Cisco DCNM プログラマブル レポート API

テンプレート

Cisco DCNM リリース 11.4(1) では、新しいテンプレート タイプ「REPORT」が 2 つのサブタイプ、UPGRADE および GENERIC とともに追加されています。テンプレート タイプは python であり、「generateReport」メソッドの実装を提供する必要があります。

アップグレード

UPGRADE テンプレートは、ISSU の前後の ISSU に使用されます。これらのテンプレートは、ISSU ウィザードに表示されます。

GENERIC

GENERIC テンプレートは、一般的なレポート目的で使用できます。たとえば、インベントリ レポートの収集です。

テンプレート構造

次の図は、テンプレート構造の例を示しています。

```
##template variables


@(IsInternal=true)
String serial_number/fabric_name;

String user_input;

##
##template content


from com.cisco.dcbu.vinci.rest.services.jython import WrappersResp
from reportlib.preport import *

def validate(context):

    respObj = WrappersResp.getRespObj()
    respObj.setSuccessRetCode()
    return respObj


def generateReport(context):

    report = Report("Report title")

    ##Report content

    respObj = WrappersResp.getRespObj()
    respObj.setSuccessRetCode()
    respObj.setValue(report)
    return respObj
##
```

serial_number or fabric_name based on the scope selected while scheduling the report. In case of UPGRADE report, always serial number will be injected

Template variable section. **All data types, annotations supported in DCNM template can be used here. User should provide these inputs while creating the report

Import necessary python lib.

Report can have optional validation method, This method will be invoked while creating the report job. Job will be created only if this method return success. This method is invoked only once and serial_number or fabric_name will not be available inside this method. More information check the API guide

report must provide implementation of generateReport(context) method.

generateReport method should return an object of type WrapperResp. Report object created above must be stored in wrappersResp using wrappersResp.setValue() API

テンプレート機能

generateReport メソッド

レポートの生成中に generateReport メソッドが呼び出されます。すべてのレポート導入ロジックを提供する必要があります。このメソッドは、コンテキストオブジェクトを受け入れます。前述のように、このメソッドは WrappersResp オブジェクトを返す必要があります。

検証メソッド

検証メソッドはオプションです。テンプレートでこのメソッドが定義されている場合、レポートアプリケーションはこのメソッドを呼び出して、ジョブの作成中に事前検証を実行します。このメソッドは、選択されたデバイスまたはファブリックに関係なく、ジョブが作成され、1回だけ呼び出された場合にのみ呼び出されます。

検証を通過した場合、このメソッドは WrappersResp を SuccessRetCode とともに返し、失敗の場合は FailureRetCode をエラー リストのエラーとともに返す必要があります。

次に例を示します。

検証に失敗しました

```
def validate (context):
```

```
    respObj = WrappersResp.getRespObj()

    ## Validation logic here

    respObj.setFailureRetcode()
    respObj.addErrorReport(template_name,error)
    return respObj
```

検証に成功しました

```
def validate (context):
    respObj = WrappersResp.getRespObj()

    ## Validation logic here

    respObj.setSuccessRetcode()
    return respObj
```

コンテキスト パラメータの内容に基づいて検証を実行できます。

コンテキスト パラメータ

コンテキスト パラメータは、次の属性で構成されます。

1. ユーザ名：ジョブを作成したユーザの名前
2. ユーザーロール：ジョブを作成したユーザのロール
3. Job ID
4. 再発：現在、1 回、毎日、毎週、毎月、オンデマンド、または定期
5. 期間：繰り返しが定期的である場合、期間には頻度が選択されます。たとえば、10 分です。

これらの値をコンテキストから読み取るには、「ジョブ コンテキスト情報の取得」で説明されている API を参照してください。

レポート レイアウト

レポートには次のコンポーネントがあります。

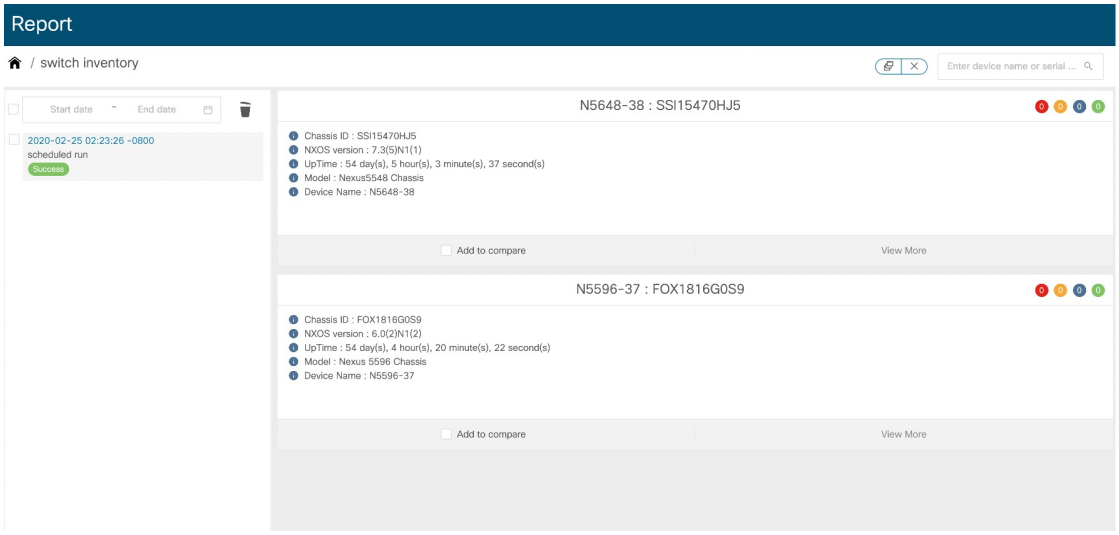
1. 要約
    1. キーと値
    2. メッセージ – 推論
2. 詳細/セクション
    1. キーと値
    2. JSON ドキュメント – カード

3. JSON ドキュメントの配列 – テーブル

3. コマンド ログ

一覧ビュー

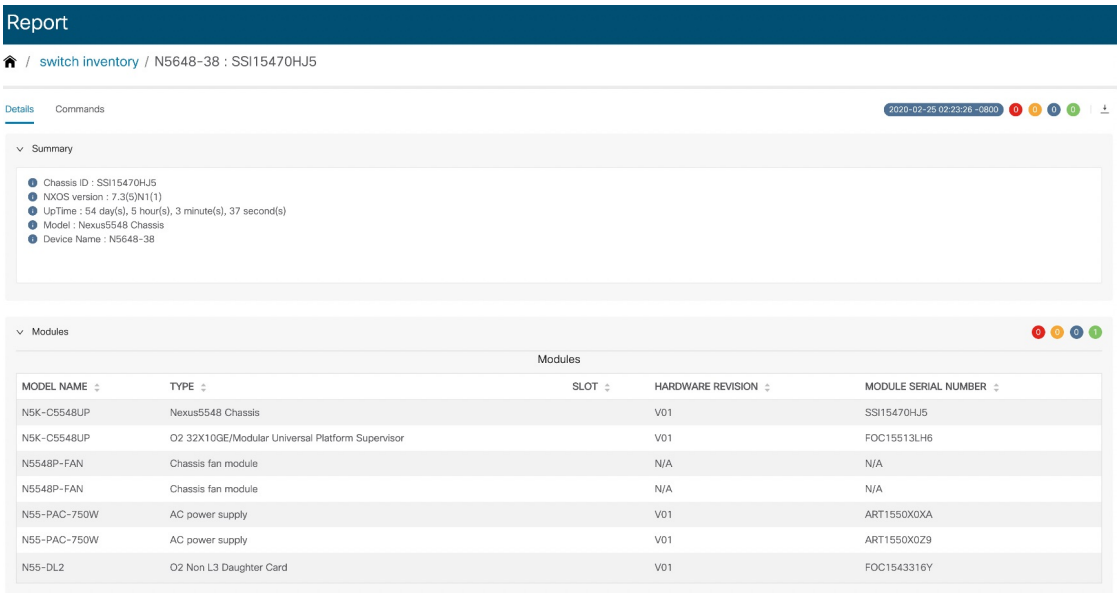
このビューには、レポートに含まれる各エンティティのサマリが表示されます。



詳細ビュー

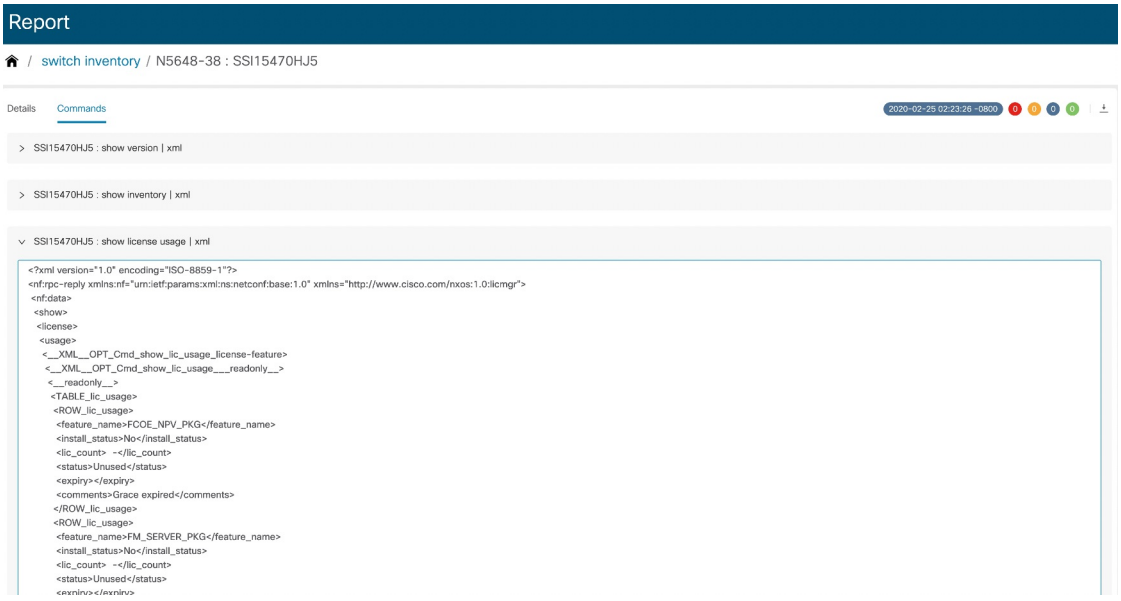
詳細表示には、サマリとともに完全なレポート JSON データが表示されます。レポートの詳細は、論理的にセクションにグループ化されます。各セクションは、折りたたみ可能なウィジェットで個別に表示されます。

サマリ表示と詳細表示の両方で、レポートで生成されたエラー、警告、情報、成功メッセージの数が表示されます。



コマンド ログ

コマンド ログには、コマンドの実行に使用された API に基づいて、レポートで実行されたすべてのコマンドが含まれます。



レポート Python ライブラリ

レポートインフラストラクチャは、レポートJSONモデルを生成するための使いやすく軽量なPython ライブラリを提供します。この API を使用するには、テンプレートに次のインポートステートメントを追加する必要があります。

```
from reportlib.preport import Report
```

レポート API

レポート オブジェクトの作成

すべてのレポートは、最初のステップとして「レポート」オブジェクトを作成する必要があります。

```
report = Report (“Report title”)
```

サマリの追加

すべてのレポートには1つのサマリを含めることができ、それはPythonディクショナリです。サマリは次のように追加できます。

```
summary = report.add_summary()
```

サマリへのコンテンツの追加

キーと値

```
summary [‘NXOS Version’] = ‘8.1(0)’
```

メッセージ – 推論

```
summary.add_message (“Simple message”)
```

（注） DCNM 11.4(1) では、DCNM はサマリの値として JSON オブジェクトをサポートしていません。次の例はサポートされていません。

```
summary[“info”] = {“key”:”value”,”key-2”:”value-2”}
```

サマリのテーブル

```
table = summary.add_table(title,_id)
```

- title：テーブルのタイトル
- _id：テーブルの一意の識別子

テーブルへの行の追加

```
table.append(value, _id)
```

- value：JSON オブジェクトです。ネストされた JSON はサポートされません。

- _id : テーブルの一意の識別子

例 :

```
table.append({'column1': 'value1','column2':'value2'},” FOX1816G0S9”)
```

セクションの追加

セクションは、レポートの内容を論理的にグループ化したものです。ユーザの判断により、これらのセクションを作成し、表示する情報を追加します。

セクションは次のように追加できます。

```
section = report.add_section (“Section title”,_id)
```

- _id : テーブルの一意の識別子
- セクション : ディクショナリです

セクションへのコンテンツの追加

キーと値

以下に示すように、単純なキーと値のペアをセクションに追加できます。

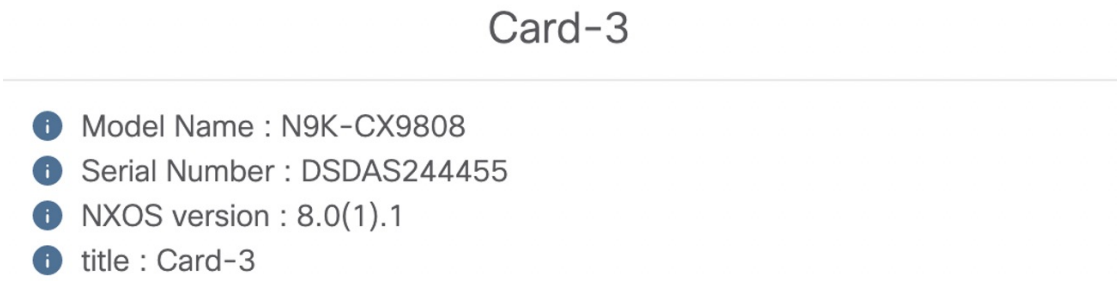
```
section[‘key’] = ‘value’
```

JSON ドキュメント – カード

任意のキーと値のペアと同じように、単一の JSON ドキュメントを追加できます。ネストされた JSON は 11.4(1) ではサポートされていません

```
section[‘key’] = {‘key’:’value’,’key-2’:’value’}
```

JSON ドキュメントは、次のようにカード ウィジェットに表示されます。



JSON ドキュメントの配列 – テーブル

section.append API を使用すると、ユーザはテーブルを作成し、行を追加できます（次の制限が付きます）。

1. すべての JSON ドキュメントには同じキーのセットが必要です
2. ネストされた JSON はサポートされません

```
section.append(key, dictionary, _id)
```

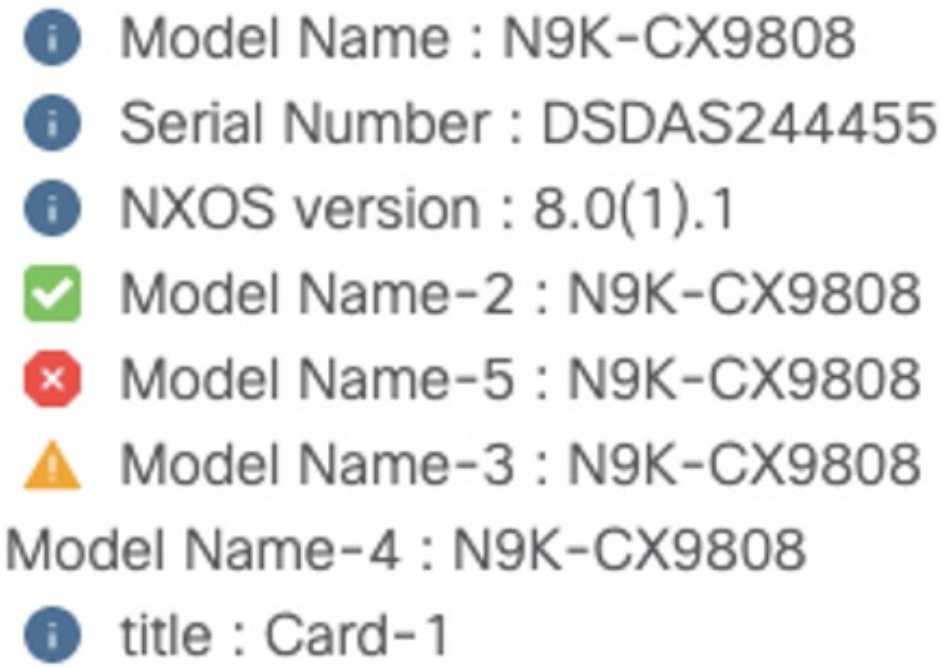
_id：テーブルの行を一意に識別する一意の識別子。_idが重複すると、一意のid違反エラーが発生します。

例：

```
section.append('Switch Details', {'name': 'N5K'},'DSDAS244455')
section.append('Switch Details', {'name': 'N6K'}, 'CSDAS244456')
section.append('Switch Details', {'name': 'N7K'}, 'ASDAS244457')
```

Formatters

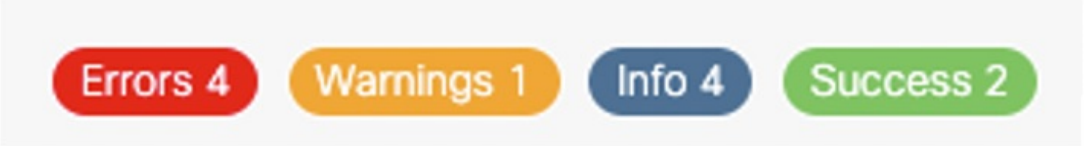
Formatter を使用すると、UI に表示される値に追加のフォーマットを追加できます。



示されているように、値を次のようにマークできます。

1. エラー
2. 成功
3. 警告
4. INFO

これらのマーカーをレポートに追加すると、対応するカウント エラー、警告、成功、情報が UI に表示されるように自動的に更新されます。




```
Formatter.add_marker(value,marker)
```

- value：マーカーを追加する値。
- マーカー：Marker.ERROR,Marker.SUCCESS,Marker.WARNING,Marker.INFO

例：

```
Formatter.add_marker (“NXOS version”,Marker.INFO)
```

グラフ

レポートは、サマリとセクションの両方でのグラフの追加をサポートしています。

サマリへのチャートの追加

```
report = Report(“title”)
summary = report.add_summary()
summary.add_chart(ChartType, _id)
```

- **ChartType**: ChartTypes.COLUMN_CHART, ChartTypes.PIE_CHART, ChartTypes.LINE_CHART.
- **_id**：チャートの一意の ID

セクションへのグラフの追加

```
report = Report(“title”)
section = report.add_section(“Section title”,_id)
section.add_chart(ChartType, _id)
```

- **ChartType**: ChartTypes.COLUMN_CHART, ChartTypes.PIE_CHART, ChartTypes.LINE_CHART
- **_id**：チャートの一意の ID

円グラフ

設定とサブタイトルのタイトル

```
pie_chart.set_title("Chart title")
pie_chart.set_subtitle(“Sub title”)
```

付加価値を加えてください。

```
pie_chart.add_value("key",value)
```

- キー：文字列キー
- 値：数字の値

縦棒グラフ

設定とサブタイトルのタイトル

```
column_chart.set_title("Chart title")
column_chart.set_subtitle("Sub title")
```

X 軸と Y 軸のタイトルを設定する

```
column_chart.set_xAxis_title("X-Axis title")
column_chart.set_yAxis_title("y-Axis title")
```

値の追加

```
bar_chart.add_value("key",value,category)
```

- キー：文字列キー
- 値：数字の値
- カテゴリ：棒グラフは、データを「カテゴリ」と呼ばれる論理グループに分割します。指定されたキーには、各カテゴリの値が必要です。

たとえば、デバイス数がキーで、ファブリック名がカテゴリです。チャートには、ファブリックごと、つまりカテゴリごとにデバイス数が含まれている必要があります。

折れ線グラフ

設定とサブタイトルのタイトル

```
line_chart.set_title("Chart title")
line_chart.set_subtitle("Sub title")
```

X 軸と Y 軸のタイトルを設定する

```
line_chart.set_xAxis_title("X-Axis title")
line_chart.set_yAxis_title("y-Axis title")
```

値の追加

```
line_chart.add_value("key",value,category)
```

- キー：文字列キー
- 値：数字の値
- カテゴリ：折れ線グラフは、データをカテゴリと呼ばれる論理グループに分割します。指定されたキーには、各カテゴリの値が必要です。

たとえば、デバイス数がキーで、ファブリック名がカテゴリです。チャートには、ファブリックごと、つまりカテゴリごとにデバイス数が含まれている必要があります。

デバイスでの CLI の実行

コマンドの表示

```
from reportlib.preport import show
cli_responses = show (serial_number ,*commands)
```

- serial_number：コマンドを実行するデバイスのシリアル番号。VDC の場合、シリアル番号は serial_number:vdc_name である必要があります。シリアル番号のリストを渡して、複数のデバイスで同じコマンド セットを実行できます。
- *commands：デバイスで実行するコマンド。可変引数です。複数のコマンドを指定できます。

例：

- 単一のスイッチでコマンドを実行する：

```
cli_responses = show(“FOX1816G0S9”,'show version | xml', 'show inventory | xml',
'show license usage | xml')
```

- 複数のスイッチでコマンドを実行する：

```
cli_responses = show( [“FOX1816G0S9”,”SSI15470HJ5”],'show version | xml', 'show
inventory | xml', 'show license usage | xml')
```

コマンドの表示と応答の保存

```
from reportlib.preport import show_and_store
cli_responses = show_and_store(report,serial_number,*commands)
```

report：レポート オブジェクトが作成されました。

serial_number：コマンドを実行するデバイスのシリアル番号。VDC の場合、シリアル番号は serial_number:vdc_name である必要があります。シリアル番号のリストを渡して、複数のデバイスで同じコマンド セットを実行できます。

*commands：デバイスで実行するコマンド。可変引数です。複数のコマンドを指定できます。

例：

- 単一のスイッチでコマンドを実行する：

```
cli_responses = show_and_store(report, “FOX1816G0S9”, 'show version | xml', 'show
inventory | xml', 'show license usage | xml')
```

- 複数のスイッチでコマンドを実行する：

```
cli_responses = show_and_store(report, [“FOX1816G0S9”,”SSI15470HJ5”], 'show version
 | xml', 'show inventory | xml', 'show license usage | xml')
```

注意：この API は、デバイスからの応答をレポートとともに elasticsearch に保存します。すべての応答を保存するとストレージが大幅に増加する可能性があるため、ユーザはこの API を使用する際に注意する必要があります。

戻り値

戻り値 API は応答のリストを返し、各応答は次の構造を持つディクショナリです。

```
{
 'status': 'success' | 'failed,
 'response':<response from device>,
 'command':<cli command>,
 'serial_number': <device serial number>
}
```

複数のスイッチの場合、応答はスイッチごとのエントリを含む応答のリストです。

```
[
{
 'status': 'success',
 'response':<response from device>,
 'command':'show version',
 'serial_number': 'FOX1816G0S9'
},
{
 'status': 'success',
 'response':<response from device>,
 'command':'show version',
 'serial_number': 'SSI15470HJ5
}
]
```

ジョブ コンテキスト情報の取得

APP からジョブをスケジュールしているときに選択された繰り返しを取得します

```
get_recurrence(context)
```

この API は、ジョブの作成中に選択された繰り返しを返します。戻り値は、NOW、ONCE、DAILY、WEEKLY、MONTHLY、ONDEMAND、および PERIODIC です。

get_period

ジョブが定期的にスケジュールされている場合、API を使用して期間情報にアクセスできます。

```
period = get_period(context)
period.get_period() will return the period
period.get_time_unit() will return time Unit (HOURS, MINUTES)
```

履歴レポートの分析

以前に生成されたレポートの取得

「get_previous_reports()」メソッドを使用すると、過去に生成されたレポートを取得できます。これは、現在のデータと履歴データに基づいて分析を実行するために使用できます。この API は、作成された時間の降順でレポートを返します。

```
List of reports = get_previous_reports (context,entity,count)
```

この API は、レポートのリストを返します。

コンテキスト：generateReport(context) メソッドから入力として受け取ったオブジェクト

エンティティ：serial_number またはファブリック名

カウント：取得するレポートの数

最も古いレポートを取得

```
oldest_report = get_oldest_report(context,entity)
```

コンテキスト：generateReport(context) メソッドから入力として受け取ったオブジェクト

エンティティ：serial_number またはファブリック名

上記の API はどちらも、情報を取得するために次の API を使用して Report オブジェクトを返します。

1. **概要を取得する**：report.get_summary()
2. **セクションの取得**：report.get_section(_id)

```
report.get_section(_id)
```

_id：セクションの一意の識別子

XML ユーティリティ

XML ツリーを取得

```
from reportlib.preport import getxmltree
xml_element_tree = getxmltree(xml_string,tag)
```

この API は、指定されたタグの下にある XML ツリーを返します。

xml_string：デバイスからの XML 応答。

タグ：XML タグ。このタグの下の完全な XML は、ElementTree として返されます。

xml_element_tree：この API は xml.etree.ElementTree オブジェクトを返します。

XML 行を取得する

CLI 応答に行がある場合、getxmlrows API を使用して行の配列を取得できます。

```
from reportlib.preport import getxmlrows
rows = getxmlrows(xml_tree,tag_xpath)
```

xml_tree：xml.etree.ElementTree オブジェクト

tag_xpath：XML レコードの xpath。詳細については、https://docs.python.org/2/library/xml.etree.elementtree.html#xpath-support を参照してください。

行：行の配列

ノード値を取得

XML ノード値は、**getnodevalue** API を使用して読み取ることができます。この API は、プリミティブ型のノード値を取得するために使用する必要があります。

```
from reportlib.preport import getnodevalue
value = getnodevalue(xml_tree,node_xpath)
```

ノードが存在するかどうかを確認する

```
from reportlib.preport import has_tag
has_tag(xml_tree,tag)
```

この API は、指定されたタグが XML ツリーに存在するかどうかに基づいて true または false を返します。

WrapperResp

すべてのレポートは、WrapperResp タイプのオブジェクトを返す必要があります。

WrapperResp は次のようにインスタンス化できます。

```
respObj = WrappersResp.getRespObj()
```

WrapperResp のリターン コードは、レポートが正常に実行されたかどうかを示します。

1. すべてのコマンドが実行され、必要な情報が抽出された場合、レポートは成功 **respObj.setSuccessRetCode()** を返します。
2. コマンドの失敗などの例外が発生した場合、レポートは失敗 **respObj.setFailureRetCode()** を返します。
3. エラーの場合、エラーの理由を **respObj.addErrorReport(template_name,error_message)** として追加できます。

Report セクションで作成されたレポート オブジェクトは、次のように WrappersResp の値に設定する必要があります。

```
respObj.setValue(report)
```

Logger

Logger を使用すると、レポート テンプレートからメッセージをログに記録できます。Logger を使用して記録されたすべての情報
は、`/usr/local/cisco/dcm/fm/logs/preport_jython.log`に記録されます。

```
Logger.info(“message”)
Logger.debug(“message”)
Logger.error(“message”)
Logger.trace(“message”)
Logger.warn(“message”)
```

翻訳について
このドキュメントは、米国シスコ発行ドキュメントの参考和訳です。リンク情報につきましては、日本語版掲載時点で、英語版にアップデートがあり、リンク先のページが移動/変更されている場合がありますことをご了承ください。あくまでも参考和訳となりますので、正式な内容については米国サイトのドキュメントを参照ください。