CHAPTER 5

SCE Subscriber API を使用したプログラミング

この章では、API プログラミングの構造、クラス、メソッド、およびインターフェイスについて詳細に説明します。

API クラス

次のリストに、API が提供するクラスのマッピングを示します。

- パッケージ com.scms.api.sce.prpc（p.5-2）
- パッケージ com.scms.api.sce（p.5-2）
- パッケージ com.scms.common（p.5-2）

パッケージ com.scms.api.sce.prpc

PRPC_SCESubscriberApi クラス（p.5-4）— 主要な API クラス

パッケージ com.scms.api.sce

- 通知リスナ（p.5-2）
- 接続モニタリング（p.5-2）
- SCE カスケード トポロジのサポート（p.5-2）
- 操作結果の処理（p.5-2）

通知リスナ

- LoginPullListener インターフェイス クラス（p.5-8）
- LogoutListener インターフェイス クラス（p.5-10）
- QuotaListenerEx インターフェイス クラス（p.5-10）

接続モニタリング

- ConnectionListener インターフェイス（p.5-14）

SCE カスケード トポロジのサポート

- RedundancyStateListener インターフェイス（p.5-15）

操作結果の処理

- OperationException クラス（p.5-19）
- SCESubscriberApi（インターフェイス）— OperationException 内部で受信できるエラー コード定数を含みます。
- OperationArguments クラス（p.5-18）
- OperationResultHandler インターフェイス（p.5-17）

パッケージ com.scms.common

com.scms.common パッケージには、API で使用されるすべてのデータ型が含まれています。

- Login_BULK クラス（p.4-9）
- LoginPullResponse_BULK クラス（p.4-12）
- NetworkAndSubscriberID_BULK クラス（p.4-11）
- PolicyProfile_BULK クラス（p.4-14）

プログラミングに関する注意事項

コールバック メソッドを使用したプログラミング

ここまで説明したように、API 操作の多くはコールバック メソッドに基づいています。ユーザは、特定のイベントが発生すると呼び出される「リスナ」を指定します。ここでは、コールバック メソッドを使用したプログラミングに関する主な注意事項を定義します。

コールバック メソッドのスレッド内では長い操作を実行しないでください。長い操作は**別のスレッド**から実行する必要があります。この推奨事項に従わないと、クライアント側でリソースが不足することがあります。

この注意事項は次の操作に適用されます。

- LoginPullListener コールバック メソッド
- LogoutListener コールバック メソッド
- QuotaListenerEx コールバック メソッド
- ConnectionListener コールバック メソッド

PRPC_SCESubscriberApi クラス

PRPC_SCESubscriberAPI クラス（com.scms.sce.api.prpc パッケージ上）は、次の機能を提供する主要な API クラスです。

- API の構築
- ただ 1 つの SCE と API の接続（接続属性の設定）
- 通知リスナの登録 / 登録解除
- 接続リスナの設定
- サブスクライバ プロビジョニング操作の実行
- SCE からの切断

API の構築

PRPC_SCESubscriberAPI には次のコンストラクタがあります。

構文：

```
public PRPC_SCESubscriberApi(String apiName, String sceHost)
throws UnknownHostException
public PRPC_SCESubscriberApi(String apiName,
String sceHost,
long autoReconnectInterval)
throws UnknownHostException
public PRPC_SCESubscriberApi(String apiName,
String sceHost,
int scePort,
long autoReconnectInterval)
throws UnknownHostException
```

パラメータ：

次に、API コンストラクタのコンストラクタ引数の説明を示します。

apiName — API 名を指定します。

（注）　API 名は SCE ごとに一意である必要があります。同じ名前を持つ API を複数構築して、単一の SCE に接続した場合、その SCE プラットフォームは複数の API を 1 つの API クライアントとして処理します。この機能は、ハイ アベイラビリティがサポートされている場合にのみ使用してください。ハイ アベイラビリティの詳細については、「ハイ アベイラビリティの実装」（p.5-39）を参照してください。

sceHost — IP アドレスまたは到達可能なホスト名のいずれかになります。

scePort — SCE に接続する PRPC プロトコル TCP ポート（デフォルト値は 14374）

autoReconnectInterval — 再接続タスクで再接続を試行するインターバル（ミリ秒単位）を、次のように定義します。

- 値が 0 以下の場合、再接続タスクはアクティブになりません（自動再接続は試行されません）。
- 値が 0 より大きくて、接続障害が発生している場合、再接続タスクは <autoReconnectInterval> ミリ秒ごとにアクティブになります。
- デフォルト値：–1（自動再接続は試行されません。）

（注）　自動再接続サポートをイネーブルにするには、API の **connect** メソッドを 1 回以上呼び出す**必要があります。**

例

次のコードは、10 秒間隔で自動再接続を行って、API を構築します。

```
PRPC_SCESuscriberAPI sceApi = new PRPC_SCESuscriberAPI("MyApi",
"10.1.1.1",
10000);
sceApi.connect();
```

次のコードは、自動再接続をサポートしないで、API を構築します。

```
PRPC_SCESuscriberAPI sceApi = new PRPC_SCESuscriberAPI("MyApi",
"10.1.1.1");
sceApi.connect();
```

リスナ セットアップ操作

API を初期化したら、API を使用するアプリケーションのタイプ、および使用するトポロジに基づいて、API に利用対象リスナを設定する必要があります。トポロジの詳細については、「サポート対象トポロジ」（p.2-6）を参照してください。

リスナ セットアップ操作の内容は、次のとおりです。

- 接続リスナの設定（詳細は「接続モニタリング」[p.5-14] を参照）
- `public void setConnectionListener(ConnectionListener listener)`
- ログインプル リスナの設定（詳細は「LoginPullListener インターフェイス クラス」[p.5-8] を参照）
- `public void registerLoginPullListener(LoginPullListener listener)`
- ログアウト リスナの設定（詳細は「LogoutListener インターフェイス クラス」[p.5-10] を参照）
- `public void registerLogoutListener(LogoutListener listener)`
- クォータ リスナの設定（詳細は「QuotaListenerEx インターフェイス クラス」[p.5-10] を参照）
- `public void registerQuotaListener(QuotaListener listener)`
- 冗長ステート リスナの設定（詳細は「RedundancyStateListener インターフェイス」[p.5-15] を参照）
- `public void setRedundancyStateListener(RedundancyStateListener listener)`

（注）　API にリスナを登録すると、SCE 内でリソースが割り当てられて、リスナへの確実なメッセージ配信がサポートされます。API を使用するアプリケーションがクラッシュし、少しあとに再起動した場合でも、メッセージは保持され、API 再接続時に SCE に送信されます。

高度なセットアップ操作

API を使用すると、特定の内部プロパティを初期化して、API をカスタマイズできます。初期化には API **init** メソッドを使用します。

（注）　設定を有効にするには、**connect** メソッドより先に **init** メソッドを呼び出す**必要があります。**

次のプロパティを設定できます。

- 出力キュー サイズ — SCE に送信されるまで API で蓄積できる要求の最大数を定義する内部バッファ サイズ（デフォルト：1024)
- 処理のタイムアウト — 応答しない PRPC プロトコル接続に関する希望のタイムアウト間隔（ミリ秒）（デフォルト：45 秒）

構文

init メソッドの構文は、次のとおりです。

```
public void init(Properties properties)
```

パラメータ

properties（java.util. プロパティ）— 「高度なセットアップ操作」（p.5-5）に記載されているプロパティの設定をイネーブルにします。

- 出力キュー サイズを設定するには、プロパティ キーとして **prpc.client.output.machinemode.recordnum** を使用します。
- 操作のタイムアウトを設定するには、プロパティ キーとして **com.scms.api.sce.prpc.regularInvocationTimeout** または **com.scms.api.sce.prpc.listenerInvocationTimeout** を使用します。

（注）　**com.scms.api.sce.prpc.listenerInvocationTimeout** は、リスナ コールバックから呼び出すことができる操作に使用します。デッドロックを回避するために、このタイムアウトは **com.scms.api.sce.prpc.regularInvocationTimeout** よりも短く設定する必要があります。

プロパティのカスタマイズ例

初期化中にプロパティをカスタマイズする例を示します。

```
// API の構築
PRPC_SCESuscriberAPI sceApi = new PRPC_SCESuscriberAPI("MyApi",
"10.1.1.1",10000);
// API の初期化
java.util.Properties p = new java.util.Properties();
p.setProperty("prpc.client.output.machinemode.recordnum", 2048+"");
api.init(p);
// API との接続
sceApi.connect();
```

（注）　**init** メソッドは、**connect** メソッドよりも先に呼び出されます。

SCE との接続

API を設定したら、SCE との接続を試行する必要があります。自動再接続機能がアクティブな場合、これ以降に発生する切断はすべて API で処理されます。

SCE に接続するには、次のメソッドを使用します。

```
public void connect() throws Exception
```

API 操作中のどの時点でも、メソッド **isConnected()** を使用して、API が SCE に接続されているかどうかを確認できます。

```
public boolean isConnected()
```

（注）　どの API インスタンスも、ただ 1 つの SCE プラットフォームとの接続をサポートしています。

getApiVersion

構文

```
public String getApiVersion()
```

説明

このメソッドは API バージョンを問い合わせます。バージョンは <Major Version.Minor Version> としてフォーマットされたストリングです。

API の終了

サーバおよびクライアントのリソースを解放するには、**disconnect** メソッドを呼び出します。

```
public void disconnect()
```

disconnect メソッドを呼び出すと、SCE と API との接続の信頼性を管理する SCE 内のリソースが解放されます。アプリケーションが再起動していて、メッセージを失いたくない場合は、disconnect メソッドを使用しないでください。

main クラス内で **finally** 文を使用することを推奨します。次に例を示します。

```
public static void main(String [] args) throws Exception
{
PRPC_SCESubscriberApi sceapi = new PRPC_SCE_SubscriberApi ("myApi",
"sceHost");
try
{
                …
                   // コードをここに記述します。
        }
        finally
        {
                sceapi.disconnect();
        }
}
```

通知リスナ

特定のイベントが発生すると、SCE プラットフォームは複数のタイプの通知を発行します。次の 3 つの通知タイプがあります。

- ログインプル通知
- ログアウト通知
- クォータ通知

通知が送信されるのは、これらの通知を待ち受けるように登録されたリスナが存在する場合のみです。通知タイプごとに異なるリスナを登録できます。これらの通知をトリガーするイベントの詳細については、「API イベント」（p.3-1）の章を参照してください。

LoginPullListener インターフェイス クラス

LoginPullListener インターフェイスは、**プル** モデルでのみ使用される一連のコールバック関数を定義します。

プル モデルでの動作を想定しているポリシー サーバで、サブスクライバ プロビジョニング プロセスのネットワーク ID 管理を行う場合は、このポリシー サーバで LoginPullListener を登録して、SCE からのログインプル要求への応答や、SCE プラットフォームの同期化を実現する必要があります。

これらの通知の待ち受けをイネーブルにするために、API では次の通知タイプに対応するようにリスナを設定できます。

```
public void registerLoginPullListener(LoginPullListener listener)
public void unregisterLoginPullListener(LoginPullListener listener)
```

（注）　API でサポートされる LoginPullListener は一度に 1 つずつです。また、複数の API で同じ LoginPullListener を登録することも避けてください。両方の SCE が同じログインプル要求に応答した場合に、SCE プラットフォームが同期されなくなることがあります。

LoginPullListener は、ログインプル通知リスナを登録できるようにする場合に実装するインターフェイスです。このインターフェイスは次のように定義されます。

```
public interface LoginPullListener
{
public void loginPullRequest (String anonymousSubscriberID,
NetworkID networkID)
public void loginPullRequestBulk(NetworkAndSubscriberID_BULK  subs)

public void getSubscribersBulkResponse(
NetworkAndSubscriberID_BULK  subs,
SubscriberBulkResponseIterator iterator)
}
```

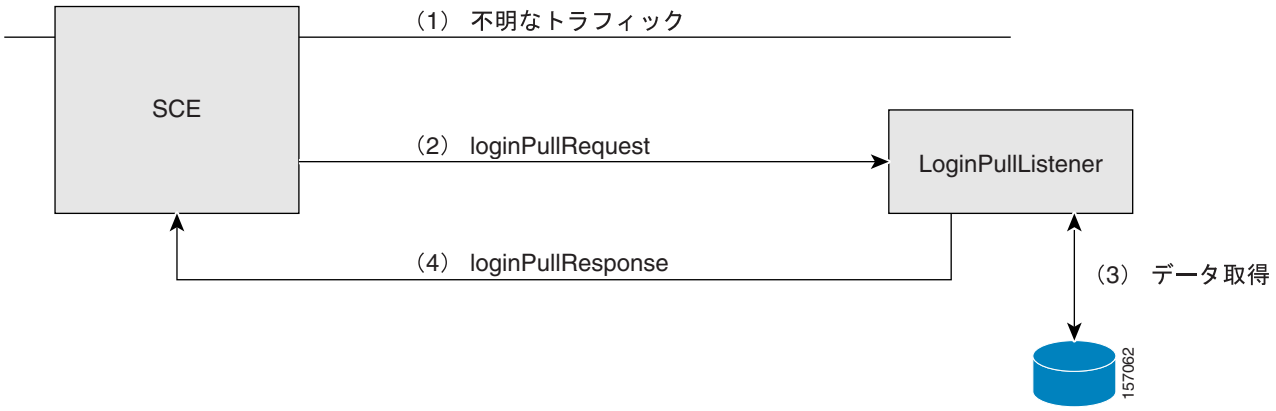
loginPullRequest コールバック メソッド

SCE で IP アドレスが不明なサブスクライバ側トラフィックが検出されると、SCE は IP アドレスに基づいてサブスクライバ ログイン情報要求を送信します（「プル モデル」[p.3-3] を参照）。通常、ポリシー サーバはこの IP が割り当てられたサブスクライバ データの設定データを使用して応答します。

この要求は登録済みリスナにディスパッチされ、**loginPullRequest** コールバック関数をトリガーします。このコールバック時に、リスナはこの IP アドレスに一致するサブスクライバのサブスクライバ情報を取得し、**loginPullResponse** をアクティブにして、その情報を SCE に配信する必要があります（「loginPullResponse 操作」[p.5-23] を参照）。この IP アドレスに対応する情報が存在しない場合、応答は発行されません。

次の図に、**loginPullRequest** コールバック メソッドを示します。

図 5-1　loginPullRequest コールバック メソッド



パラメータ

- **anonymousSubscriberID** — **loginPullResponse** 操作には、このアノニマス サブスクライバ ID を指定する**必要があります**（「loginPullResponse 操作」[p.5-23] を参照）。「アノニマス サブスクライバ ID」（p.2-2）も参照してください。
- **networkID** — 不明なサブスクライバのネットワーク ID。詳細は「ネットワーク ID」（p.2-2）を参照してください。

loginPullRequestBulk コールバック メソッド

このコールバック関数は、上記の **loginPullRequest** コールバック関数のバルク バージョンです。

パラメータ

- **subs** — 複数のサブスクライバの NetworkID とアノニマス ID のペアを含みます。詳細は、loginPullRequest コールバック メソッドの「パラメータ」（p.5-9）を参照してください。

ポリシー サーバがこの要求に応答するには、loginPullBulkResponse メソッドをアクティブにするか、バルク内のネットワーク ID ごとに loginPullResponse メソッドをアクティブにします。「loginPullResponseBulk 操作」（p.5-24）および「loginPullResponse 操作」（p.5-23）を参照してください。subs パラメータ内のデータに関して反復操作を実行するには、バルク クラスから提供される next() 反復メソッドを使用します（「バルク イテレータ」[p.4-8] を参照）。

GetSubscribersBulkResponse コールバック メソッド

このコールバック メソッドは、**プル** モデル内の SCE 同期プロセス中に使用します。詳細は、「SCE-API の同期」（p.5-33）を参照してください。

LogoutListener インターフェイス クラス

サブスクライバ プロビジョニング プロセスのネットワーク ID 管理操作を行うポリシー サーバでは、特定のサブスクライバが SCE プラットフォームから実際に削除された場合に通知される LogoutListener を登録しなければならないことがあります。

API を使用すると、ログアウト通知を受信できるように **LogoutListener** を設定できます。

```
public void registerLogoutListener(LogoutListener listener)
public void unregisterLogoutListener(LogoutListener listener)
```

（注）　API でサポートされる LogoutListener は一度に 1 つずつです。

ここでは、**LogoutListener** インターフェイスのコールバック関数について説明します。

- logoutIndication コールバック メソッド（p.5-10）
- logoutBulkIndication コールバック メソッド（p.5-10）

logoutIndication コールバック メソッド

subscriberID で識別されるサブスクライバの最終 Network ID のログアウトを識別すると、SCE プラットフォームは、ログアウト通知を発行します。これにより、登録されたすべてのログアウト通知リスナに関する **logoutIndication** コールバック関数が呼び出されます。

```
public void logoutIndication(String subscriberID)
```

パラメータ

- **subscriberID —** サブスクライバの一意の ID。詳細は、「サブスクライバ ID」（p.4-2）を参照してください。SCE はこのサブスクライバ ID を処理しなくなります。

logoutBulkIndication コールバック メソッド

サブスクライバ グループの最終 Network ID のログアウトを識別すると、SCE プラットフォームはログアウト バルク通知を発行します。これにより、登録されたすべてのログアウト通知リスナに関する **logoutBulkIndication** コールバック関数が呼び出されます。

```
public void logoutBulkIndication(SubscriberID_BULK subs)
```

パラメータ

- **subs —** ログアウトしたサブスクライバのサブスクライバ ID を含みます。詳細は「SubscriberID_BULK クラス」（p.4-11）を参照してください。

QuotaListenerEx インターフェイス クラス

（注）　バージョン 3.0.5 以降、QuotaListener インターフェイスは廃止されています。代わりに、QuotaListenerEx を使用してください。下位互換性を保つために QuotaListener インターフェイスは存続していますが、バージョン 3.0.5 の API と統合する場合は、QuotaListenerEx インターフェイスを使用する必要があります。

サブスクライバ プロビジョニング プロセスでクォータ管理操作を行うポリシー サーバでは、SCE プラットフォームから発行されるクォータ関連通知を受信できなければなりません。

API を使用すると、クォータ通知を受信できるように **QuotaListener** を設定できます。

```
public void registerQuotaListener(QuotaListener listener)
public void unregisterQuotaListener(QuotaListener listener)
```

（注）　API でサポートされる QuotaListener は一度に 1 つずつです。

（注）　下位互換性を保つために QuotaListener インターフェイスが使用されていますが、QuotaListenerEx を実装するオブジェクトを渡すことを推奨します。

ここでは、**QuotaListenerEx** インターフェイスのコールバック関数について説明します。

（注）　クォータ コールバック メソッドのバルク バージョンは、このリリースの API では使用されません。

- quotaStatusIndication コールバック メソッド（p.5-11）
- quotaStatusBulkIndication コールバック メソッド（p.5-12）
- quotaBelowThresholdIndication コールバック メソッド（p.5-12）
- quotaBelowThresholdIBulkndication コールバック メソッド（p.5-12）
- quotaDepletedIndication コールバック メソッド（p.5-12）
- quotaDepletedBulkIndication コールバック メソッド（p.5-13）
- quotaStateRestore コールバック メソッド（p.5-13）
- quotaStateBulkRestore コールバック メソッド（p.5-13）

quotaStatusIndication コールバック メソッド

クォータ ステータス通知は、特定のサブスクライバに特定のクォータ バケット セットの残りの値を配信します。この通知は SCE によって定期的に発行されるか、または **getQuotaStatus** 操作が呼び出された場合に発行され（「getQuotaStatus 操作」[p.5-31] を参照）、**quotaStatusIndication** コールバック関数を起動すると登録済みリスナに配信されます。

```
public void quotaStatusIndication(String subscriberID,
Quota quota)
```

パラメータ

- **subscriberID** — サブスクライバの一意の ID。詳細は「サブスクライバ ID」（p.4-2）を参照してください。
- **quota** — サブスクライバのクォータ。詳細は「サブスクライバ クォータ」（p.4-5）を参照してください。

quotaStatusBulkIndication コールバック メソッド

クォータ ステータス バルク通知は、サブスクライバ グループに特定のクォータ バケット セットの残りの値を配信します。この通知は SCE によって定期的に発行されるか、または **getQuotaStatus** 操作が呼び出された場合に発行され（「クォータ ステータス取得イベント」[p.3-6] を参照）、**quotaStatusIndication** コールバック関数を起動すると登録済みリスナに配信されます。

```
public void quotaStatusBulkIndication(Quota_BULK subs)
```

定期的に発行される通知の期間を設定できます。詳細については、『*Cisco Service Control Application for Broadband User Guide*』を参照してください。

パラメータ

- **subs** — サブスクライバのバルクのクォータ データを含みます。詳細は「Quota_BULK クラス」（p.4-14）を参照してください。

quotaBelowThresholdIndication コールバック メソッド

サブスクライバのクォータが設定済みしきい値よりも低下すると、SCE プラットフォームは **quotaBelowThresholdIndication** コールバック関数を起動して、登録済みリスナに配信される通知を発行します。

```
public void quotaBelowThresholdIndication(String subscriberID,
Quota quota)
```

パラメータ

- **subscriberID** — サブスクライバの一意の ID。詳細は「サブスクライバ ID」（p.2-2）を参照してください。
- **quota** — サブスクライバのクォータ。詳細は「サブスクライバ クォータ」（p.4-5）を参照してください。

quotaBelowThresholdIBulkndication コールバック メソッド

サブスクライバ グループのクォータが設定済みしきい値よりも低下すると、SCE プラットフォームは **quotaBelowThresholdBulkIndication** コールバック関数を起動して、登録済みリスナに配信される通知を発行します。

```
public void quotaBelowThresholdBulkIndication(Quota_BULK subs)
```

パラメータ

- **subs** — サブスクライバのバルクのクォータ データを含みます。詳細は「Quota_BULK クラス」（p.4-14）を参照してください。

quotaDepletedIndication コールバック メソッド

サブスクライバのクォータが枯渇すると、SCE プラットフォームは **quotaDepletedIndication** コールバック関数を起動して、登録済みリスナに配信される通知を発行します。

```
public void quotaDepletedIndication(String subscriberID,
Quota quota)
```

パラメータ

- **subscriberID** — サブスクライバの一意の ID。詳細は「サブスクライバ ID」（p.2-2）を参照してください。
- **quota** — サブスクライバのクォータ。詳細は「サブスクライバ クォータ」（p.4-5）を参照してください。

quotaDepletedBulkIndication コールバック メソッド

サブスクライバ グループのクォータが枯渇すると、SCE プラットフォームは **quotaDepletedBulkIndication** コールバック関数を起動して、登録済みリスナに配信される通知を発行します。

```
public void quotaDepletedBulkIndication (SubscriberID_BULK subs)
```

パラメータ

- **subs** — クォータが枯渇したサブスクライバの名前を含みます。詳細は「SubscriberID_BULK クラス」（p.4-11）を参照してください。

quotaStateRestore コールバック メソッド

サブスクライバがポリシー サーバにログインすると、ポリシー サーバは SCE へのログイン操作を実行します。SCE は **quotaStateRestore** コールバック関数を起動して、SCE 内のサブスクライバ クォータを復元するための要求をポリシー サーバに発行します。ポリシー サーバは「クォータ更新イベント」（p.3-5）を使用して、この関数に応答しなければなりません。

```
public void quotaStateRestore(String subscriberID,
Quota quota)
```

パラメータ

- **subscriberID** — サブスクライバの一意の ID。詳細は「サブスクライバ ID」（p.2-2）を参照してください。
- **quota** — サブスクライバのクォータ。詳細は「サブスクライバ クォータ」（p.4-5）を参照してください。この通知が作成された時点ですべてのクォータ バケットが空であるため、バケット ID 配列のサイズは 0 です。

quotaStateBulkRestore コールバック メソッド

サブスクライバ グループがポリシー サーバにログインすると、ポリシー サーバは SCE へのログイン操作を実行します。SCE は **quotaStateBulkRestore** コールバック関数を起動して、SCE 内のサブスクライバ クォータを復元するための要求をポリシー サーバに発行します。ポリシー サーバは「クォータ更新イベント」（p.3-5）を使用して、この関数に応答しなければなりません。

```
public void quotaStateBulkRestore(SubscriberID_BULK subs)
```

パラメータ

- **subs** — クォータが枯渇したサブスクライバの名前を含みます。詳細は「SubscriberID_BULK クラス」（p.4-11）を参照してください。

接続モニタリング

SCMS SCE Subscriber API は SCE プラットフォームとの接続をモニタします。SCE との接続確立時または切断時に特定の処理を実行するように要求しているポリシー サーバでは、ConnectionListener インターフェイスを実装できます。

ConnectionListener インターフェイス

API を使用すると、接続リスナを設定できます。

```
setConnectionListener(ConnectionListener listener)
```

接続リスナは、次のように定義されたインターフェイスです。

```
public interface ConnectionListener {
/**
* SCE との接続がダウンしている場合に呼び出されます。
*/
public void connectionIsDown();
/**
* SCE との接続が確立されている場合に呼び出されます。
*/
public void connectionEstablished();
}
```

SCE との同期を開始するには、接続確立コールバックを使用します。詳細は「SCE-API の同期」（p.5-33）を参照してください。

切断リスナ：例

次に、**stdout** にメッセージを出力して処理を戻す切断リスナの単純な実装例を示します。

```
import com.scms.api.sce.ConnectionListener;
public class MyConnectionListener implements ConnectionListener {
public void connectionIsDown(){
System.out.println("Message: connection is down.");
return;
}
public void connectionEstablished(){
System.out.println("Message: connection is established.");
// SCE との同期を開始するスレッドを起動します。
}
}
```

SCE カスケード トポロジのサポート

SCMS SCE Subscriber API は SCE カスケード トポロジをサポートしています。カスケード SCE プラットフォームに接続されているポリシー サーバはカスケード構成内のどの SCE がアクティブでどれがスタンバイなのかを認識する必要があります。ポリシー サーバはアクティブな SCE **だけに** ログオン操作を送信します。同様に、ポリシー サーバがサブスクライバ同期を実行しなければならないのはアクティブな SCE **だけ**です。

スタンバイ SCE は、アクティブ SCE からサブスクライバについて学習します。これによってステートフル フェールオーバーが実現されます。ポリシー サーバは、アップデートされた最新のサブスクライバ情報を取得できるように、フェールオーバー イベントを識別し、アクティブになった SCE と同期をとる必要があります。

アクティブな SCE を知るために、ポリシー サーバには RedundancyStateListener インターフェイスを実装できます。

- isRedundancyStatusActive メソッド（p.5-15）
- RedundancyStateListener インターフェイス（p.5-15）
- カスケード違反エラーを無視する SCE の設定（p.5-16）

isRedundancyStatusActive メソッド

API には、SCE の冗長ステートを監視する目的で、RedundancyStateListener インターフェイスと **isRedundnacyStatusActive** メソッドがあります。

```
public boolean isRedundancyStatusActive()
```

このメソッドからの戻り値の意味は次のとおりです。

- TRUE — SCE の現在の状態がアクティブである場合
- FALSE — そうでない場合

SCE がアクティブかどうかを検証するために、カスケード SCE の初回の接続時と SCE へのログオン操作の送信前にこのメソッドを使用することを推奨します。

RedundancyStateListener インターフェイス

カスケード SCE の状態変化を監視できるようにするため、この API では冗長ステート リスナの設定が可能です。

```
setRedundancyStateListener(RedundancyStateListener listener)
```

冗長ステート リスナには、カスケード SCE の冗長ステートがアクティブからスタンバイへ、あるいはその逆に変化すると呼び出されるコールバック メソッドを定義します。

冗長ステート リスナは、次のように定義されたインターフェイスです。

```
public interface RedundancyStateListener {
public void redundancyStateChanged(SCESubscriberApi sceApi,
boolean isActive);
}
```

（注）　ポリシー サーバは、アクティブになった SCE に対して同期手順を実行する必要があります。これは、SCE への接続確立時にポリシー サーバによって実行される手順と同様です。

（注）　API インスタンスごとに特定の SCE プラットフォームとの接続が確立されます。したがって、カスケードの設定では、2 つの SCE Subscriber API インスタンスが必要です。

パラメータ

- **sceApi** — 状態が変化した SCE を表す API インスタンス。このパラメータを使用すると、複数の SCE に 1 つのリスナを実装できます。
- **isActive** — SCE がアクティブになった場合 TRUE になります。SCE が非アクティブになると、FALSE になります。

カスケード違反エラーを無視する SCE の設定

SCE 3.1.0 は、スタンバイ SCE にログオン操作が実行されるとエラーを返すようにデフォルトで設定されています。この動作を変更するには、SCE に `ignore-cascade-violation` CLI を使用します。

カスケード違反を無視するように SCE を設定するには、SCE プラットフォームで次の CLI を使用します。

```
(config)#>management-agent sce-api ignore-cascade-violation
```

カスケード違反が無視されるかどうかを表示するには、SCE プラットフォームで次の CLI を使用します。

```
#>show management-agent sce-api
```

カスケード違反時にエラーを送信するように SCE を設定するには、SCE プラットフォームで次の CLI を使用します。

```
(config)#>no management-agent sce-api ignore-cascade-violation
```

このフラグをデフォルト値（カスケード違反時にエラーを送信する）に設定するには、SCE プラットフォームで次の CLI を使用します。

```
(config)#>default management-agent sce-api ignore-cascade-violation
```

（注）　カスケード違反の無視を SCE に設定するのは、既存の SCE API コードとの下位互換性が必要な場合だけにしてください。カスケード機能を十分に活用するためには、SCE の冗長ステートを監視して使用する必要があります。

結果処理

API を使用すると、**操作ごとに**結果ハンドラを設定して、操作結果を異なる方法で処理できます。

SCE で実行された操作結果が API に戻されると、OperationResultHandler インターフェイスの handleOperationResult コールバックが呼び出されます。

特定の操作で結果処理が不要な場合は、**handler** 引数に **null** を挿入します。

（注） すべての操作に、同じ操作結果ハンドラを渡すことができます。

OperationResultHandler インターフェイス

このインターフェイスは、API を通して実行される操作結果を受信する場合に実装します。

操作結果ハンドラは次の単純なメソッドを使用して呼び出されます。

```
public interface OperationResultHandler {
/**
* 結果を処理します。
*/
public void handleOperationResult(Object[] result,
OperationArguments handback);

}
```

API を通じて実行した操作結果について通知を受けたい場合は、このインターフェイスを実装する必要があります。

（注） OperationResultHandler インターフェイスは結果を取得する唯一の方法です。API メソッドが発信側に返された直後に、結果を返すことはできません。操作結果を受信できるようにするには、処理を呼び出すときに各操作の結果ハンドラを設定します（例を参照）。

次に、OperationResultHandler インターフェイスから返されるデータを示します。

- **result —** 実際の操作結果。配列内の各エントリは次のいずれかの値をとります。
  - **NULL —** 操作に成功したことを示します。
  - **OperationException —** 操作に失敗したことを示します（以下を参照）。非バルク操作の場合、結果配列にはエントリが 1 つのみ含まれます。
- バルク操作の場合、結果配列の各エントリは、バルク操作の関連エントリに対応します。
- **handback —** 操作を呼び出すたびに、API はこのオブジェクトを自動的に提供します。このオブジェクトには、呼び出し時に渡されたすべての引数など、呼び出された操作に関する情報が含まれています。この操作の入力引数は、API マニュアルに記載された引数名で取得されます。たとえば、このデータを使用して、操作失敗後にパラメータを検査 / 出力したり、操作呼び出しを反復したりできます。

（注） バルク オブジェクトを含む操作では、バルク内の特定の要素に対して操作が失敗した場合でも、バルクが終了するまでバルク処理が継続します。

OperationArguments クラス

処理名を取得するには、次のメソッドを使用します。

```
public String getOperationName()
```

引数名を取得するには、次のメソッドを使用します。

```
public String[] getArgumentNames()
```

特定の操作引数を取得するには、次のメソッドを使用します。引数として、操作シグニチャ内の処理の引数名を使用します。

```
public Object getArgument(String name)
```

例

OperationResultHandler インターフェイスの実装例：

```
public class MyOperationHandler implements OperationResultHandler
{
long sucessCounter = 0;
long errorCounter = 0;

public void handleOperationResult(Object[] result,
OperationArguments handback)
{
for (int index=0; index <result.length; index++)
{
if (result[index]==null)
{
// 成功
successCounter++;
}
else
{
// 失敗
errorCounter++;
// エラーの詳細を抽出
OperationException ex = (OperationException)result[index];
// 操作名を抽出
String operationName = handback.getOperationName();

// 操作名およびエラー メッセージを出力
System.out.println("Error for operation "+
operationName +":" +
ex.getErrorMessage());
// 操作引数を出力
String[] argNames = handback.getArgumentNames();
if (argNames!=null)
{
for (int j=0; j<argNames.length; j++)
{
System.out.println(argNames[j]+  "="+
handback.getArgument(argNames[j]));
}
}
}
}
}
}
```

（注）　上記の実装例は、正規の操作とバルク操作の両方に使用できます。

次に、ログイン操作の結果ハンドラの例を示します。

```
public class LoginOperationHandler implements OperationResultHandler
{

public void handleOperationResult(Object[] result,
OperationArguments handback)
{
for (int index=0; index <result.length; index++)
{
if (result[index]!=null)
{
// 失敗
// エラーの詳細を抽出
OperationException ex =
(OperationException)result[index];
// 操作名およびエラー メッセージを出力
System.out.println("Error for login operation "+
":" + ex.getErrorMessage());
// サブスクライバ ID パラメータ値を出力
System.out.println("subscriberID"+
handback.getArgument("subscriberID"));
}
}
}
}
```

OperationException クラス

com.scms.api.sce.OperationException Java クラスは、通常の Java の使用法と対照的に、SCMS SCE Subscriber API の関数エラーをすべて提供します。このような「対照的な」アプローチが採用されたのは、SCMS SCE Subscriber API に「言語およびプロトコルに依存しない」特性があり、以降に実装するすべての SCE API で同じ外観が必要となるためです（Java、C、C++）。各 OperationException 例外は、次の情報を提供します。

- 一意のエラー コード（**long**）
- 情報メッセージ（**java.lang.String**）
- サーバ側のスタック トレース（**java.lang.String**）

エラー コードおよびその意味についての詳細は、「エラー コードのリスト」（p.A-1）を参照してください。

サブスクライバ プロビジョニング操作

ここでは、サブスクライバ プロビジョニングのために使用できる API のメソッドを示します。各メソッドのシグニチャのあとに、その入力パラメータと戻り値を記載します。

SCE との接続が確立される前に呼び出されたすべてのメソッドは、**java.lang.IllegalStateException** を戻します。

ログオン操作

構文

```
void login(String subscriberID,
NetworkID networkID,
boolean networkIdAdditive,
PolicyProfile policy,
QuotaOperation quotaOperation,
OperationResultHandler handler) throws Exception
```

説明

SCE にサブスクライバを追加したり、サブスクライバを更新したりします。この操作は次のアルゴリズムに従って実行されます。

- サブスクライバ ID が SCE 内に存在しない場合は、データがすべて指定された新しいサブスクライバが追加されます。

- サブスクライバ ID が存在する場合は、次のようになります。
  - **networkIdAdditive** フラグが TRUE に設定されている場合は、サブスクライバの既存の networkID に、指定した networkID が追加されます。それ以外の場合は、指定した networkID で既存の networkID が置き換えられます。
  - **policy** — ポリシーは新しいポリシー値で*アップデートされます*。PolicyProfile 内で指定されていないサブスクライバ ポリシー エントリは、変更されないか、またはデフォルト値を使用して作成されたままになります。
  - **quota** — クォータはバケット値および与えられた操作に基づいて*アップデートされます*。「サブスクライバ クォータ」（p.4-5）を参照してください。
- 別のサブスクライバと輻輳している networkID がある場合は、別のサブスクライバの networkID が暗黙的にログアウトして、新しいサブスクライバがログインします。

関連イベントについては、「プッシュ モデル」（p.3-2）を参照してください。

パラメータ

subscriberID — サブスクライバの一意の ID。サブスクライバ ID のフォーマットについては、「サブスクライバ ID」（p.2-2）を参照してください。

networkID — サブスクライバのネットワーク ID。詳細は「ネットワーク ID のマッピング」（p.4-2）を参照してください。

networkIDAdditive — このフラグが TRUE に設定されている場合は、サブスクライバの既存の networkID に、指定した networkID が追加されます。それ以外の場合は、指定した networkID で既存の networkID が置き換えられます。

policy — サブスクライバのポリシー プロファイル。詳細は「SCA BB サブスクライバのポリシー プロファイル」（p.4-4）を参照してください。

quota — サブスクライバのクォータ。詳細は「サブスクライバ クォータ」（p.4-5）を参照してください。

handler — この操作の結果ハンドラ。*OperationResultHandler* インターフェイスについては、「結果処理」（p.5-17）を参照してください。

エラー コード

次に、このメソッドで返されることがあるエラー コードのリストを示します。

- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_RESOURCE_SHORTAGE
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

エラー コードについては、「エラー コードのリスト」（p.A-1）を参照してください。

例

次の例では、既存のマッピングは変更せずに、*john* という名前の既存のサブスクライバに IP アドレス 192.168.12.5 を追加します。

```
login(
"john",                              // サブスクライバ名
new NetworkID(new String[]{"192.168.12.5"},
SCESubscriberApi.ALL_IP_MAPPINGS),
true,                                // isMappingAdditive が true
null,                                // ポリシーなし
null);                               // クォータなし
```

次の例では、IP アドレス 192.168.12.5 を追加して、直前のマッピングを上書きします。

```
login(
"john",                              // サブスクライバ名
new NetworkID(new String[]{"192.168.12.5"},
SCESubscriberApi.ALL_IP_MAPPINGS),
false,                               // isMappingAdditive が false
null,                                // ポリシーなし
null);                               // クォータなし
```

その他の例については、「ログインおよびログアウト」（p.5-40）を参照してください。

loginBulk 操作

構文

```
void loginBulk(Login_BULK subsBulk,
OperationResultHandler handler) throws Exception
```

説明

バルク内のサブスクライバごとに、ログイン操作で指定されたロジックを適用します。

パラメータ

subsBulk — 「Login_BULK クラス」（p.4-9）を参照してください。

handler — この操作の結果ハンドラ。*OperationResultHandler* インターフェイスについては、「結果処理」（p.5-17）を参照してください。

エラー コード

次に、このメソッドで返されることがあるエラー コードのリストを示します。

- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_RESOURCE_SHORTAGE
- ERROR_CODE_OPERATION_ABORTED

- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

エラー コードについては、「エラー コードのリスト」（p.A-1）を参照してください。

loginPullResponse 操作

構文

```
void loginPullResponse(String subscriberID,
String anonymousSubscriberID,
NetworkID networkID,
PolicyProfile policy,
QuotaOperation quota,
OperationResultHandler handler) throws Exception
```

説明

SCE からの **loginPullRequest** 呼び出しへの応答、または **loginPullBulkRequest** への応答として、SCE にサブスクライバ ログイン情報を送信します。

関連イベントについては、「プル モデル」（p.3-3）を参照してください。

パラメータ

subscriberID — サブスクライバの一意の ID。サブスクライバ ID のフォーマットについては、「サブスクライバ ID」（p.2-2）を参照してください。

anonymousSubscriberID — アノニマス サブスクライバの ID。これは、loginPullRequest/loginPullBulkRequest 通知内で SCE により送信されます（「LoginPullListener インターフェイス クラス」[p.5-8] を参照）。詳細は「アノニマス サブスクライバ ID」（p.2-2）を参照してください。

networkID — サブスクライバのネットワーク ID。詳細は「ネットワーク ID のマッピング」（p.4-2）を参照してください。この ID には、loginPullRequest から受信したネットワーク ID が含まれている必要があります。SCE のこのサブスクライバに別のネットワーク ID が設定されている場合は、既存のネットワーク ID にこのネットワーク ID が追加されます。

policy — サブスクライバのポリシー プロファイル。詳細は「SCA BB サブスクライバのポリシー プロファイル」（p.4-4）を参照してください。

quota — サブスクライバのクォータ。詳細は「サブスクライバ クォータ」（p.4-5）を参照してください。

handler — この操作の結果ハンドラ。*OperationResultHandler* インターフェイスについては、「結果処理」（p.5-17）を参照してください。

エラー コード

次に、このメソッドで返されることがあるエラー コードのリストを示します。

- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_RESOURCE_SHORTAGE
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

エラー コードについては、「エラー コードのリスト」（p.A-1）を参照してください。

loginPullResponseBulk 操作

構文

```
void loginPullResponseBulk(LoginPullResponse_BULK subsBulk,
OperationResultHandler handler) throws Exception
```

説明

バルク内のサブスクライバごとに、loginPullResponse 操作で指定されたロジックを適用します。

関連イベントについては、「プル モデル」（p.3-3）を参照してください。

パラメータ

subsBulk — 「LoginPullResponse_BULK クラス」（p.4-12）を参照してください。

handler — この操作の結果ハンドラ。**OperationResultHandler** インターフェイスについては、「結果処理」（p.5-17）を参照してください。

エラー コード

次に、このメソッドで返されることがあるエラー コードのリストを示します。

- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_RESOURCE_SHORTAGE
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

エラー コードについては、「エラー コードのリスト」（p.A-1）を参照してください。

ログアウト操作

構文

```
void logout(String subscriberID,
NetworkID networkID,
OperationResultHandler handler) throws Exception
```

説明

SCE からサブスクライバの特定の networkID を削除します。この ID が指定されたサブスクライバの最終 networkID の場合は、SCE からサブスクライバが削除されます。サブスクライバ ID を指定しない場合は、このネットワーク ID が属するサブスクライバに関係なく、指定されたネットワーク ID が SCE から削除されます。ネットワーク ID を指定しない場合は、このサブスクライバのすべてのネットワーク ID が削除されます。

サブスクライバ レコードが SCE 内にない場合は、ログアウト操作に成功します。

関連イベントについては、「ログアウト イベント」（p.3-3）を参照してください。

パラメータ

subscriberID — サブスクライバの一意の ID。サブスクライバ ID のフォーマットについては、「サブスクライバ ID」（p.2-2）を参照してください。

networkID — サブスクライバのネットワーク ID。詳細は「ネットワーク ID のマッピング」（p.4-2）を参照してください。

handler — この操作の結果ハンドラ。**OperationResultHandler** インターフェイスについては、「結果処理」（p.5-17）を参照してください。

エラー コード

次に、このメソッドで返されることがあるエラー コードのリストを示します。

- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_OPERATION_ABORTED

エラー コードについては、「エラー コードのリスト」（p.A-1）を参照してください。

logoutBulk 操作

構文

```
void logoutBulk(NetworkAndSubscriberID_BULK  subsBulk,
OperationResultHandler handler) throws Exception
```

説明

バルク内のサブスクライバごとに、ログアウト操作で指定されたロジックを適用します。

関連イベントについては、「ログアウト イベント」（p.3-3）を参照してください。

パラメータ

subsBulk — 「NetworkAndSubscriberID_BULK クラス」（p.4-11）を参照してください。

handler — この操作の結果ハンドラ。**OperationResultHandler** インターフェイスについては、「結果処理」（p.5-17）を参照してください。

エラー コード

次に、このメソッドで返されることがあるエラー コードのリストを示します。

- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_OPERATION_ABORTED

エラー コードについては、「エラー コードのリスト」（p.A-1）を参照してください。

networkIdUpdate 操作

構文

```
void networkIDUpdate(String subscriberID,
NetworkID networkID,
boolean networkIdAdditive,
OperationResultHandler handler) throws Exception
```

説明

既存のサブスクライバのネットワーク ID を追加または置換します。

（注）　この操作が有効なのは、SCE にサブスクライバ レコードがある場合のみです。それ以外の場合、操作は失敗します。

関連イベントについては、「ネットワーク ID 更新イベント」（p.3-4）を参照してください。

パラメータ

subscriberID — サブスクライバの一意の ID。サブスクライバ ID のフォーマットについては、「サブスクライバ ID」（p.2-2）を参照してください。

networkID — サブスクライバのネットワーク ID。詳細は「ネットワーク ID のマッピング」（p.4-2）を参照してください。

networkIDAdditive — このフラグが TRUE に設定されている場合は、サブスクライバの既存の networkID に、指定した networkID が追加されます。それ以外の場合は、指定した networkID で既存の networkID が置き換えられます。

エラー コード

次に、このメソッドで返されることがあるエラー コードのリストを示します。

- ERROR_CODE_SUBSCRIBER_NOT_EXIST
- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_RESOURCE_SHORTAGE
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

エラー コードについては、「エラー コードのリスト」（p.A-1）を参照してください。

networkIdUpdateBulk 操作

構文

```
void networkIDUpdateBulk(NetworkAndSubscriberID_BULK subsBulk,
OperationResultHandler handler) throws Exception
```

説明

バルク内のサブスクライバごとに、networkIDUpdate 操作で指定されたロジックを適用します。

関連イベントについては、「ネットワーク ID 更新イベント」（p.3-4）を参照してください。

パラメータ

subsBulk — 「NetworkAndSubscriberID_BULK クラス」（p.4-11）を参照してください。

handler — この操作の結果ハンドラ。**OperationResultHandler** インターフェイスについては、「結果処理」（p.5-17）を参照してください。

エラー コード

次に、このメソッドで返されることがあるエラー コードのリストを示します。

- ERROR_CODE_SUBSCRIBER_NOT_EXIST
- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_RESOURCE_SHORTAGE
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

エラー コードについては、「エラー コードのリスト」（p.A-1）を参照してください。

profileUpdate 操作

構文

```
void profileUpdate(String subscriberID,
PolicyProfile policy,
OperationResultHandler handler) throws Exception
```

説明

既存のサブスクライバのポリシー プロファイルを変更します。サブスクライバ レコードが SCE 内にない場合は、この操作に失敗します。

関連イベントについては、「プロファイル更新イベント」（p.3-5）を参照してください。

パラメータ

subscriberID — サブスクライバの一意の ID。サブスクライバ ID のフォーマットについては、「サブスクライバ ID」（p.2-2）を参照してください。

policy — サブスクライバのポリシー プロファイル。詳細は「SCA BB サブスクライバのポリシー プロファイル」（p.4-4）を参照してください。

handler — この操作の結果ハンドラ。**OperationResultHandler** インターフェイスについては、「結果処理」（p.5-17）を参照してください。

エラー コード

次に、このメソッドで返されることがあるエラー コードのリストを示します。

- ERROR_CODE_SUBSCRIBER_NOT_EXIST
- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER

- ERROR_CODE_NO_APPLICATION_INSTALLED

エラー コードについては、「エラー コードのリスト」（p.A-1）を参照してください。

profileUpdateBulk 操作

構文

```
void profileUpdateBulk(PolicyProfile_BULK subsBulk,
OperationResultHandler handler) throws Exception
```

説明

バルク内のサブスクライバごとに、profileUpdate 操作で指定されたロジックを適用します。

関連イベントについては、「プロファイル更新イベント」（p.3-5）を参照してください。

パラメータ

subsBulk — 「PolicyProfile_BULK クラス」（p.4-14）を参照してください。

handler — この操作の結果ハンドラ。**OperationResultHandler** インターフェイスについては、「結果処理」（p.5-17）を参照してください。

エラー コード

次に、このメソッドで返されることがあるエラー コードのリストを示します。

- ERROR_CODE_SUBSCRIBER_NOT_EXIST
- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

エラー コードについては、「エラー コードのリスト」（p.A-1）を参照してください。

quotaUpdate 操作

構文

```
void quotaUpdate(String subscriberID,
QuotaOperation quotaOperation,
OperationResultHandler handler) throws Exception
```

説明

サブスクライバのクォータの更新処理を実行します。

関連イベントについては、「クォータ更新イベント」（p.3-5）を参照してください。

パラメータ

subscriberID — サブスクライバの一意の ID。サブスクライバ ID のフォーマットについては、「サブスクライバ ID」（p.2-2）を参照してください。

quotaOperations — サブスクライバのクォータに対して実行するクォータ処理。詳細は「サブスクライバ クォータ」（p.4-5）を参照してください。

handler — この操作の結果ハンドラ。**OperationResultHandler** インターフェイスについては、「結果処理」（p.5-17）を参照してください。

エラー コード

次に、このメソッドで返されることがあるエラー コードのリストを示します。

- ERROR_CODE_SUBSCRIBER_NOT_EXIST
- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

エラー コードについては、「エラー コードのリスト」（p.A-1）を参照してください。

quotaUpdateBulk 操作

構文

```
void quotaUpdateBulk(QuotaOperation_BULK subsBulk,
OperationResultHandler handler) throws Exception
```

説明

バルク内のサブスクライバごとに、quotaUpdate 操作のロジックを適用します。

関連イベントについては、「クォータ更新イベント」（p.3-5）を参照してください。

パラメータ

subsBulk — 「QuotaOperation_BULK クラス」（p.4-15）を参照してください。

handler — この操作の結果ハンドラ。**OperationResultHandler** インターフェイスについては、「結果処理」（p.5-17）を参照してください。

エラー コード

次に、このメソッドで返されることがあるエラー コードのリストを示します。

- ERROR_CODE_SUBSCRIBER_NOT_EXIST
- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

エラー コードについては、「エラー コードのリスト」（p.A-1）を参照してください。

getQuotaStatus 操作

構文

```
void getQuotaStatus(String subscriberID,
Quota quota,
OperationResultHandler handler) throws Exception
```

説明

指定されたクォータ バケット セットの現在のクォータ残量を問い合わせる要求を送信します。この要求のあとに、問い合わせ対象データを含む getQuotaStatusIndication が続きます（**非同期**）。「quotaStatusIndication コールバック メソッド」（p.5-11）を参照してください。

関連イベントについては、「クォータ ステータス取得イベント」（p.3-6）を参照してください。

パラメータ

subscriberID — サブスクライバの一意の ID。サブスクライバ ID のフォーマットについては、「サブスクライバ ID」（p.2-2）を参照してください。

quota — 取得するクォータ バケットの名前リスト（値は除外）を含みます。バケット名のみでの構築方法についての詳細は、「サブスクライバ クォータ」（p.4-5）を参照してください。

handler — この操作の結果ハンドラ。**OperationResultHandler** インターフェイスについては、「結果処理」（p.5-17）を参照してください。

エラー コード

次に、このメソッドで返されることがあるエラー コードのリストを示します。

- ERROR_CODE_SUBSCRIBER_NOT_EXIST
- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

エラー コードについては、「エラー コードのリスト」（p.A-1）を参照してください。

getQuotaStatusBulk 操作

構文

```
void getQuotaStatuBulk(Quota_BULK subsBulk,
OperationResultHandler handler) throws Exception
```

説明

このメソッドは、上記の getQuotaStatus メソッドのバルク バージョンです。

関連イベントについては、「クォータ ステータス取得イベント」（p.3-6）を参照してください。

パラメータ

subsBulk — 「Quota_BULK クラス」（p.4-14）を参照してください。

handler — この操作の結果ハンドラ。**OperationResultHandler** インターフェイスについては、「結果処理」（p.5-17）を参照してください。

エラー コード

次に、このメソッドで返されることがあるエラー コードのリストを示します。

- ERROR_CODE_SUBSCRIBER_NOT_EXIST
- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

エラー コードについては、「エラー コードのリスト」（p.A-1）を参照してください。

SCE-API の同期

SCE およびポリシー サーバで切断、ログオン メッセージの消失、リブートなどが発生して、サブスクライバのデータに矛盾が発生した場合は、問題が発生することがあります。これらの問題が発生すると、サブスクライバのトラフィックが別のサブスクライバであるかのように分類されたり、サブスクライバのトラフィックに不正なサービスが実行されたり、リソースが失われたりすることがあります。

このような矛盾を防止するには、API を使用して SCE とポリシー サーバ間のサブスクライバ データを同期させて、通信チャネルの信頼性をできるかぎり高い状態に維持します。同期を開始するのは、**常に**ポリシー サーバです（API を使用）。

（注） 同時に複数のポリシー サーバで同期プロセスを実行すると、SCE 内のサブスクライバ情報がすべてのサーバと整合性がとれなくなります。

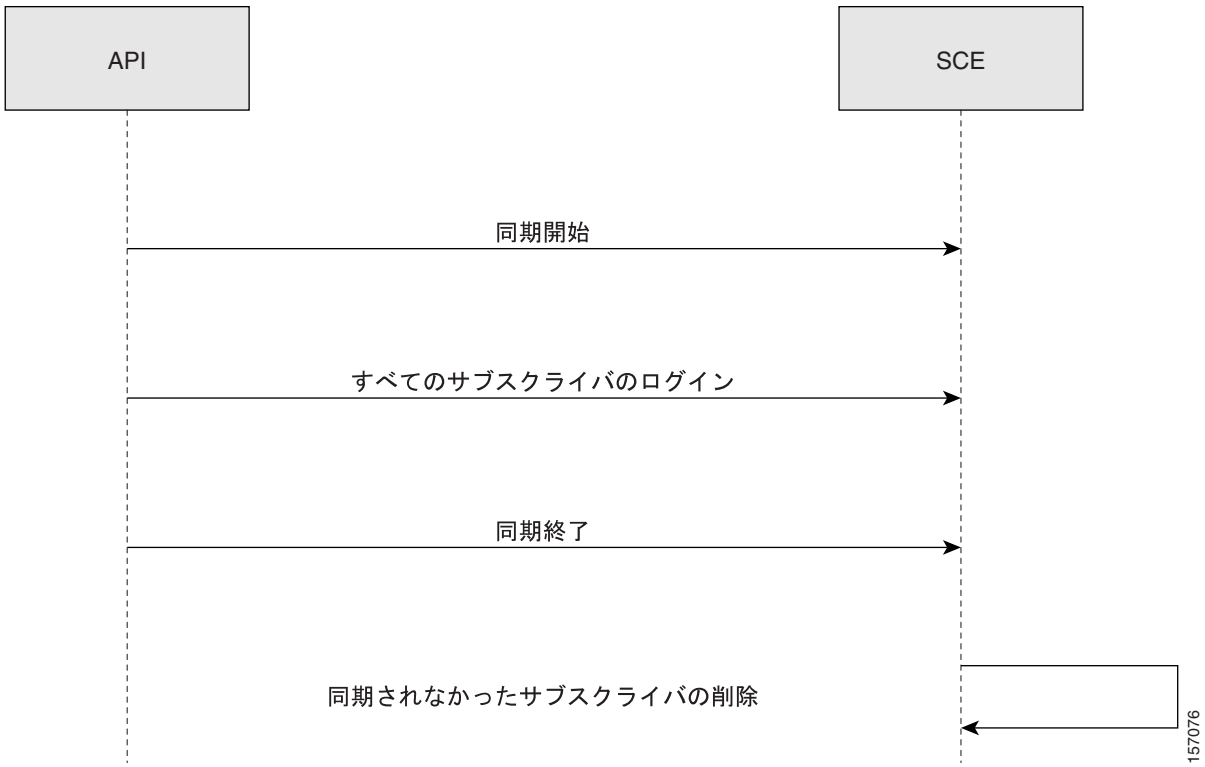
次に、同期実行中にポリシー サーバが従う必要がある、同期に関する注意事項を示します。

- プッシュ モデルの同期手順（p.5-33）
- プル モデルの同期手順（p.5-35）

プッシュ モデルの同期手順

1. ポリシー サーバから SCE に SCE の同期開始を通知します。
2. SCE が処理する必要があるすべてのサブスクライバに、ポリシー サーバがログインします。可能であれば、ログイン操作をバルクで実行します。
3. ポリシー サーバから SCE に同期終了を通知します。
4. SCE が同期プロセスに含まれないサブスクライバ データをすべて削除します。

図 5-2　プッシュ モデルの同期手順



（注）　同期プロセス中に、正規ログイン操作を実行できます。

次に、*プッシュ* モデルでの同期手順に使用するメソッドについて説明します。

- synchronizePushStart（p.5-34）
- synchronizePushEnd（p.5-35）

synchronizePushStart

構文

```
void synchronizePushStart(OperationResultHandler handler)
```

説明

この操作は、サーバとの同期を開始することを SCE に通知する場合に限って、*プッシュ* モデルで使用します。SCE はすべてのサブスクライバ データに「ダーティ ビット」をマークします。このデータが同期プロセス中に再適用された場合、ダーティ ビットはリセットされます。このメソッドを呼び出すたびに、同期プロセスは再起動されます。

パラメータ

handler — この操作の結果ハンドラ。**OperationResultHandler** インターフェイスについては、「結果処理」（p.5-17）を参照してください。

synchronizePushEnd

構文

```
void synchronizePushEnd(boolean success, OperationResultHandler handler)
```

説明

この操作は、サーバとの同期が終了したことを SCE に通知する場合に限って、*プッシュ* モデルで使用します。SCE はサブスクライバ データベース全体をスキャンして、**synchronizePushStart** で「ダーティ ビット」が割り当てられたデータを検索し、削除します。

パラメータ

success — 同期に成功したことを SCE に通知するフラグ

handler — この操作の結果ハンドラ。*OperationResultHandler* インターフェイスについては、「結果処理」（p.5-17）を参照してください。

プル モデルの同期手順

1. ポリシー サーバから SCE に SCE の同期開始を通知します。
2. ポリシー サーバが、現在処理中のサブスクライバ ID およびネットワーク ID をすべて SCE から取得します。
3. ポリシー サーバが同期外れをすべて修正します。

アルゴリズム

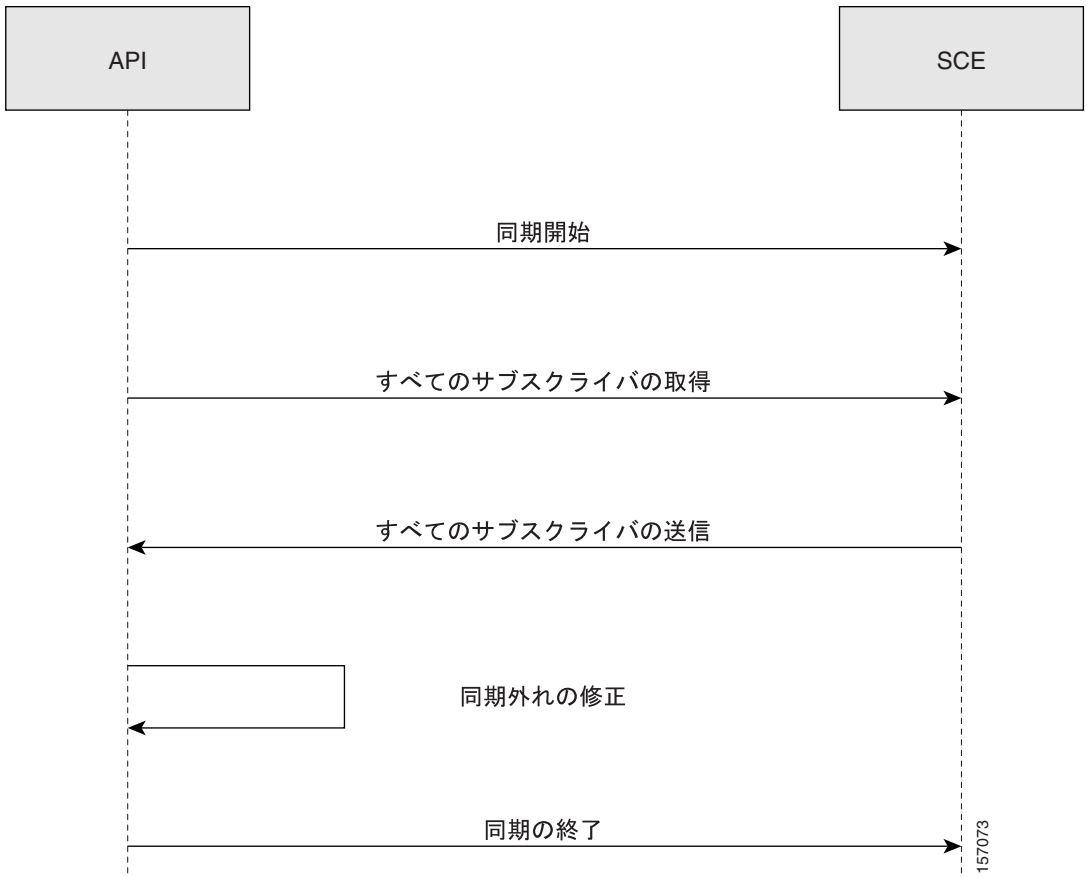
同期手順を計画する場合は、次のアルゴリズム テンプレートを使用します。

取得したサブスクライバ（<SubscriberID, IP address>）ごとに次の手順を実行します。

- ポリシー サーバ データベースに <SubscriberID, IP address> が存在する場合、
  SCE にポリシー プロファイルおよび networkID アップデートを送信します。
- それ以外の場合、
  SCE にログアウトおよびサブスクライバ IP を送信します。

ステップ 2 および 3 は同時に一括して実行されます。

図 5-3　プル モデルの同期手順



（注）　同期プロセス中に、正規ログイン操作を実行できます。

次に、*プル*モデルでの同期手順に使用するメソッドについて説明します。

- synchronizePullStart（p.5-36）
- synchronizePullEnd（p.5-37）
- getSubscribersBulk（p.5-37）

synchronizePullStart

- 構文（p.5-36）
- 説明（p.5-36）
- パラメータ（p.5-37）

構文

```
void synchronizePullStart(OperationResultHandler handler)
```

説明

この操作は、サーバとの同期を開始する必要があることを SCE に通知する場合に限って、*プル* モデルで使用します。

パラメータ

handler — この操作の結果ハンドラ。**OperationResultHandler** インターフェイスについては、「結果処理」（p.5-17）を参照してください。

synchronizePullEnd

構文

```
void synchronizePullEnd(boolean success, OperationResultHandler handler)
```

説明

この操作は、サーバとの同期が終了したことを SCE に通知する場合に限って、*プル* モデルで使用します。

パラメータ

handler — この操作の結果ハンドラ。**OperationResultHandler** インターフェイスについては、「結果処理」（p.5-17）を参照してください。

success — 同期に成功したことを SCE に通知するフラグ

getSubscribersBulk

構文

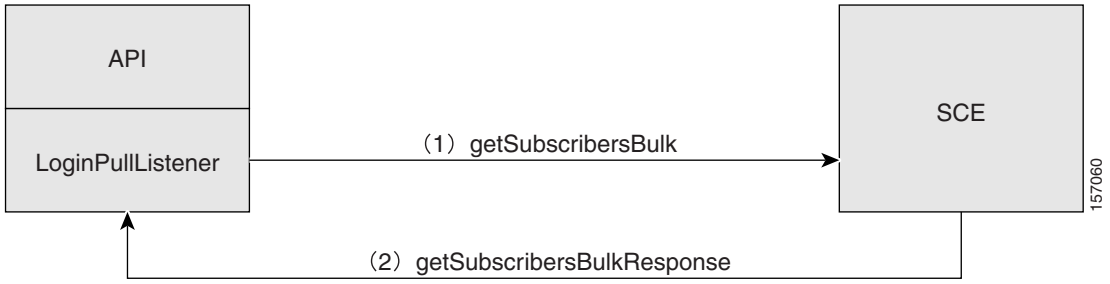
```
void getSubscribersBulk(int bulkSize,
SubscribersBulkResponseIterator iterator,
OperationResultHandler handler)
```

説明

この操作は、SCE が現在処理中のサブスクライバのバルクを取得するために、*プル* モデルの同期プロセスで使用します（「プル モデルの同期手順」[p.5-35]）。

この要求（**getSubscribersBulk**）を受信すると、SCE は subscriberID および対応する NetworkID を含む **getSubscribersBulkResponse** 通知を非同期に発行します（「LoginPullListener インターフェイス クラス」を参照）。このメソッドは、次に呼び出される **getSubscribersBulk** に渡されるイテレータを指定します。反復の終了を示すために、最後のバルクのイテレータは null になります。

図 5-4　Get Subscribers Bulk



パラメータ

bulkSize — 取得するバルクのサイズ。最大バルク サイズは 100 エントリに制限されています。

iterator — SCE 側のサブスクライバのイテレータ。このイテレータは getSubscribersBulkResponseIndication で受信され、次に呼び出される getSubscribersBulk メソッドに渡す必要があります。最初に getSubscribersBulk メソッドを呼び出す場合は、イテレータとして **null** を使用します（**null** を使用すると、最初から処理が開始されます）。

handler — この操作の結果ハンドラ。**OperationResultHandler** インターフェイスについては、「結果処理」（p.5-17）を参照してください。

高度な API プログラミング

ハイ アベイラビリティの実装

API のハイ アベイラビリティ サポート機能では、ポリシー サーバのハイ アベイラビリティ方式が 2 ノード クラスタ タイプであり、同時にアクティブにできるサーバは 1 台のみであると想定しています。別のサーバ（スタンバイ）は、SCE に接続されません。

アクティブ サーバに障害が発生すると、ユーザの 2 ノード クラスタ方式によって、スタンバイ サーバにフェールオーバーします。

（注）　SCE プラットフォームをプロビジョニングするポリシー サーバごとに、ハイ アベイラビリティをそれぞれ同時に実装できます。

SCMS SCE Subscriber API でハイ アベイラビリティを実装するには、次の作業を行う必要があります。

- 2 台のポリシー サーバに対応するように 2 ノード クラスタを設定します。
- *同じ API 名*を持つ API インスタンスをクラスタ内のサーバ（ノード）ごとに 1 つずつ、合計 2 つ構築します（コンストラクタの説明については、「API の構築」[p.5-4] を参照）。クラスタ実行中に SCE プラットフォームに接続する必要がある API インスタンスは 1 つのみです。フェールオーバーが発生すると、障害のあるサーバが SCE から切断され、スタンバイ サーバがアクティブになり、定義済みタイムアウト内に SCE に再接続します（「API 切断タイムアウトの設定」[p.1-7] を参照）。API 名が同じであるため、SCE は同じ API が再接続し、情報が失われないかのように動作します。

（注）　障害のあるポリシー サーバで使用する API 内で、**unregisterXXXListener** メソッドを暗黙的に呼び出さないでください。データが消失することがあります。**disconnect()** メソッドを呼び出しても、リスナは登録解除されません。

API コードの例

ここでは、API を使用するためのコード例を示します。

ログインおよびログアウト

次の例では、事前定義された数のサブスクライバが SCE にログインしたのち、ログアウトします。この例では自動再接続のサポートを使用するため、接続リスナは定義しません。

次のコード例には、成功した結果および失敗した結果をカウントする**結果ハンドラ**の実装例も含まれています。

```
// 操作結果を処理するクラス
import com.scms.api.sce.OperationArguments;
import com.scms.api.sce.OperationException;
import com.scms.api.sce.OperationResultHandler;
public class MyOperationResultHandler implements OperationResultHandler
{
long count = 0;

public void handleOperationResult(Object[] result,
OperationArguments handback)
{
for (int index=0; index <result.length; index++)
{
count++;
if (result[index]==null)
{
//100 回の操作ごとに success を出力します。
//if (++count%100 == 0)
{
System.out.println("\tsuccess "+count);
}
}
else // エラー - すべてのエラーを出力
{
// 失敗
count++;
// エラーの詳細を抽出
OperationException ex =
(OperationException)result[index];
// 操作名を抽出
String operationName = handback.getOperationName();
// 操作名およびエラー メッセージを出力
System.out.println("Error for operation "+
operationName+": "+
ex.getMessage());
}
}
}
public synchronized void waitForLastResult(int lastResult)
{
while (count<lastResult)
{
try
{
wait(100);
}
catch (InterruptedException ie)
{
ie.printStackTrace();
}
```

```
}
}
}
```

受信したログアウト通知数をカウントする単純な LogoutListener 実装を含むクラス：

```
import com.scms.api.sce.LogoutListener;
import com.scms.common.NetworkAndSubscriberID_BULK;
import com.scms.common.SubscriberID_BULK;
class MyLogoutListener implements LogoutListener
{
long count = 0;

public void logoutIndication(String subscriberID)
{
increaseCounter(1);
}

synchronized void increaseCounter(long value)
{
count = count + value;
}

synchronized long getCounter()
{
return count;
}
// 結果数「last result」を受け取るまで待機します。
public synchronized void waitForLastResult(int lastResult)
{
while (count<lastResult)
{
try
{
wait(100);
}
catch (InterruptedException ie)
{
ie.printStackTrace();
}
}
}

public void logoutBulkIndication(SubscriberID_BULK subs)
{
increaseCounter(subs.getSize());
}
}
```

main メソッドを含むクラス：

```
import com.scms.api.sce.prpc.PRPC_SCESubscriberApi;
import com.scms.common.*;
public class LogonPolicyServer {
public static void main (String args[]) throws Exception
{
int numSubscribersToLogin = 500;
//5 秒の再接続インターバルで API をインスタンス化します。
PRPC_SCESubscriberApi api = new PRPC_SCESubscriberApi(
"myAPI",
args[0], // SCE の IP
5000);
try {
// 操作結果ハンドラをインスタンス化します。
// すべての操作に対してハンドラを 1 つ使用します。
MyOperationResultHandler resultHandler =
new MyOperationResultHandler();
// ログアウト リスナをインスタンス化します。
MyLogoutListener listener = new MyLogoutListener();
// ログアウト通知に登録します。
api.registerLogoutListener(listener);
// SCE に接続します。
api.connect();
//ログイン
System.out.println("login of "+numSubscribersToLogin+
" subscribers");
PolicyProfile pp = new PolicyProfile(
new String[]{"packageId=1",
"monitor=1"});
for (int i=0; i<numSubscribersToLogin; i++)
{
api.login("sub"+i,
new NetworkID(getMappings(i), // ip を生成します。
NetworkID.ALL_IP_MAPPINGS),
true,                          // 追加フラグ
pp,                            // ポリシー
null,                          // クォータなし
resultHandler);
}
// サブスクライバがログインするまで待機します。
resultHandler.waitForLastResult(numSubscribersToLogin);
// すべてのサブスクライバをログアウトします。
System.out.println("logout of "+numSubscribersToLogin+
" subscribers");
for (int i=0; i<numSubscribersToLogin; i++)
{
NetworkID nid = new NetworkID(getMappings(i),
NetworkID.ALL_IP_MAPPINGS);
api.logout("sub"+i,nid,resultHandler);
}
// すべてのサブスクライバがログアウトするまで待機します。
// ただし、今回は、
// ログアウト リスナを使用して結果をカウントします。
listener.waitForLastResult(numSubscribersToLogin);
}
finally
{
api.unregisterLogoutListener
api.disconnect();
}
}
//サンプル プログラムの「自動」マッピング ジェネレータ
private static String[] getMappings(int i) {
return new String[]{ "10." +((int)i/65536)%256 + "." +
((int)(i/256))%256 + "." + (i%256)};
}
}
```

ログインプル要求およびログインプル応答

次のコード例は、ログインプル要求およびログインプル応答の操作方法を示します。

このクラスは、ログアウトおよびログイン プル通知のためのリスナの実装例です。

```
import java.util.Iterator;
// 直前の例の結果ハンドラ
import MyOperationResultHandler;
import com.scms.api.sce.*;
import com.scms.common.*;
class MyListener implements LoginPullListener, LogoutListener
{
// 通知カウンタ
long logoutCount = 0;
long pullCount=0;
// API インスタンス - SCE へのログインプル応答を送信する場合に使用します。
PRPC_SCESubscriberApi api = null;

// 直前の（ログインおよびログアウト）例
// から操作ハンドラを構築します。
MyOperationResultHandler h = new MyOperationResultHandler();
public MyListener(PRPC_SCESubscriberApi api)
{
this.api = api;
}
// ログアウト カウンタを増加させます。
public void logoutIndication(String subscriberID)
{
increaseLogoutCounter(1);
System.out.println("Got logout notification " +
getLogoutCounter());
}
// ログアウト カウンタを増加させます。
public void logoutBulkIndication(SubscriberID BULK subs)
{
System.out.println("Got logout notification");
increaseLogoutCounter(subs.getSize());
}
public void loginPullRequest (String anonymousSubscriberID,
NetworkID networkID)
{
try
{
increasePullCounter(1);
System.out.println("Got pull request" + getPullCounter());

// ポリシーを作成します。
PolicyProfile pp = new PolicyProfile(
new String[]{"packageId=1",
"monitor=1"});
// プル応答で応答します。
// サブスクライバ名を取得します。取得元は、
// ポリシー サーバ データベースなどです。
// この例では、サブスクライバ カウンタに基づく
// 固定名を使用します。
api.loginPullResponse(anonymousSubscriberID,
"sub"+getPullCounter(),
networkID,
pp,     // ポリシー
null,   // クォータなし
h); // 直前の例のハンドラ
}
catch (Exception ex)
{
System.out.println(ex.getMessage());
}
}
public void loginPullRequestBulk(NetworkAndSubscriberID BULK subs)
```

```
{
try
{
increasePullCounter(subs.getSize());
System.out.println("Got pull request" + getPullCounter());
// バルク形式のプル応答で応答します。
PolicyProfile pp = new PolicyProfile(
new String[]{"packageId=1",
"monitor=1"});
LoginPullResponse_BULK responseBulk =

new LoginPullResponse_BULK();
Iterator subsIterator = subs.getIterator();
// 受信したバルクを反復し（IP およびアノニマス ID）
// 応答バルクを作成します。
int count=0;
while(subsIterator.hasNext())
{
// サブスクライバ名を取得します。取得元は、
// ポリシー サーバ データベースなどです。
// この例では、サブスクライバ カウンタに基づく
// 固定名を使用します。
String subName = "sub_"+count;
SubscriberData sub = (SubscriberData)subsIterator.next();
// バルクからサブスクライバ マッピングを抽出し、
// これらのマッピングに基づいて新しい NetworkID を構築します。
NetworkID subNetId = new NetworkID(sub.getMappings(),
NetworkID.ALL_IP_MAPPINGS);
responseBulk.addEntry(sub.getAnonymousSubscriberID(),
subName,
subNetId,
true,
pp,
null);
count++;
}
// 上記のように構築されたバルクをバルク応答で使用します。
// 直前の例のハンドラを使用します。
api.loginPullBulkResponse(responseBulk,h);
}
catch (Exception ex)
{
System.out.println(ex.getMessage());
}
}
public void getSubscribersBulkResponse(
NetworkAndSubscriberID BULK subs,
SubscruberBulkResponseIterator iterator)
{
// この例では実装しません。
}

synchronized void increaseLogoutCounter(long value)
{
logoutCount = logoutCount + value;
}
synchronized void increasePullCounter(long value)
{
pullCount = pullCount + value;
}
synchronized long getPullCounter()
{
return pullCount;
}
synchronized long getLogoutCounter()
{
return logoutCount;
}
// 結果数「last result」を受け取るまで待機します。
```

```
public synchronized void waitForPullResult(int lastResult) {
while (pullCount<lastResult) {
try {
wait(100);
} catch (InterruptedException ie) {
ie.printStackTrace();

}
}
}
public synchronized void waitForLogoutResult(int lastResult) {
while (logoutCount<lastResult) {
try {
wait(100);
} catch (catch (InterruptedException ie) {
ie.printStackTrace();
}
}
}
}
```

main メソッドを含むクラス：

```
import java.util.Iterator;
import com.scms.api.sce.*;
import com.scms.common.*;
public class LogonPolicyServer {
static PRPC_SCESubscriberApi api = null;

// このサンプル プログラムは SCE からのプル要求を待機して、
// プル応答で応答します。
// 500 のサブスクライバがすべてログインすると、プログラムが終了します。
public static void main (String args[]) throws Exception
{
int numSubscribersToLogin = 500;
//5 秒の再接続インターバルで API をインスタンス化します。
api = new PRPC_SCESubscriberApi("myAPI","1.1.1.1",5000);
// 上記の例から、操作結果ハンドラを
// 構築します。
MyOperationResultHandler handler =
new MyOperationResultHandler();

// ログアウトおよびログインプル リスナをインスタンス化します。
MyListener listener = new MyListener(api);
try
{
// ログアウト通知に登録します。
api.registerLogoutListener(listener);
api.registerLoginPullListener(listener);
// SCE に接続します。
api.connect();

// SCE からのログインプル要求を待機します。
// これらの要求は、SCE に不明サブスクライバのトラフィックが
// ある場合に送信されます。
System.out.println("Waiting for pull requests for "+
numSubscribersToLogin+
" subscribers");
// すべてのサブスクライバがログインするまで待機します。
listener.waitForPullResult(numSubscribersToLogin);
//すべてのサブスクライバをログアウトします。
System.out.println("logout of "+numSubscribersToLogin+
" subscribers");
for (int i=0; i<numSubscribersToLogin; i++)
{
api.logout("sub"+i,null,handler);
}
// すべてのサブスクライバがログアウトするまで待機します。
listener.waitForLogoutResult(numSubscribersToLogin);
```

```
}
finally
{
api.unregisterLoginPullListener();
api.unregisterLogoutListener();
api.disconnect();
}
}
}
```