Quota Provisioning API

この章では、外部 Quota Provisioning（QP）API について説明します。

この章の内容は次のとおりです。

- 外部クォータ プロビジョニング（p.7-2）
- クォータ プロビジョニングのライフサイクル（p.7-4）
- 制限事項（p.7-5）
- 外部 Quota Provisioning API のインストール（p.7-5）
- QP API（Java）のメソッド（p.7-6）
- QP API（Java）のコード例（p.7-9）
- QP API（C）のメソッド（p.7-11）
- QP API（C）のコード例（p.7-13）
- エラー コードと例外処理（p.7-15）

外部クォータ プロビジョニング

外部クォータ プロビジョニングは、Service Control パートナーがアプリケーション認識能力のあるクォータ ベースおよびボリューム ベース サービスを作成するための、クォータ（割り当て量）実施メカニズムです。このメカニズムは、外部エンティティによるサブスクライバ レベルでのクォータ プロビジョニングを可能にし、各ベンダー製のサブスクライバ管理プラットフォームとの統合を実現します。

外部クォータ プロビジョニングについての詳細は、『*Service Control Application Suite for Broadband User Guide*』を参照してください。

Quota Provisioning API（QP API）は Service Control SM API の拡張機能であり、C/C++ および Java の両方のバージョンが用意されています。QP API は、SM への接続およびサブスクライバの追加と設定に関しては SM API の機能に依存します。QP API は、クォータの設定（setSubscriberQuota）、クォータの追加（addSubscriberQuota）、およびサブスクライバのクォータ バケットにあるクォータ残量の読み取り（getRemainingSubscriberQuota）といった機能を追加します。

SM API の詳細については、『*C/C++ API for SM Guide*』および『*Java API for SM Guide*』を参照してください。

これらの機能により、OSS システムで、このようなサービス モデルをプリペイド形式やクォータに基づく消費量で使用するアプリケーション / サービスに関して、サブスクライバのトラフィック使用状況を制御するためのロジックを開発できます。

外部クォータ プロビジョニングに対応するサービス コンフィギュレーション

システムに適用する SCAS BB サービス コンフィギュレーション（PQB）を作成するとき、QP API を有効に活用するためのいくつかのガイドラインがあります。

- パッケージ :
  - Package Quota Management Mode（パッケージ クォータ管理モード）を「External」に設定する必要があります。
  - バケットを設定するとき、適切なバケット タイプを設定する必要があります。使用可能なタイプは、「Volume（容量、L3 KB）」または「Number of Sessions（セッション数）」です。
  - サービス ルールにおける使用量制限の定義では、適切なバケットを選択する必要があります。サービス トラフィックは、選択したバケットからクォータを消費します。ルールの違反処理アクションを使用して、バケットの使用中にトラフィックに割り当てるサービス レベルを設定できます。
- **RDR**：関連する RDR の生成をアクティブにするには、RDR Settings で次の RDR をイネーブルにする必要があります。
  - Quota Breach（クォータ違反）RDR
  - Remaining Quota（クォータ残量）RDR
  - Quota Threshold（クォータ スレッシュホールド）RDR

パッケージ、ルール、および RDR の設定についての詳細は、『*Service Control Application Suite for Broadband User Guide*』を参照してください。

クォータ バケットの状態

ここでは、サブスクライバがクォータ バケットからクォータを消費するとき、および QP API によって追加のクォータをプロビジョニングするときの、クォータ バケットのさまざまな状態について説明します。

バケットの状態としては、次の 3 種類があります。

- **スレッシュホールドより上**：バケットが**スレッシュホールドより上**の場合、クォータは消費されており、クォータ残量が Remaining Quota RDR でレポートされます。
- **スレッシュホールドより下**：クォータの残量が**スレッシュホールドより下**になると、Quota Threshold RDR が生成されます。この RDR に対応して、クォータの追加または設定を行えば、バケットを再びスレッシュホールドより上にして、枯渇を未然に防げる可能性があります。クォータの残量は、クォータの消費に応じて Remaining Quota RDR でレポートされます。
- **枯渇（Depleted）**：クォータが 0 未満になると、バケットは**枯渇**します。この場合、バケットはクォータの欠損すなわち「マイナス」のクォータ残量を維持し、これが Remaining Quota RDR でレポートされます。「枯渇」の状態になると、Quota Breach RDR が生成され、ルールの違反処理アクション（定義されている場合）がトラフィックに適用されます。この時点でクォータの追加または設定を行えば、バケットの枯渇状態を解除できる可能性があります。

初期化されていないバケットのクォータ量は 0 です。つまり、クォータが 1 回でも使用されると、バケットが枯渇します。

クォータ プロビジョニングのライフサイクル

ここでは、クォータ プロビジョニング処理と、サブスクライバがログインおよびログアウトし、ネットワーク トラフィックを発生させる各段階でのサブスクライバのクォータ状態について、そのライフサイクルを説明します。

クォータ プロビジョニング処理に関して理解しておくべき重要な点は、QP API の関連するメソッドによって要求した Add または Set などの処理が、キューに格納されて即時に結果が返されるとしても、実際にクォータが変更されるのは、サブスクライバがトラフィックを生成したとき、すなわち、サブスクライバが何らかのネットワーク アクティビティを実行したとき*だけ*であるという点です。したがって、外部のプロビジョニング システムがクォータの変更を要求してから、その変更が実際に行われるまでの間、サブスクライバの状態はクォータ変更を反映しません。そのため、この中間的な時点でサブスクライバのクォータ残量を読み取っても、表示される値には、最新の変更が組み込まれていません。

次の図で、クォータ プロビジョニング処理およびサブスクライバのネットワーク アクティビティの結果、クォータが変化する例を示します。

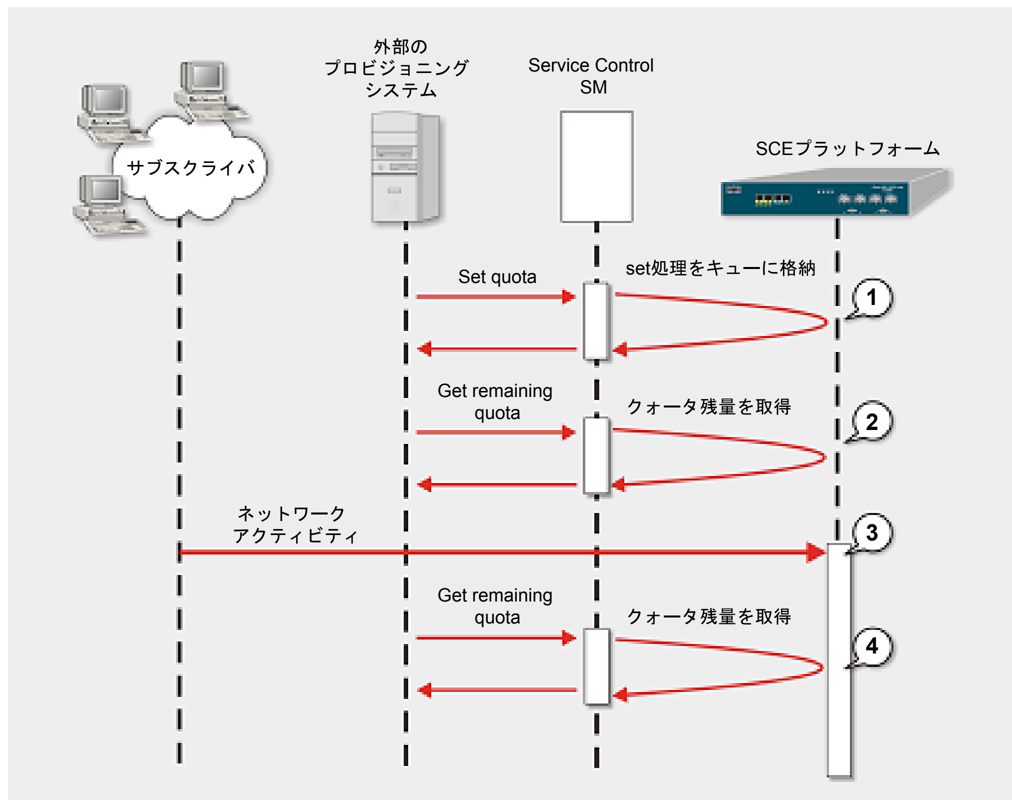
ステップ 1　外部クォータ プロビジョニング システムがクォータの変更（この例では set-quota）を実行すると、その変更はキューに格納され、まだ処理されません（図中 1）。

ステップ 2　したがって、外部クォータ プロビジョニング システムがクォータ残量を読み取ると、変更前の古い値が表示されます（図中 2）。

ステップ 3　サブスクライバがトラフィックを発生させて初めて、変更が行われます（図中 3）。

ステップ 4　このとき、外部クォータ プロビジョニング システムがクォータ残量を読み取ると、更新済みの値が表示されます（図中 4）。

図 7-1　クォータ プロビジョニングのライフサイクル



制限事項

- **プラスのクォータ残量は、バケットごとに 256 GB** という上限があります。

  この 256 GB という限界を超えてサブスクライバのクォータを増やそうとすると、エラーにはなりませんが、結果のクォータ残量がマイナスに変わります。したがって、add-quota コマンドを使用するときは注意が必要です。

  同様に、サブスクライバのクォータ欠損は、バケットごとに最大で 256 GB です。サブスクライバのいずれかのバケットが欠損しているとき、そのバケットの残量はマイナスの値（-256 GB まで）です。このマイナスの値をさらに超過すると、システムは過剰消費されたバケットへの課金を停止し、残量は -256 GB のままになります。

- **連続 256 回の set-quota 処理に関する問題**

  これも上記の問題と同様ですが、サブスクライバがログオフしているときや非アクティブのときは、set-quota コマンドによるクォータ残量の設定を頻繁に行うべきではありません。正確には、サブスクライバが非アクティブのときに set-quota コマンドで 256 回連続してクォータ バケットを更新すると、エラーにはなりませんが、バケットの残量が不正確になります。

外部 Quota Provisioning API のインストール

C および Java に対応する QP API は、それぞれ別のファイルでパッケージ化されています。このファイルの内容は次のとおりです。

- qp-c-api-dist.tar.gz
  - マニュアル
  - インクルード ファイル
  - Solaris SO ファイル
  - WinNT DLL ファイルおよび LIB ファイル
- qpapi.jar
- qpapi-javadoc.zip

インストールするには、最初に SM API をインストールしたあと、QP API を同じ手順でインストールします。

コンパイルして実行するには、SM API のコンパイルおよび実行手順に従い、QP API ファイルも *PATH/Class-path* に追加します。Java API を使用する場合は、classpath に *um_core.jar* をインクルードします。*um_core.jar* は、SCAS BB パッケージに含まれています。

QP API（Java）のメソッド

ここでは、Java 用のブロッキング QP API のメソッドを示します。各メソッドのシグニチャに続き、入力パラメータおよび戻り値を説明します。

addSubscriberQuota

構文

```
void addSubscriberQuota(String subscriberName,
                        int[] quota)
throws QPApiException, ConnectionDownException
```

説明

特定のサブスクライバのクォータ バケットの現在のクォータに、特定のクォータを追加します。

パラメータ

`subscriberName`：サブスクライバ ID

`quota`：特定のサブスクライバのクォータ バケットの現在のクォータに追加する、16 のクォータ値（L3 KB またはセッション数）の配列

戻り値

なし

addSubscriberQuota

構文

```
void addSubscriberQuota(String subscriberName,
                              int bucketNum
                        int[] quota)
throws QPApiException, ConnectionDownException
```

説明

特定のサブスクライバの特定のクォータ バケットの現在のクォータに、特定のクォータを追加します。

パラメータ

subscriberName：サブスクライバ ID

bucketNum：バケット番号

quota：特定のサブスクライバのクォータ バケットの現在のクォータに追加するクォータ値（L3 KB またはセッション数）

戻り値

なし

getSubscriberQuota

構文

```
int[] getSubscriberQuota(String subscriberName)
throws QPApiException, ConnectionDownException
```

説明

特定のサブスクライバの各クォータ バケットのクォータ残量を取得します。

パラメータ

subscriberName：サブスクライバ ID

戻り値

特定のサブスクライバの各クォータ バケットにある、クォータ残量の値（L3 KB またはセッション数）の配列

setSubscriberQuota

構文

```
void setSubscriberQuota(String subscriberName,
                        int[] quota)
throws QPApiException, ConnectionDownException
```

説明

特定のサブスクライバのクォータ バケットに、特定のクォータを設定します。

パラメータ

`subscriberName`：サブスクライバ ID

`quota`：特定のサブスクライバのクォータ バケットに設定する、16 のクォータ値（L3 KB またはセッション数）の配列

戻り値

なし

QP API（Java）のコード例

ここでは、QP API（Java）を使用したコード例を示します。

```
import java.util.Arrays;
import com.cisco.apps.scas.api.QPApiException;
import com.cisco.apps.scas.api.QPBlockingApi;
import com.cisco.management.framework.client.ConnectionDownException;
/**
 * External Quota Provisioning Example
 **/
public class ExternalQPExample {

                public static final String SM_ADDRESS = "10.1.12.65";

                static public void main(String[] args) throws Exception{

                        QPBlockingApi qpBlockingApi = null;
                        try{
                                //instantiate api and create a connection
                                qpBlockingApi = new QPBlockingApi();
                                qpBlockingApi.connect(SM_ADDRESS);

                                int[] defaultQuota = new int[16];

                              //provision each bucket of subscriber "sub1"
                              //with 1000 KBytes or 1000 Sessions.
                                Arrays.fill(defaultQuota,1000);
                                qpBlockingApi.setSubscriberQuota( "sub1",
defaultQuota);

                              //dd to bucket 2 of subscriber "sub1"
                              //with 500 KBytes or 500 Sessions.
                                qpBlockingApi.addSubscriberQuota( "sub1", 0, 500);

                               //get "sub1" remaining quota - this method
```

```
                              //invocation will work only if the
                              //subscriber is logged in.
                              //The remaining quota will reflect
                              //the last two modifications if
        //subscriber has generated traffic
                                int[] remainingQuota =
                                               qpBlockingApi.getRemainingSu
bscriberQuota("sub1");
                         printRemainingQuota( remainingQuota);

                        }catch(QPApiException e){
                                System.out.println("Error Code is: " + e.getCode()
);
                                System.out.println("Error Message is: " +
e.getMessage());
                        }catch(ConnectionDownException e){
                               System.out.println("Error due to connection failure:
" + e.getMessage());
                        }catch( IllegalStateException e ){
                               System.out.println("Error due to connection failure:
" + e.getMessage());
                        }finally{
                                if( qpBlockingApi != null &&
qpBlockingApi.isConnected() )
                                        qpBlockingApi.disconnect();
                        }

                        System.exit(0);
                }

           private static void printRemainingQuota(int[] remainingQuota) {
                        for (int bucketIdx = 0;
                                bucketIdx < remainingQuota.length;
                                bucketIdx++) {
                                System.out.println(
                                        "bucketIdx="
                                                + bucketIdx
                                                + ",remainingQuota="
                                              + remainingQuota[bucketIdx]);
                        }
                }


}
```

QP API（C）のメソッド

ここでは、C に対応するブロッキング QP API のメソッドを示します。各メソッドのシグニチャに続き、入力パラメータおよび戻り値を説明します。

（注）

QP API C 関数名には「QPB_」というプレフィクスが付きます。すべての QP API C 関数で、最初のパラメータは、`argApiHandle`（`init` 関数をコールして作成される API ハンドル）です。

addQuota

構文

```
ReturnCode* QPB_addQuota (QPB_HANDLE argApiHandle, char* argName, int* argQuotas)
```

説明

特定のサブスクライバのクォータ バケットの現在のクォータに、特定のクォータを追加します。

パラメータ

`argName`：サブスクライバ ID

`argQuotas`：特定のサブスクライバのクォータ バケットの現在のクォータに追加する、16 のクォータ値（L3 KB またはセッション数）の配列

戻り値

ReturnCode 構造体のポインタ

getRemainingQuota

構文

```
ReturnCode* QPB_getRemainingQuota (QPB_HANDLE argApiHandle, char* argName)
```

説明

特定のサブスクライバの各クォータ バケットのクォータ残量を取得します。

パラメータ

`argName`：サブスクライバ ID

戻り値

特定のサブスクライバの各クォータ バケットにあるクォータ残量の値（L3 KB またはセッション数）の整数配列を格納した **`ReturnCode`** 構造体のポインタ

setQuota

構文

```
ReturnCode* QPB_setQuota (QPB_HANDLE argApiHandle, char* argName, int* argQuotas)
```

説明

特定のサブスクライバのクォータ バケットに特定のクォータを設定します。

パラメータ

`argName`：サブスクライバ ID

`argQuotas`：特定のサブスクライバのクォータ バケットに設定する、16 のクォータ値（L3 KB またはセッション数）の配列

戻り値

ReturnCode 構造体のポインタ

QP API（C）のコード例

ここでは、QP API（C）を使用したコード例を示します。

- サブスクライバのクォータの設定、クォータの追加、およびクォータ残量の取得

```
#include <stdio.h>
#include "QpApiBlocking_c.h"
void onExampleDisconnect()
{
    printf("*** DISCONNECTED ***\n");
}
int example(char* argSmAddress)
{
    // init
    printf("initializing\n");
    QPB_HANDLE api;
    api = QPB_init(10,0,20000,10,30);
    if (api == NULL) {
        printf("init failed\n");
        return -1;
    }
    QPB_setName(api,"qp-example");

    // set a disconnect-listener
    QPB_setDisconnectListener(api,onExampleDisconnect);

    // connect
    printf("connecting\n");
    int cnt = 0;
    while (QPB_connect(api,argSmAddress,14374) == false) {
        if (cnt++ > 10) {
            printf("connect failed, too many reconnects, aborting\n");
            return -1;
        }
    }


    // quota operations

    // prepare a quota bucket array [100, 200, 300,...]
    int quotas[16];
    for (int i = 0; i < 16; ++i) {
        quotas[i] = (i+1)*100;
    }

    // set the quota to subscriber "subs1"
    printf("setting quota\n");
    ReturnCode* rt;
rt = QPB_setQuota(api,(char*)"subs1",quotas);
    if (isReturnCodeError(rt)) {
        printf((char*)"set-quota failed\n");
        printReturnCode(rt);
        return -1;
    }
    freeReturnCode(rt);
// add quota to subscriber "subs2"
```

```
    printf("adding quota\n");
    rt = QPB_addQuota(api,(char*)"subs2",quotas);
    if (isReturnCodeError(rt)) {
        printf((char*)"add-quota failed\n");
        printReturnCode(rt);
        return -1;
    }
    freeReturnCode(rt);

    // getting remaining quota of subscriber "subs3"
    // this method invocation will work only if the subscriber is logged in.
    // the remaining quota will reflect recent modifications if the
    // subscriber has generated traffic
    printf("getting remaining quota\n");
    rt = QPB_getRemainingQuota(api,(char*)"subs3");
    printReturnCode(rt);
    if (isReturnCodeError(rt)) {
        printf("get-remaining-quota failed\n");
        return -1;
    }
    freeReturnCode(rt);


    // disconnect
    if (QPB_disconnect(api) == false) {
        printf("disconnect failed\n");
        return -1;
    }
    QPB_release(api);

    return 0;
}


int main(int argc, char* argv[])
{
    char* smAddress = (char*)"10.1.12.82";
    printf("SM address: %s\n",smAddress);

    if (example(smAddress) < 0) {
        printf("example failed\n");
        return -1;
    }
    return 0;
}
```

エラー コードと例外処理

Quota Provisioning API のエラー コード

次の表に、Quota Provisioning API で発生する可能性のあるエラー コードを示します。Quota Provisioning API を SM API と併用する場合は、後者のエラー コードが表示されることもあります。そのようなエラー コードの詳細は、『*smartSUB Programmer's Guide*』の*付録 A* を参照してください。

表 7-1　Quota Provisioning API のエラー コード

エラー コード	説明	推奨する措置 / 注意事項
40,000	不正な引数による例外 : 指定したクォータの値が不正です（たとえば、値が大きすぎる）。	引数に指定したすべてのクォータ値が有効かどうかを確認します。
40,002	ログインしていないサブスクライバに関する例外 : ログインしていないサブスクライバに関して処理を実行しようとしました。	サブスクライバが SCE デバイスにログインしてから再度実行してください。
40,003	不明の例外 : 予期されないエラーが発生しました。	カスタマー サポートに連絡してください。
40,004	タイムアウト例外 : SM データベースが長期にわたってロックされた場合に発生します（内部エラー）。	カスタマー サポートに連絡してください。
40,030	非アクティブなサブスクライバに関する例外 : サブスクライバのコンテキストが予想外に見つからない場合に発生します（内部エラー）。	カスタマー サポートに連絡してください。

いずれのエラーも、すべての QP API メソッドで発生するわけではありません。

Java API での例外の管理

Java API で発生する例外は、次の 3 タイプです。

- **ConnectionDownException** : SM への確立済みの接続が、処理の途中で切断された場合に発生します。
- **IllegalStateException** : SM への接続が突発的に切断された場合に発生することがあります。
- **QPApiException** : その他のすべてのエラー。

QPApiException が発生した場合は、qpApiException.getCode() を使用し、上記の（または SM のマニュアルの*付録 A* にある）表の「エラー コード」欄と比較してください。qpApiException.getMessage() を使用して、ストリング エラー メッセージを受信します。

C/C++ API でのエラー コードの管理

関数コールが失敗すると、エラー コードが返されます。そのエラー コードを、上記の（または SM のマニュアルの*付録 A* にある）表の「エラー コード」欄と比較してください。