SYBASE®
An SAP® Company

Java SDK ガイド

Sybase Event Stream Processor 5.0

ドキュメント ID： DC01752-01-0500-01
改訂： 2011 年 12 月

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

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、セレクタ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接続がサポートされていました。ESP でも自動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 Java SDK では、すべてのエンティティに共通のライフサイクルがあり、共通のアクセス・モードを使用できます。

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、サブスクライバがあります。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。

データを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認識されます。 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、プロジェクト、パブリッシャ、サブスクライバ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を処理されます。

次の例はコールバック・モードでプロジェクトにアクセスする場合を示します。コールバック・モードを使用していて、コールバック・イベントを受け取る必要がある場合、対象のエンティティの接続を呼び出す前に、コールバック・ハンドラを登録します。

```
    ProjectOptions opts = new
ProjectOptions.Builder().setAccessMode(AccessMode.CALLBACK).create(
);

   Project project = SDK.getInstance().getProject(projectUri, creds,
opts);



    project.setCallback(EnumSet.allOf(ProjectEvent.Type.class), new
ProjectHandler("Handler"));



    project.connect(60000);



    //

    // Wait or block.
 Rest of the project lifecycle is handled in the project callback
handler

    //




    // Project handler class

    public class ProjectHandler implements Callback

    {

        String m_name;
```

```
        ProjectHandler(String name) {
            m_name = name;
        }

        public String getName() {
            return m_name;
        }

        public void processEvent(ProjectEvent pevent)
        {
            Project p = pevent.getProject();
            try {
                switch ( pevent.getType() ) {
                // Project has connected - can retrieve streams,
deployment etc.
                case CONNECTED:
                    String[] streams =
pevent.getProject().getModelledStreamNames();
                    break;
               // Project disconnected - only call possible connect
again
                case DISCONNECTED:
                    break;
              // Project closed - this object should not be accessed
anymore by user code
                case CLOSED:
                    break;
                case STALE:
                case UPTODATE:
                    break;
```

```
                case ERROR:
                    break;
                }
            } catch (IOException e) {
            }
        }
    }
```

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。セレクタで監視対象のイベントの情報とエンティティを登録します。 次に、**Selector.select()** 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、SdkEvent 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ー (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・タイプを処理します。 このモードでは、単一のバックグラウンド更新スレッドを使用して、更新の監視を行います。 更新が検出されると、該当のイベントが作成され、セレクタにプッシュされます。 その後、そのイベントは各スレッドで処理されます。

次に、異なるエンティティの多重化の例を示します。

```
        Uri cUri = new Uri.Builder(REAL_CLUSTER_URI).create();



        Selector selector = SDK.getInstance().getDefaultSelector();



        ServerOptions srvopts = new
ServerOptions.Builder().setAccessMode(AccessMode.SELECT).create();

        Server server = SDK.getInstance().getServer(cUri, creds,
srvopts);



        ProjectOptions prjopts = new
ProjectOptions.Builder().setAccessMode(AccessMode.SELECT).create();

       Project project = null; //SDK.getInstance().getProject(cUri,
creds, prjopts);
```

```
        SubscriberOptions subopts = new
SubscriberOptions.Builder().setAccessMode(AccessMode.SELECT).create
();

        Subscriber subscriber = null; //
SDK.getInstance().getProject(cUri, creds, prjopts);



        PublisherOptions pubopts = new
PublisherOptions.Builder().setAccessMode(AccessMode.SELECT).create(
);

        Publisher publisher = null; //
SDK.getInstance().getProject(cUri, creds, prjopts);



        server.connect();



        server.selectWith(selector,
EnumSet.allOf(ServerEvent.Type.class));



        // Your logic to exit the loop goes here ...

        while (true) {



            List<sdkevent> events = selector.select();



            for (SdkEvent event : events) {

                switch (event.getCategory()) {

                // Server events

                case SERVER:

                    ServerEvent srvevent = (ServerEvent) event;

                    switch (srvevent.getType()) {



                    // Server has connected - can now perform
operations, such as adding removing
```

```
                    // applications.
                    case CONNECTED:
                    case MANAGER_LIST_CHANGE:
                        Manager[] managers =
srvevent.getServer().getManagers();
                        for (Manager m : managers)
                            System.out.println("Manager:" + m);
                        break;
                    case CONTROLLER_LIST_CHANGE:
                        Controller[] controllers =
srvevent.getServer().getControllers();
                        for (Controller cn : controllers)
                            System.out.println("Controller:" + cn);
                        break;
                    case WORKSPACE_LIST_CHANGE:
                        break;


                  // This indicates that the Server has updated its
state with the latest running application
                    // information.
 Project objects can now be retrieved
                    case APPLICATION_LIST_CHANGE:
                    case DISCONNECTED:
                    case CLOSED:
                    case ERROR:
                        break;
                    }
                    break;


                // Project events
```

```
                case ESP_PROJECT:
                    ProjectEvent prjevent = (ProjectEvent) event;
                    switch (prjevent.getType()) {
                    case CONNECTED:
                    case DISCONNECTED:
                    case CLOSED:
                    case ERROR:
                    case WARNING:
                        break;
                    }
                    break;

                // Publisher events
                case PUBLISHER:
                   PublisherEvent pubevent = (PublisherEvent) event;
                    switch (pubevent.getType()) {
                    case CONNECTED:
                    // The publisher is read.
This event is to be used to publish data in callback mode
                    case READY:
                    case DISCONNECTED:
                    case CLOSED:
                        break;
                    }
                    break;

                // Subscriber events
                case SUBSCRIBER:
                 SubscriberEvent subevent = (SubscriberEvent) event;
```

```
                    switch (subevent.getType()) {
                    case CONNECTED:
                    case SUBSCRIBED:
                    case SYNC_START:
                    // There is data.
This event is to be used to retrieve the subscribed data.
                    case DATA:
                    case SYNC_END:
                    case DISCONNECTED:
                    case CLOSED:
                    case UNSUBSCRIBED:
                    case DATA_INVALID:
                    case ERROR:
                    case STREAM_EXIT:
                    case DATA_LOST:
                        break;
                    }
                    break;
                }
            }
        }
```

パブリッシュ

SDK では、データをプロジェクトにパブリッシュするさまざまなオプションを利用できます。

データのパブリッシュの手順は以下のとおりです。

1. パブリッシュする必要がある対象のプロジェクトのパブリッシャを取得する。 パブリッシャは、直接作成するか、以前に取得または接続したプロジェクトのオブジェクトから作成できます。
2. パブリッシュする対象のストリームの MessageWriter を作成する。 単一のパブリッシャから複数の MessageWriter を作成できます。
3. RelativeRowWriter メソッドを作成する。
4. データ・バッファをフォーマットし、RelativeRowWriter メソッドを使用してパブリッシュする。
5. データをパブリッシュする。

パブリッシャは、スレッドに対応してますが、MessageWriter と RelativeRowWriter は対応していません。 そのため、MessageWriter と RelativeRowWriter へのアクセスを同期するようにしてください。

SDK では、高可用性 (HA) 構成において、非同期でのパブリッシュを除き、自動のパブリッシャ・スイッチオーバがサポートされます。SDK では、パブリッシュ済みのレコードが認識されないため、このインスタンスで自動スイッチオーバを実行すると、レコードの削除や重複が起こる可能性があります。

SDK には、パブリッシャの動作を調整するさまざまなオプションがあります。 パブリッシャを作成するときに、PublisherOptions オブジェクトを使用して、これらのオプションを指定します。 パブリッシャの作成後は、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、直接モード、コールバック・モード、およびセレクト・アクセス・モードを使用できます。

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・スレッドに入力され、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、個別に書き込むか、エンベロープまたはトランザクショ

ン・バッチでクループ化して書き込めます。 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。

注意： シャインスルーを使用して、更新レコード内のデータが NULL である場合に、以前のデータを保持するようにするには、トランザクション・バッチを使用せず、個別にまたはエンベロープを使用してローをパブリッシュします。

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、データの受信確認なしでプラットフォームにデータを送信し続けます。 ただし、同期モードが設定されている場合、データ送信時は毎回プラットフォームからの受信確認を待ってから、次のデータが送信されます。 非同期モードでパブリッシュするときは、クライアント・アプリケーションを終了する前か、パブリッシャを終了する前に、**Publisher.commit()** コールを明示的に発行してください。 これは、書き込まれたデータのすべてがプラットフォームから必ず読み取られるようにするために行います。

一般的には、パブリッシュ・コールからのリターン・コードは、ローの送信が成功したかどうかを示します。プラットフォームでの処理中に発生したエラー (重複データの挿入など) は返されません。 パブリッシュ・コールからのリターン・コードの意味は、アクセス・モードと、同期転送または非同期転送のどちらを選択したかによって異なります。

コールバック・アクセス・モードまたはセレクト・アクセス・モードを使用する場合、リターン・コードでは、SDK でデータ・キューを作成できるかどうかのみが示されます。データがソケットに実際に書き込まれたかどうかは、該当のイベントによって示されます。コールバック・アクセス・モードおよびセレクト・アクセス・モードでは、現在同期パブリッシュは実行できません。

直接アクセス・モードを使用している場合、使用されている転送の種類によってパブリッシュ・コールからのリターン・コードの内容が決定します。パブリッシュが非同期モードで実行されている場合、リターン・コードでは、データがソケットに書き込まれたことのみが示されます。 パブリッシュが同期モードで実行されている場合、パブリッシュ・コールからのリターン・コードでは、プラットフォームから送信された応答コードが示されます。

パブリッシュ・コールでは、プラットフォームでの処理中に発生したエラー (重複データの挿入など) は返されません。

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、PublisherEvent.READY イベントによって実行されます。PublisherEvent.READY 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 PublisherEvent.READY イベントに対しては、1 つのアクションのみ行えます。

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。

```
PublisherOptions opts = new
PublisherOptions.Builder().setAccessMode(AccessMode.CALLBACK).creat
e();
        Publisher publisher = project.createPublisher(opts);
        PublisherHandler handler = new PublisherHandler();

publisher.setCallback(EnumSet.allOf(PublisherEvent.Type.class),
handler);
        publisher.connect();
```

下記のコード・スニペットは、データをパブリッシュする別の方法を示します。

この例では、直接アクセス・モードでトランザクション・ブロックを使用してパブリッシュする場合を示します。

```
    // The Project must be connected first

    project.connect(60000);



    Publisher publisher = project.createPublisher();

    publisher.connect();



    Stream stream = project.getStream("Stream");

    MessageWriter mw = publisher.getMessageWriter(s);

    RelativeRowWriter writer = mw.getRelativeRowWriter();

    // It is more efficient to cache this

    DataType[] types = stream.getEffectiveSchema().getColumnTypes();



    // Your logic to loop over data to publish

    while (true) {
```

```
// Logic to determine if to start a transaction block
if ( ...)
    mw.startTransaction(0);

// Loop over rows in a single block
while (true) {

    // Loop over columns in a row
    for (i = ...) {

        writer.startRow();
        writer.setOperation(Operation.INSERT);

        switch (types[i]) {
        case DATE:
            writer.setDate(datevalue);
            break;
        case DOUBLE:
            writer.setDouble(doublevalue);
            break;
        case INTEGER:
            writer.setInteger(intvalue);
            break;
        case LONG:
            writer.setLong(longvalue);
            break;
        case MONEY:
```

```
                    break;
                case STRING:
                    writer.setString(stringvalue);
                    break;
                case TIMESTAMP:
                    writer.setTimestamp(tsvalue);
                    break;
                //
                // Other data types
                //
                }
            }
            writer.endRow();
        }

        // Logic to determine if to end block
        if ( ...)
            mw.endBlock();
    }

    publisher.commit();
```

この例では、コールバック・アクセス・モードでパブリッシュする場合を示します。パブリッシャの接続前にアクセス・モードが設定されています。この設定は、コールバック・アクセス・モードおよびセレクト・アクセス・モードを使用する場合に必要です。

```
    p.connect(60000);


    PublisherOptions opts = new PublisherOptions.Builder()
        .setAccessMode(AccessMode.CALLBACK)
```

```
        .create();


    Publisher pub = p.createPublisher(opts);


    PublisherHandler handler = new PublisherHandler();
    pub.setCallback(EnumSet.allOf(PublisherEvent.Type.class),
handler);
    pub.connect();


    // Block/wait. Publishing happens in the callback handler
    // ....


    //
    // Publisher callback handler
    //
    public class PublisherHandler implements Callback
    {
        Stream m_stream;
        MessageWriter m_mwriter;
        RelativeRowWriter m_rowwriter;


        public PublisherHandler() throws IOException {
        }


        public String getName() {
            return "PublishHandler";
        }


        public void processEvent(PublisherEvent event)
```

```
        {
            switch (event.getType()) {
            case CONNECTED:
                // It is advisable to create and cache these
                try {
                    m_stream =
event.getPublisher().getProject.getStream("Stream");
                    m_mwriter =
event.getPublisher().getMessageWriter(mstr);
                    m_rowwriter = mwriter.getRelativeRowWriter();
                } catch (IOException e) {
                    e.printStackTrace();
                }
                break;

            case READY:
                // Publishing code goes here.
              // NOTE: Only a single publish or a commit call can be
made in one READY callback

                break;

            case ERROR:
            case DISCONNECTED:
            case CLOSED:
                break;
            }
        }
    }
```

サブスクライブ

SDK には、プロジェクトのデータをサブスクライブするさまざまなオプションがあります。

SDK でデータをサブスクライブするには、以下の手順に従ってください。

1. サブスクライバ・オブジェクトを作成する。 オブジェクトは直接作成するか、プロジェクト・オブジェクトから取得します。
2. サブスクライバ・オブジェクトを設定する。
3. 対象のストリームにサブスクライブする。
4. 直接アクセス・モードで Subscriber.getNextEvent() オブジェクトを使用して、イベントを取得する。 コールバック・アクセス・モードおよびセレクト・アクセス・モードでは、イベントが生成され、ユーザ・コードに渡されます。
5. データ・イベントに使用するため、MessageReader を取得する。 これにより、プラットフォームから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・ローか、複数のデータ・ローから成るトランザクション・ブロックまたはエンベロープ・ブロックが含まれている場合があります。
6. 1 つまたは複数の RowReaders を取得する。 RowReader のメソッドを使用して、各フィールドのデータを読み取ります。

この例では、デフォルトのオプションで直接アクセス・モードのストリームにサブスクライブする場合を示します。

```
p.connect(60000);

subscriber = p.createSubscriber();

String strName = "WIN1";


subscriber.subscribeStream("WIN1");

subscriber.connect();


// Various data type we will be reading

BigDatetime bigdatetime = null;

Money m = null;

byte[] binary = null;
```

```
        // Logic to exit loop goes here
        while (true) {
            SubscriberEvent event = subscriber.getNextEvent();
            switch (event.getType()) {
            case SYNC_START:
                break;
            case SYNC_END:
                break;

            // There is data to read
            case DATA:
                while ( reader.hasNextRow() ) {
                    RowReader rr = reader.nextRowReader();
                 for (int j = 0; j < rr.getSchema().getColumnCount();
++j) {
                       if ( rr.isNull(j))
                           continue;

                     // This is legal but it is better to cache the
data types array

                     switch ( rr.getSchema().getColumnTypes()[j]) {
                       case INTEGER:
                           rr.getInteger(j);
                           break;
                       case LONG:
                           rr.getLong(j);
                           break;
                       case STRING:
```

```
                        rr.getString(j);
                        break;
                    case TIMESTAMP:
                        rr.getTimestamp(j));
                        break;
                    case MONEY01:
                        m = rr.getMoney(j);
                        break;

                    // ...
                    // process other data types
                    // ...

                    }
                }
            }
            break;
        }

    }
    subscriber.disconnect();
```

フェールオーバの処理

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または自動のフェールオーバがサポートされます。

- **クラスタ・フェールオーバ** – 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、サーバ・インスタンスは切断されず、ServerEvent.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、ServerEvent.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、ServerEvent.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** – Event Stream Processor クラスタでは、プロジェクトにフェールオーバを設定できます。 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)。 この機能を使用するには、プロジェクト・インスタンスがクラスタでプロジェクトが再開されるのを監視し、再接続するように設定します。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。 SDK でプロジェクトの停止が検出された場合、ProjectEvent.STALE イベントが生成されます。 再接続できる場合、ProjectEvent.UPTODATE イベントが生成され、それ以外の場合は、ProjectEvent.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・モードで配備することができます。 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すべてのデータは、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／アクティブで配備されていると、プロジェクトに接続しているインスタンスが停止した場合に、別のインスタンスへの接続が試行されます。フェールオーバとは異なり、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。 プロジェクトで使用できるほ

か、パブリッシュとサブスクライブでこのモードを使用できます。 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すると、インスタンスが停止した場合でも SDK ではサブスクリプションは停止されず、SubscriberEvent.DATA_LOST イベントが生成されます。 その後 SDK では、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。 その後サブスクリプション・クライアントは、SubscriberEvent.SYNC_START イベントを受信し、その後データ・イベント、SubscriberEvent.SYNC_END イベントの順に受信します。 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。

Java 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

A

J

あ

さ

は

ふ