、監視対象のイベントと一緒に NetEspSelector インスタンスに渡します。次に、NetEspSelector インスタンスで `select(...)` メソッドを呼び出すと、監視対象の更新がバックグラウンドで発生するまでブロックします。この関数では、NetEspEvent オブジェクトのリストが返されます。まずイベントのカテゴリ (サーバ、プロジェクト、パブリッシャ、サブスクライバ) を決定し、次に適切なイベント・タイプ进行处理します。このモードでは、SDK は1つのバックグラウンド更新スレッドを使用して更新を監視します。更新が検出されると、該当のイベントが作成され、NetEspSelector にプッシュされます。その後、イベントは各スレッドで処理されます。

SDK の起動

操作を実行する前に、SDK を起動します。

1. 次のようにエラー・メッセージ・ストアを作成します。

```
NetEspError error = new NetEspError();
```

2. .NET SDK のインスタンスを取得して、起動メソッドを呼び出します。

```
NetEspSdk s_sdk = NetEspSdk.get_sdk();  
s_sdk.start(espError);
```


サーバへの接続

SDK の起動時に、サーバに接続します。

前提条件

SDK を起動します。

手順

1. URI オブジェクトを作成します。

```
NetEspUri uri = new NetEspUri();
uri.set_uri("esp://myserver:19011", error);
```

2. クレデンシャルを作成します。クレデンシャルのタイプは、クラスタで設定されているセキュリティ・メソッドに応じて異なります。

```
NetEspCredentials creds = new
NetEspCredentials(NetEspCredentials.NET_ESP_CREDENTIALS_T.NET_ESP
_CREDENTIALS_SERVER_RSA);
creds.set_user("auser");
creds.set_password("1234");
creds.set_keyfile("../test_data/keys/client.pem");
```

3. オプションを設定します。

```
NetEspServerOptions options = new NetEspServerOptions();
options.set_mode(NetEspServerOptions.NET_ESP_ACCESS_MODE_T.NET_CA
LLBACK_ACCESS);
```

4. サーバに接続します。

```
server = new NetEspServer(uri, creds, options);
int rc = server.connect(error);
```

プロジェクトの取得と接続

データのパブリッシュまたはサブスクライブを行うには、プロジェクト・インスタンスを取得して接続します。

1. プロジェクトを取得します。

```
NetEspProject project = server.get_project("workspacename",
"projectname",
error);
```

2. プロジェクトに接続します。

```
project.connect(error);
```


パブリッシュ

SDK には、プロジェクトにデータをパブリッシュするためのさまざまなオプションがあります。

データをパブリッシュする手順は、次のとおりです。

1. 以前接続していた `NetEspProject` インスタンスから `NetEspPublisher` を作成します。
2. パブリッシュ先のストリームに対する `NetEspMessageWriter` を作成します。1 つの `NetEspPublisher` から複数の `NetEspMessageWriter` を作成できます。
3. `NetEspRelativeRowWriter` を作成します。
4. `NetEspRelativeRowWriter` メソッドを使用してパブリッシュするようにデータ・バッファをフォーマットします。
5. データをパブリッシュします。

`NetEspPublisher` はスレッドセーフですが、`NetEspMessageWriter` と `NetEspRelativeRowWriter` はスレッドセーフではありません。そのため、`NetEspMessageWriter` と `NetEspRelativeRowWriter` へのアクセスを同期するようにしてください。

SDK には、`NetEspPublisher` の動作を調整するさまざまなオプションがあります。`NetEspPublisher` を作成する場合に `NetEspPublisherOption` を使用してこれらのオプションを指定します。作成後は、オプションは変更できません。パブリッシュでは、SDK のその他のエンティティと同様に、直接、コールバック、セレクトの各アクセス・モードも使用できます。

SDK ではアクセス・モードを利用できるほか、内部バッファリング機能がサポートされます。パブリッシュがバッファに保存されると、データはまず内部キューに書き込まれます。このデータはパブリッシュ・スレッドで選択されて、プラットフォームに書き込まれます。バッファリング機能は、直接アクセス・モードの場合のみ利用できます。バッファに保存されたパブリッシュで、直接アクセス・モードを使用することにより、最も効率的なスループットが得られる可能性があります。

その他に、バッチ・モードと同期モードの設定がパブリッシュに影響を及ぼします。バッチ・モードは、データ・ローがソケットに書き込まれる方法を指定します。データ・ローは、個別に書き込むことも、エンベロープまたはトランザクション・バッチでグループ化することもできます。エンベロープは、個別のローをグループ化して、プラットフォームに書き込まれます。また、プラットフォームによってソケットから一括で読み取られます。これにより、ネットワーク・スループットが向上します。トランザクション・バッチは、エンベロープ・バッチと同様に、データをグループ化して、書き込みと読み取りが行われます。ただし、

トランザクション・バッチの場合、バッチ内のすべてのローの処理が完了した場合のみ、プラットフォームでグループが処理されます。ローの処理に失敗すると、バッチ全体がロールバックされます。

同期モードの設定では、SDK とプラットフォームの間のパブリッシュ・ハンドシェイクを制御できます。SDK は、デフォルトでは、データの受信確認を待たずにプラットフォームにデータを送信します。ただし、同期モードが `true` に設定されている場合は、SDK はプラットフォームからの受信確認を待ってから、データの次のバッチを送信します。この場合は、アプリケーション・レベルの配信は保証されますが、スループットが低下します。

同期モードでのパブリッシュの場合は、スループットは向上しますが、アプリケーション・レベルの配信は保証されません。TCP でもアプリケーション・レベルの配信は保証されないため、クライアントが終了すると、TCP バッファ内のデータは失われる可能性があります。そのため、同期モードでパブリッシュする場合は、クライアントが終了する前にコミットを実行します。

コールバック・モードまたはセレクト・モードでパブリッシュを実行する場合、注意すべき点があります。これらのモードは、`NET_ESP_PUBLISHER_EVENT_READY` イベントによって実行されます。このイベントは、パブリッシャで、より多くのデータを受信する準備ができていることを示します。これによって、ユーザはデータのパブリッシュまたはコミットの発行を行えますが、1つの `NET_ESP_PUBLISHER_EVENT_READY` イベントに対して実行できるアクションは1つのみです。

すべてのエンティティと同様に、コールバック・モードでパブリッシャを実行し、受信通知が必要な場合、イベントのトリガの前にコールバック・ハンドラを登録します。次に例を示します。

```
net_esp_publisher_options_set_access_mode(options, CALLBACK_ACCESS,
error);
net_esp_publisher_set_callback(publisher, events, callback, NULL,
error)
net_esp_publisher_connect(publisher, error);
```

直接アクセス・モードでのパブリッシュ

直接アクセス・モードでのパブリッシュは複数の手順から成るプロセスです。パブリッシャを作成して、接続してから、パブリッシュ先のストリームとパブリッシュ対象のデータを指定します。

次のコードは、データをパブリッシュする1つの方法です。具体的なパブリッシュ・シナリオに合わせて、必要に応じてこの例を変更してください。

1. パブリッシャを作成します。

```
NetEspPublisher publisher = project.create_publisher(null,  
error);
```

2. パブリッシャに接続します。

```
Publisher.connect(error);
```

3. ストリームを取得します。

```
NetEspStream stream = project.get_stream("WIN2", error);
```

4. メッセージ・ライタを取得します。

```
NetEspMessageWriter writer = publisher.get_message_writer(stream,  
error);
```

5. ロー・ライタを取得して開始し、1つのローを挿入する opcode を設定します。

```
NetEspRelativeRowWriter rowwriter =  
writer.get_relative_row_writer(error);  
rowwriter.start_row(error);  
rowwriter.set_opcode(1, error);
```

6. カラム値を1つ目のカラムから順に設定します。カラムのデータ型に適した set メソッドを呼び出します。たとえば、カラムのデータ型が文字列の場合は、次のように指定します。

```
rc = rowwriter.set_string("some value", error);
```

7. すべてのカラム値を設定し終わったら、ローを終了します。

```
rc = rowwriter.end_row(error);
```

8. データをパブリッシュします。

```
rc = publisher.publish(writer, error);
```

パブリッシュ

サブスクライブ

SDK には、プロジェクトをサブスクライブするためのさまざまなオプションがあります。

SDK を使用してデータをサブスクライブする手順は、次のとおりです。

1. 以前接続していた `NetEspProject` インスタンスから `NetEspSubscriber` を作成します。
2. ストリームをサブスクライブします。
3. 直接アクセス・モードでは、`NetEspSubscriber.get_next_event()` を使用してイベントを取得します。コールバック・アクセス・モードとセレクト・アクセス・モードでは、SDK によってイベントが作成されて、ユーザ・コードに戻されます。
4. データ・イベントの場合は、`NetEspMessageReader` を取得します。これにより、プラットフォームからの 1 つのメッセージがカプセル化されます。カプセルには、単一のデータ・ローカ、複数のデータ・ローカから成るトランザクション・ブロックまたはエンベロップ・ブロックが含まれている場合があります。
5. `NetEspRowReader` を 1 つ以上取得します `NetEspRowReader` のメソッドを使用して、個々のフィールドを読み取ります。

コールバック・モードでのストリームのサブスクライブ

コールバック・モードでのサブスクライブは複数の手順から成るプロセスです。サブスクライバとコールバック・レジストリを作成し、サブスクライバに接続してから、ストリームをサブスクライブします。

次のコードは、サブスクライブの 1 つの方法です。具体的なサブスクライブ・シナリオに合わせて、必要に応じてこのサンプルを変更してください。

1. サブスクライバを作成します。

```
NetEspSubscriberOptions options = new NetEspSubscriberOptions();
options.set_mode(NetEspSubscriberOptions.NET_ESP_ACCESS_MODE_T.NET_CALLBACK_ACCESS);
NetEspSubscriber subscriber =
project.create_subscriber(options,error);
```

2. コールバック・レジストリを作成します。

```
NetEspSubscriber.SUBSCRIBER_EVENT_CALLBACK callbackInstance = new
NetEspSubscriber.SUBSCRIBER_EVENT_CALLBACK(subscriber_callback);
subscriber.set_callback(NetEspSubscriber.NET_ESP_SUBSCRIBER_EVENT
.NET_ESP_SUBSCRIBER_EVENT_ALL, callbackInstance, null, error);
```

3. サブスクライバに接続します。

```
subscriber.connect(error);
```

4. ストリームをサブスクライブします。

```
subscriber.subscribe_stream(stream, error);
```

- コールバック関数の実装は、次のようになります。

```
Public static void subscriber_callback(NetEspSubscriberEvent
event, ValueType
data) {
    switch (evt.getType())
    {
        case (uint)
        (NetEspSubscriber.NET_ESP_SUBSCRIBER_EVENT.NET_ESP_SUBSCRIBER_
EVENT_CONNECTED):
            Console.WriteLine("the callback happened:
connected!");
            break;
        (uint)
        ( NetEspSubscriber.NET_ESP_SUBSCRIBER_EVENT.NET_ESP_SUBSCRIBER
_EVENT_DATA):
            //handleData
            ...
            break;

            default:
                break;
    }
}
} //end subscriber_callback
```

- handleData の実装は、次のようになります。

```
NetEspRowReader row_reader = null;
while ((row_reader = evt.getMessageReader().next_row(error)) !=
null) {
    for (int i = 0; i < schema.get_numcolumns(); ++i)
    {
        if ( row_reader.is_null(i) == 1) {
            Console.Write("null, ");
            continue;
        }
        switch
        (NetEspStream.getType(schema.get_column_type((uint)i, error)))
        {
            case
            NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_INTEGER:
                ivalue = row_reader.get_integer(i,
error);
                Console.Write(ivalue + ", ");
                break;
            case
            NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_LONG:
                lvalue = row_reader.get_long(i, error);
```



```
        Console.WriteLine(lvalue + ", ");
        break;
    case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_FLOAT:
        fvalue = row_reader.get_float(i,
error);
        Console.WriteLine(fvalue + ", ");
        break;
    case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_STRING:
        svalue = row_reader.get_string(i,
error);
        Console.WriteLine(svalue);
        break;
    case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_DATE:
        dvalue = row_reader.get_date(i, error);
        Console.WriteLine(dvalue + ", ");
        break;
    case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_TIMESTAMP:
        tvalue = row_reader.get_timestamp(i,
error);
        Console.WriteLine(tvalue + ", ");
        break;
    case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_BOOLEAN:
        boolvalue = row_reader.get_boolean(i,
error);
        Console.WriteLine(boolvalue + ", ");
        break;
    case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_BINARY:
        uint buffersize = 256;
        binvalue = row_reader.get_binary(i,
buffersize, error);
        Console.WriteLine(System.Text.Encoding.Default.GetString(binvalue)
+ ", ");
        break;
    case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_INTERVAL:
        intervalvalue = row_reader.get_interval(i,
error);
        Console.WriteLine(intervalvalue + ", ");
        break;
    case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_MONEY01:
        mon = row_reader.get_money(i, error);
        Console.WriteLine(mon.get_long(error) + ",
");
        break;
    case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_MONEY02:
        lvalue =
row_reader.get_money_as_long(i, error);
```

サブスクリイブ

```
        Console.WriteLine(lvalue + ", ");
        break;
    case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_MONEY03:
        mon = row_reader.get_money(i, error);
        Console.WriteLine(mon.get_long(error) + ",
");
        break;
    case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_MONEY10:
        mon = row_reader.get_money(i, error);
        Console.WriteLine(mon.get_long(error) + ",
");
        break;
    case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_MONEY15:
        mon = row_reader.get_money(i, error);
        Console.WriteLine(mon.get_long(error) + ",
");
        break;
    case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_BIGDATETIME:
        bdt2 = row_reader.get_bigdatetime(i,
error);
        long usecs =
bdt2.get_microseconds(error);
        Console.WriteLine(usecs + ", ");
        break;
    }
}

rc = subscriber.disconnect(error);
}
```

SDK の停止

操作が完了したら、.NET SDK を停止してリソースを解放します。

.NET SDK を停止するには、次のコマンドを使用します。

```
s_sdk.stop(espError);
```

SDK の停止

フェールオーバーの処理

SDK では、さまざまな状況で、完全に透過的なフェールオーバーまたは自動のフェールオーバーがサポートされます。

- **クラスタのフェールオーバー** – バックエンド・コンポーネントとの接続に使用される URI には、クラスタ・マネージャの指定内容のリストを含めることができます。SDK では、このリストへの接続が透過的に維持されます。クラスタ・マネージャのいずれか1つが停止した場合、SDK では別のインスタンスへの接続が試行されます。既知のインスタンスすべてへの接続に失敗した場合、エラーが返されます。コールバック・アクセス・モードまたはセレクト・アクセス・モードで実行している場合、SDK を構成して、接続が切断されるまでの許容範囲のレベルを1つ上げることができます。この場合、既知のすべてのマネージャ・インスタンスが停止しても、NetEspServer インスタンスは切断されず、NET_ESP_SERVER_EVENT_STALE イベントが生成されます。一定の回数(指定可能)の試行の後、再接続できた場合、NET_ESP_SERVER_EVENT_UPTODATE イベントが生成されます。それ以外の場合は、接続は切断され、NET_ESP_SERVER_EVENT_DISCONNECTED イベントが生成されます。
- **プロジェクトのフェールオーバー** – Event Stream Processor クラスタでは、プロジェクトにフェールオーバーを設定できます。構成設定により、クラスタでは、プロジェクトが終了したことが検出されると、プロジェクトが再開します(ただし、ユーザが明示的にプロジェクトを終了した場合は、プロジェクトは再開されません)。この機能に対応するため、NetEspProject インスタンスを使用してクラスタでプロジェクトの再開と再接続を監視できます。この機能は、コールバック・モードまたはセレクト・モードの場合にのみ使用できます。プロジェクトが停止したことを SDK が検出すると、NET_ESP_PROJECT_EVENT_STALE イベントが生成されます。再接続できた場合は、NET_ESP_PROJECT_EVENT_UPTODATE イベントが生成されます。それ以外の場合は、NET_ESP_PROJECT_EVENT_DISCONNECTED イベントが生成されます。
- **アクティブ/アクティブの配備** – プロジェクトをアクティブ/アクティブ・モードで配備できます。このモードでは、クラスタは1つのプロジェクトで、プライマリ・インスタンスとセカンダリ・インスタンスの2つのインスタンスを開始できます。プライマリ・インスタンスにパブリッシュされたデータは、セカンダリ・インスタンスに自動的にミラーリングされます。SDK では、アクティブ/アクティブの配備がサポートされます。アクティブ/アクティブの配備に接続している場合に、現在接続しているインスタンスが停止すると、NetEspProject はもう1つのインスタンスへの再接続を試みます。フェールオーバーとは異なり、これは透過的に行われます。そのため、再接続できた場合で

も、それは明示的に示されません。NetEspProject のほかに、パブリッシュとサブスクライブでもこのモードはサポートされています。アクティブ/アクティブで配備されたプロジェクトにサブスクライブしている場合は、インスタンスが停止しても SDK はサブスクリプションを切断せず、

NET_ESP_SUBSCRIBER_EVENT_DATA_LOST イベントが生成されます。その後、ピア・インスタンスへの再接続が試行されます。再接続できた場合、同じストリームを再度サブスクライブします。次に、サブスクライブ・クライアントは、NET_ESP_SUBSCRIBER_EVENT_SYNC_START イベントに続けて、データ・イベントを受け取り、最後に

NET_ESP_SUBSCRIBER_EVENT_SYNC_END イベントを受け取ります。クライアントはこのシーケンスを使用して、必要に応じて、データ面での一貫性を維持できます。パブリッシュ中に再接続もできますが、この機能は同期モードでのパブリッシュでのみ利用できます。同期モード以外の場合、SDK では、データの一貫性を保証できません。パブリッシュ中の再接続は、透過的に行われ、外部のユーザ・イベントは生成されません。

API リファレンス

メソッド、関数、その他のプログラミング・ビルディング・ブロックの詳細については、API のマニュアルをダウンロードして参照してください。

.NET API のマニュアルをダウンロードして、ローカル・マシンにインストールしてください。

ダウンロードが完了したら、次の手順に従ってください。

1. ローカル・マシンのロケーションを指定し、そこにファイルを抽出します。
2. ファイルを抽出したロケーションを参照します。
3. `index.html` を開いて、API マニュアルを参照します。

索引

S

SDK

- 起動 4
- 停止 15

U

URI

- 作成 5

あ

- アクセス・モード
 - コールバック 3
 - セレクト 3
 - 直接 3

く

- クラスの詳細 19

こ

- コールバック・モード
 - 例 11

さ

- サーバ
 - 接続 5
- サブスクライブ
 - コールバック・モード 11
 - ストリーム 11
 - 概要 11
 - 例 11

は

- パブリッシュ
 - スループットの向上 7
 - プロジェクト 7
 - モード 7
 - 直接アクセス・モード 8
 - 例 8
- パブリッシュのモード
 - バッチ・モード 7
 - 同期モード 7

ふ

- フェールオーバー
 - アクティブ/アクティブ 17
 - クラスタ 17
 - プロジェクト 17
- フォールト・トレランス 17
- プロジェクト
 - パブリッシュ 7
 - 接続 5

め

- メソッドの詳細 19

り

- リファレンス
 - クラス 19
 - メソッド 19
 - 関数 19

