MarkLogic サーバー

Node.js Application Developer's Guide

MarkLogic 9
2017 年 5 月

最終更新：9.0-2、2017 年 7 月

目次

Node.js Application Developer’s Guide

1.0 Node.js クライアント API の概要

Node.js クライアント API を使用すると、MarkLogic データベース内のドキュメントやセマンティックデータの読み取り、書き込み、クエリを行う Node.js アプリケーションを作成できます。

Node.js API は、GitHub で管理されているオープンソースプロジェクトです。ソースにアクセスしたり、問題を報告またはレビューしたり、プロジェクトに参加したりするには、http://github.com/marklogic/node-client-api にアクセスします。

1.1 はじめに

このセクションでは、ドキュメントをデータベースにロードする方法、ドキュメントをクエリする方法、ドキュメントの一部を更新する方法、ドキュメントをデータベースから読み取る方法について示します。これらの基本的な機能を使うと、ここで紹介した以外にもさまざまなことができます。このセクションの最後に、Node.js クライアント API の理解を深めるのに役立つリソースを示します。

開始する前に、「必須のソフトウェア」（15 ページ）に示されているソフトウェアがインストールされていることを確認してください。また、`node` および `npm` コマンドに対してパスが通っていなければなりません。

> **注：** Microsoft Windows を使用している場合は、Cygwin シェルではなく、DOS コマンドシェルを使用してください。Cygwin では、`node` および `npm` の環境はサポートされていません。

次の手順では、Node.js クライアント API をインストールし、いくつかの単純な JSON ドキュメントをデータベースにロードして、ドキュメントを検索および変更する方法について説明します。

1. MarkLogic をまだ準備していない場合は、http://developer.marklogic.com にアクセスして MarkLogic サーバーをダウンロードし、インストールして起動します。

2. この説明に従って例を実行するプロジェクトディレクトリを作成または選択します。以降の説明は、そのディレクトリに移動した状態であることを前提として話を進めます。

3. パブリック npm リポジトリから最新バージョンの Node.js クライアント API をダウンロードして、プロジェクトディレクトリにインストールします。次に例を示します。

```
npm install marklogic
```

4. MarkLogic 接続情報を設定します。次のコードを `my-connection.js` という名前のファイルにコピーします。MarkLogic サーバーの接続情報は、自身の環境に合わせて変更します。少なくとも `user` と `password` の値を変更する必要があります。また少なくとも `rest-reader` ロールと `rest-writer` ロールまたは同等の権限を持っている MarkLogic ユーザーを選択してください。詳細については、「セキュリティ要件」（15 ページ）を参照してください。

```
module.exports = {
  connInfo: {
    host: 'localhost',
    port: 8000,
    user: 'user',
    password: 'password'
  }
};
```

このガイドの以降の例では、この接続設定モジュールがパス `./my-connection.js` で存在していることを想定しています。

5. ドキュメント例をデータベースにロードします。次のスクリプトをファイルにコピーして、`node` コマンドを使用して実行します。複数の JSON ドキュメントが、`DatabaseClient.documents.write` を使用してデータベースに挿入されます。

```
// Load documents into the database.

var marklogic = require('marklogic');
```

```
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

// Document descriptors to pass to write().
var documents = [
  { uri: '/gs/aardvark.json',
    content: {
      name: 'aardvark',
      kind: 'mammal',
      desc: 'The aardvark is a medium-sized burrowing,
nocturnal mammal.'
    }
  },
  { uri: '/gs/bluebird.json',
    content: {
      name: 'bluebird',
      kind: 'bird',
      desc: 'The bluebird is a medium-sized, mostly
insectivorous bird.'
    }
  },
  { uri: '/gs/cobra.json',
    content: {
      name: 'cobra',
      kind: 'mammal',
      desc: 'The cobra is a venomous, hooded snake of the
family Elapidae.'
    }
  },
];

// Load the example documents into the database
db.documents.write(documents).result(
  function(response) {
    console.log('Loaded the following documents:');
    response.documents.forEach( function(document) {
      console.log('  ' + document.uri);
    });
  },
  function(error) {
    console.log(JSON.stringify(error, null, 2));
  }
);
```

次のような出力が表示されます。

```
Loaded the following documents:
  /gs/aardvark.json
  /gs/bluebird.json
  /gs/cobra.json
```

6. データベースを検索します。次のスクリプトをファイルにコピーして、node コマンドを使用して実行します。このスクリプトは、JSON プロパティ kind の値が 'mammal' のドキュメントをデータベースから取得します。

```
// Search for documents about mammals, using Query By
Example.
// The query returns an array of document descriptors,
one per
// matching document.The descriptor includes the URI and
the
// contents of each document.

var marklogic = require('marklogic');
var my = require('./my-connection.js');

var db = marklogic.createDatabaseClient(my.connInfo);
var qb = marklogic.queryBuilder;

db.documents.query(
  qb.where(qb.byExample({kind: 'mammal'}))
).result( function(documents) {
    console.log('Matches for kind=mammal:')
    documents.forEach( function(document) {
      console.log('\nURI: ' + document.uri);
      console.log('Name: ' + document.content.name);
    });
}, function(error) {
    console.log(JSON.stringify(error, null, 2));
});
```

次のような出力が表示されます。間違って cobra（コブラ）が mammal（哺乳類）としてラベル付けされている点に注目してください。次の手順で、コンテンツ内のこのエラーを修正します。

```
Matches for kind=mammal:

URI: /gs/cobra.json
Name: cobra

URI: /gs/aardvark.json
Name: aardvark
```

7. ドキュメントにパッチを適用します。前の手順では、間違って cobra（コブラ）が mammal（哺乳類）としてラベル付けされていました。この手順では、`/gs/cobra.json` の `kind` プロパティを `'mammal'` から `'reptile'`（爬虫類）に変更します。次のスクリプトをファイルにコピーして、`node` コマンドを使用して実行します。

```
// Use the patch feature to update just a portion of a
document,
// rather than replacing the entire contents.

var marklogic = require('marklogic');
var my = require('./my-connection.js');

var db = marklogic.createDatabaseClient(my.connInfo);
var pb = marklogic.patchBuilder;

db.documents.patch(
  '/gs/cobra.json',
  pb.replace('/kind', 'reptile')
).result( function(response) {
    console.log('Patched ' + response.uri);
}, function(error) {
    console.log(JSON.stringify(error, null, 2));
});
```

次のような出力が表示されます。

```
Patched /gs/cobra.json
```

8. 検索を再度実行するか、URI でドキュメントを取得して変更を確認します。URI で /gs/cobra.json を取得するには、次のスクリプトをファイルにコピーし、node コマンドを使用して実行します。

```
// Read documents from the database by URI.

var marklogic = require('marklogic');
var my = require('./my-connection.js');

var db = marklogic.createDatabaseClient(my.connInfo);

db.documents.read(
  '/gs/cobra.json'
).result( function(documents) {
  documents.forEach( function(document) {
    console.log(JSON.stringify(document, null, 2) + '\n');
  });
}, function(error) {
    console.log(JSON.stringify(error, null, 2));
});
```

次のような出力が表示されます。

```
{
  "uri": "/gs/cobra.json",
  "category": "content",
  "format": "json",
  "contentType": "application/json",
  "contentLength": "106",
  "content": {
    "name": "cobra",
    "kind": "reptile",
    "desc": "The cobra is a venomous, hooded snake of the
family Elapidae."
  }
}
```

9. 必要に応じてこのドキュメント例を削除します。次のスクリプトをファイルにコピーして、node コマンドを使用して実行します。ドキュメントが削除されたことを確認するには、手順 8 のスクリプトを再度実行します。

```
// Remove the example documents from the database.
// This example removes all the documents in the directory
// /gs/.You can also remove documents by document URI.
```

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');

var db = marklogic.createDatabaseClient(my.connInfo);

db.documents.removeAll(
  {directory: '/gs/'}
).result( function(response) {
  console.log(response);
});
```

次のような出力が表示されます。

```
{ exists: false, directory: '/gs/' }
```

ドキュメントの削除は、冪等（べきとう）操作（何回繰り返しても、1 回だけ実行したときと同じ結果になる操作）です。スクリプトを再度実行しても同じ出力が生成されます。

API の理解をさらに深めるには、次のリソースを参照してください。

目的	参照先
さらに多くの例について調べる	API とともに配布されている例とテスト。ソースは http://github.com/marklogic/node-client-api から入手できます。また、API のインストール後は `node_modules/marklogic` ディレクトリから使用できます。
ドキュメントとメタデータの読み取りおよび書き込みについて学ぶ	「ドキュメントの操作」（35 ページ）
ドキュメントの検索と、レキシコンおよびインデックスのクエリについて学ぶ	「ドキュメントとメタデータのクエリ」（131 ページ） Search Developer's Guide
コンテンツ変換機能やリソースサービス拡張機能などの拡張ポイントについて学ぶ	「拡張機能、変換機能、サーバーサイドコードの実行」（234 ページ）

目的	参照先
下位レベルの API ドキュメントについて調べる	Node.js API リファレンス API リファレンスのローカルコピーを生成することもできます。詳細については、次の GitHub のプロジェクトページを参照してください： http://github.com/marklogic/node-client-api

1.2 必須のソフトウェア

Node.js クライアント API を使用するには、次のソフトウェアが必要です。

- MarkLogic 8 以降。バージョン 2.0.x の Node.js クライアント API の機能は、MarkLogic 9 以降でのみ使用できます。
- Node.js、バージョン 6.3.1 以降。Node.js は、http://nodejs.org から入手できます。
- Node.js Package Manager ツール（npm）。サポートされている Node.js バージョンと互換性がある最新バージョンを推奨します。
- 認証に Kerberos を使用することを予定している場合は、MIT Kerberos ソフトウェアが必要です。詳細については、「Kerberos 認証の使用」（31 ページ）を参照してください。

このガイドの例は、`node` および `npm` コマンドに対してパスが通っていることを想定しています。

1.3 セキュリティ要件

ここでは、Node.js クライアント API が使用する基本的なセキュリティモデルと、それを変更または拡張する必要がある一般的な状況について説明します。ここでは、次の内容を取り上げます。

- 基本的なセキュリティ要件
- ドキュメントアクセスの制御
- 異なるデータベースに対するリクエストの評価
- サーバーサイドコードの評価と呼び出し
- CombinedQueryDefinition の使用

1.3.1 基本的なセキュリティ要件

`DatabaseClient` オブジェクトを作成するときに指定するユーザーは、ターゲットデータベース内のドキュメントの読み取りまたは更新パーミッションなど、実行する操作によってアクセスされるコンテンツに対して適切な URI 権限を持っている必要があります。

Node.js クライアントは、MarkLogic サーバーと通信するのに MarkLogic REST クライアント API を使用するため、同じセキュリティモデルを使用します。適切な URI 権限に加え、ユーザーは次に示す事前定義されたロールまたは同等の権限が必要です。表内の各ロールの機能は、その下のロールに含まれます。

ロール	説明
`rest-extension-user`	リソースサービス拡張メソッドへのアクセスを有効にします。このロールは、他の事前定義された REST API ロールでは暗黙ですが、カスタムロールを定義するときには明示的に含める必要があります。
`rest-reader`	ドキュメントおよびメタデータの取得といった、読み取り操作を実行できます。このロールにはその他の権限がないため、コンテンツを読み取るために追加の権限が必要になることもあります。
`rest-writer`	ドキュメント、メタデータ、または設定情報の作成といった、書き込み操作を実行できます。このロールにはその他の権限がないため、コンテンツを書き込むために追加の権限が必要になることもあります。
`rest-admin`	インスタンスの作成およびインスタンス設定の管理など、管理操作を実行できます。このロールにはその他の権限がないため、追加の権限が必要になることもあります。

REST インスタンスに関連付けられているデフォルトデータベース以外のデータベースを使用する場合や、`DatabaseClient` の `eval` メソッドまたは `invoke` メソッドを使用する場合など、一部の操作には追加の権限が必要です。このような要件については、以降で詳しく説明します。

1.3.2 ドキュメントアクセスの制御

Node.js クライアント API のデフォルトロールを使用して作成するドキュメントは、`rest-reader` ロールの読み取りパーミッションと `rest-writer` ロールの更新パーミッションを持ちます。デフォルトでは、`rest-reader` ロールを持っているユーザーは `rest-reader` として作成されたすべてのドキュメントを読み取ることができ、`rest-writer` ロールを持っているユーザーは `rest-writer` として作成されたすべてのドキュメントに書き込むことができます。この動作は、ドキュメントパーミッションやカスタムロールを使用して無効にできます。

特定のユーザーに対するアクセスを制限するには、ユーザーをデフォルトの `rest-*` ロールに割り当てるのではなく、カスタムロールを作成します。例えば、カスタムロールを使用して、あるグループのユーザーが、別のグループが作成したドキュメントを表示できないようにできます。

詳細については、『REST Application Developer's Guide』の「Controlling Access to Documents and Other Artifacts」を参照してください。

1.3.3 異なるデータベースに対するリクエストの評価

`DatabaseClient` を作成して MarkLogic サーバーインスタンスに接続すると、接続先の REST インスタンスにはデフォルトのコンテンツデータベースが関連付けられています。`DatabaseClient` を作成するときに別のデータベースを指定することもできますが、別のデータベースに対して操作を実行するには `http://marklogic.com/xdmp/privileges/xdmp-eval-in` 権限または同等の権限が必要です。

アプリケーションで別のデータベースを使用できるようにするには、次の手順を実行します。

1. `rest-*` ロールの適切な組み合わせに加え、`xdmp:eval-in` 実行権限を持つロールを作成します（権限を既存のロールに追加することもできます）。
2. 手順 1 のロールをユーザーに割り当てます。
3. 手順 2 のユーザーで `DatabaseClient` を作成します。

この処理を簡単に実行する方法の 1 つとして、事前定義された `rest-*` ロールの 1 つを継承して、`eval-in` 権限を追加する方法が挙げられます。

ロールと権限の詳細については、『Security Guide』を参照してください。REST API インスタンスの管理の詳細については、「REST API インスタンスの管理」（281 ページ）を参照してください。

1.3.4 サーバーサイドコードの評価と呼び出し

MarkLogic サーバーでは、`DatabaseClient.eval` および `DatabaseClient.invoke` 操作を使用して任意のコードを評価できます。これらの操作では、`rest-reader` や `rest-writer` などの通常の REST API ロールではなく（またはそれに加えて）特別な権限が必要です。

詳細については、「必要な権限」（266 ページ）を参照してください。

1.3.5 CombinedQueryDefinition の使用

`queryBuilder` や `valuesBuilder` などのビルダーインターフェイスを使用するのではなく、`CombinedQueryDefinition` を使用してクエリを作成する場合、クエリを評価するために `rest-admin` ロールまたは同等の権限を必要とするクエリオプションを指定することが可能です。

詳細については、『REST Application Developer's Guide』の「Using Dynamically Defined Query Options」を参照してください。

1.4 用語と定義

このガイドでは、次の用語と定義を使用します。

用語	定義
REST クライアント API	RESTful な HTTP リクエストを使用して MarkLogic と通信するアプリケーションを開発するための MarkLogic API。Node.js クライアント API は、REST クライアント API の上位に構築されています。
REST API インスタンス	REST クライアント API のリクエストを処理するように特別に設定された MarkLogic HTTP アプリケーションサーバー。Node.js クライアント API は REST API インスタンスを必要とします。このインスタンスは、MarkLogic のインストール後にポート 8000 で使用できるようになります。詳細については、「REST API インスタンスとは」（281 ページ）を参照してください。
npm	Node.js パッケージマネージャ。npm は、Node.js クライアント API とその依存関係をダウンロードしてインストールする目的で使用します。

用語	定義
ビルダー	クエリ（`marklogic.queryBuilder`）およびドキュメントパッチ（`marklogic.patchBuilder`）などの潜在的に複雑なデータ構造を構築するための関数を公開する、Node.js クライアント API 内のインターフェイス。
Promise	Promise は、非同期イベントの結果とやり取りするための JavaScript インターフェイス。詳細については、「Promise 結果処理パターン」（23 ページ）を参照してください。
MarkLogic モジュール	Node.js クライアント API をカプセル化するモジュール。`require()` を使用してアプリケーションにモジュールを含めます。詳細については、「MarkLogic 名前空間」（20 ページ）を参照してください。
ドキュメントディスクリプタ	ドキュメントコンテンツとメタデータを名前付きの JavaScript オブジェクトプロパティとしてカプセル化するオブジェクト。詳細については、「ドキュメントディスクリプタ」（21 ページ）を参照してください。
データベースクライアント	REST API インスタンスを通じて MarkLogic サーバーへの接続をカプセル化する特別なオブジェクト。ほぼすべての Node.js クライアント API 操作が、データベースクライアントオブジェクトを通じて実行されます。詳細については、「データベースクライアントの作成」（26 ページ）を参照してください。
git	ソースコントロール管理システム。Node.js クライアント API ソースをチェックアウトして使用するには、git クライアントが必要です。
GitHub	Node.js クライアント API プロジェクトをホストするオープンソースプロジェクトリポジトリ。詳細については、http://github.com/ を参照してください。

1.5 主な概念と規則

- MarkLogic 名前空間
- パラメータの受け渡し規則
- ドキュメントディスクリプタ
- サポートされている結果処理手法
- Promise 結果処理パターン

- ストリーム結果処理パターン
- データベースへのストリーミング
- エラー処理

1.5.1 MarkLogic 名前空間

Node.js クライアント API ライブラリは、データベースクライアントファクトリメソッドに加えて、`queryBuilder`（検索）、`valuesBuilder`（値クエリ）、`patchBuilder`（ドキュメントの部分的な更新）などのビルダーへのアクセスを提供する、名前空間をエクスポートします。

MarkLogic モジュールをコードに含めるには、`require()` を使用して結果を変数にバインドします。例えば、独自の Node.js プロジェクトにおいてモジュールにインストールした場合は、これに「marklogic」という名前を付けて含めることができます。

```
var ml = require('marklogic');
```

任意の変数名を使用できますが、このガイドの例では `ml` であることを想定しています。

1.5.2 パラメータの受け渡し規則

多くの入力パラメータ値を必要とする Node.js クライアント API 関数は、そのような値を呼び出しオブジェクトの名前付きプロパティとして受け付けます。例えば、`createDatabaseClient()` メソッドを呼び出すときは、ホスト名、ポート、データベース名、およびその他のいくつかの接続プロパティを指定できます。この際、これらの値を次のような単一のオブジェクトにカプセル化します。

```
ml.createDatabaseClient({host: 'some-host', port: 8003,
...});
```

パラメータ値には 1 つあるいは複数の値を指定でき、プロパティ値には 1 つの値または値の配列を指定できます。関数によっては、配列またはリストのいずれかをサポートしています。次に例を示します。

```
db.documents.write(docDescriptor)
db.documents.write([docDescriptor1, docDescriptor2, ...])
db.documents.write(docDescriptor1, docDescriptor2, ...)
```

関数に、他のパラメータなしで頻繁に使用されるパラメータがある場合は、呼び出しオブジェクトでカプセル化する代わりに、パラメータを直接渡すことができます。例えば、DatabaseClient.documents.remove は、複数のプロパティを指定可能な呼び出しオブジェクト、または単一の URI 文字列を受け付けます。

```
db.documents.remove('/my/doc.json')
db.documents.remove({uri: '/my/doc.json', txid: ...})
```

特定の操作の詳細については、Node.js API リファレンスを参照してください。

1.5.3 ドキュメントディスクリプタ

ドキュメントディスクリプタは、ドキュメントコンテンツとメタデータを名前付きの JavaScript オブジェクトプロパティとしてカプセル化するオブジェクトです。DatabaseClient.documents.read および DatabaseClient.documents.write などの Node.js クライアント API ドキュメント操作は、ドキュメントディスクリプタを受け付けたり返したりします。

通常、ドキュメントディスクリプタには、少なくともデータベース URI と、ドキュメントコンテンツとドキュメントメタデータのいずれかまたは両方を表すプロパティが含まれています。例えば次のコードは、URI が /doc/example.json のドキュメントのドキュメントディスクリプタです。ドキュメントは JSON ドキュメントであるため、そのコンテンツは JavaScript オブジェクトとして表すことができます。

```
{ uri : 'example.json', content : {some : 'data'} }
```

すべてのプロパティが常に存在しているわけではありません。例えば、ドキュメントのコンテンツのみを読み取った場合、結果のドキュメントディスクリプタにはメタデータ関連のプロパティは含まれません。同様に、コンテンツと collections メタデータプロパティのみを挿入した場合、入力ディスクリプタには permissions プロパティまたは quality プロパティは含まれません。

```
{ uri : 'example.json',
  content : {some : 'data'},
  collections : ['my-collection']
}
```

content プロパティには、オブジェクト、文字列、Buffer、または ReadableStream を指定できます。

プロパティ名の完全なリストについては、Node.js API リファレンスの「DocumentDescriptor」を参照してください。

1.5.4 サポートされている結果処理手法

Node.js クライアント API のほとんどの関数が、MarkLogic サーバーから返された結果を処理する次の方法をサポートしています。

- コールバック：`result` 関数を呼び出して、成功またはエラーコールバック関数を渡します。結果を同期する必要がない場合はこのパターンを使用します。次に例を示します。

  ```
  db.documents.read(...).result(function(response) {...})
  ```

- Promise：`result` 関数を呼び出し、Promise を通じて結果を処理します。データベースにドキュメントを書き込み、続けて検索を実行する場合など、インタラクションを連結する場合は Promise を使用します。要求したすべてのデータが MarkLogic サーバーから返されるまで、成功コールバックは呼び出されません。次に例を示します。

  ```
  db.documents.read(...).result().then(function(response)
  {...})...
  ```

  詳細については、「Promise 結果処理パターン」（23 ページ）を参照してください。

- オブジェクトモードストリーミング：`stream` 関数を呼び出し、`Readable` ストリームを通じて結果を処理します。ドキュメントやその他の項目を完全に受け取るたびに、このプログラムに制御が戻ります。JSON ドキュメントを読み取っている場合は、コールバックを呼び出す前に JavaScript オブジェクトに変換されます。次に例を示します。

  ```
  db.documents.read(...).stream().pipe(...)
  ```

  詳細については、「ストリーム結果処理パターン」（24 ページ）を参照してください。

- チャンクモードストリーミング：「chunked」引数を指定して `stream` 関数を呼び出し、`Readable` ストリームを通じて結果を処理します。ある一定のバイト数に到達するたびにこのプログラムに制御が戻ります。コールバックへの入力はバイトストリームです。

  ```
  db.documents.read(...).stream('chunked').pipe(...)
  ```

  詳細については、「ストリーム結果処理パターン」（24 ページ）を参照してください。

従来のコールバックまたは Promise パターンを使用した場合は、MarkLogic からすべての結果が返されるまで、プログラムに制御が戻りません。これは、少量のドキュメントを返す読み取り操作または書き込み操作など、大量のデータを返さない操作に適しています。ストリーミングでは結果を段階的に処理できるため、サイズの大きいファイルや大量のドキュメントを扱う場合はストリーミングのほうが適しています。

注： 成功したコールバックのユーザーコードのエラーは、次のエラーコールバックで処理されます。したがって、このようなエラーを処理するために catch 節を含める必要があります。詳細については、「エラー処理」（26 ページ）を参照してください。

1.5.5 Promise 結果処理パターン

Node.js クライアント API 関数は、`result()` メソッドを持つオブジェクトを返します。このオブジェクトは、Promise オブジェクトを返します。Promise は、非同期イベントの結果とやり取りするための JavaScript インターフェイスです。Promise には、`then`、`catch`、および `finally` の各メソッドが用意されています。詳細については、http://promisesaplus.com/ を参照してください。Promise は、連結して複数の操作を同期できます。

Promise の `then` メソッドに渡す成功コールバックは、MarkLogic とのインタラクションが完了し、すべての結果を受け取るまで呼び出されません。Promise パターンは、操作を同期するのに適しています。

例えば、次のようなシーケンスを使用してデータベースにドキュメントを挿入し、挿入の完了後にそのドキュメントをクエリして、クエリ結果を操作できます。

```
db.documents.write(...).result().then(
  function(response) {
    // search the documents after insertion
    return db.documents.query(...).result();
  }).then( function(documents) {
    // work with the documents matched by the query
  });
```

完全な例については、「例：マルチステートメントトランザクションでの Promise の使用」（232 ページ）を参照してください。

注： 成功したコールバック内のユーザーコードで発生したエラーを処理するために、catch 節を Promise チェーンに含める必要があります。詳細については、「エラー処理」（26 ページ）を参照してください。

Node.js クライアント API は、ストリームパターンによる結果の処理もサポートしています。大量のデータを処理する場合は、Promise よりもストリームのほうが適しています。詳細については、「ストリーム結果処理パターン」（24 ページ）を参照してください。

1.5.6 ストリーム結果処理パターン

Node.js クライアント API 関数は、`stream` メソッドを持つオブジェクトを返します。このオブジェクトは、MarkLogic からの結果に対して `Readable` ストリームを返します。ストリームでは、結果を段階的に処理できます。大量のドキュメントを読み取る場合、またはドキュメントのサイズが大きい場合は、ストリーミングの使用を検討してください。

ストリームは、MarkLogic サーバーから結果のデータを受け取りながらこれを処理できるため、大量のデータを処理する場合は Promise よりも少ないメモリオーバーヘッドで優れたスループットを提供します。

サポートされているのは、次の 2 つのストリームモードです。

- オブジェクトモード : ドキュメントやその他の項目を完全に受け取るたびにプログラムに制御が戻ります。インタラクションの単位は、ドキュメントディスクリプタです。JSON ドキュメントの場合、ディスクリプタ内のコンテンツは使いやすくするために JavaScript オブジェクトに変換されます。オブジェクトモードは、デフォルトのストリーミングモードです。
- チャンクモード : 一定のバイト数を受け取るたびに、プログラムに制御が戻ります。インタラクションの単位は、不透明型のバイトストリームです。チャンクモードは、値 `'chunked'` を stream メソッドに渡して有効にします。

結果の各部分をドキュメントまたはオブジェクトとして処理する必要がある場合は、オブジェクトモードが最適です。例えば、ドメインオブジェクトのコレクションをデータベース内に JSON ドキュメントとして保持し、その後アプリケーション内でそれらを JavaScript オブジェクトとしてリストアする場合などです。チャンクモードは、ラージバイナリファイルをデータベースから読み取って、それをファイルに保存する場合など、大量のデータを不透明型で処理するのに最適です。

次のコードスニペットは、オブジェクトモードでストリームを使用して、複数のドキュメントがデータベースから取得されたときにそれらを処理します。完全なドキュメントを受け取るたびに、ストリーム `on('data')` コールバックがドキュメントディスクリプタによって呼び出されます。また、すべてのドキュメントを受け取ると、`on('end')` コールバックが呼び出されます。

```
db.documents.read(uri1, uri2, uriN).stream()
  .on('data', function(document) {
    // process one document
  }).on('end', function() {
    //wrap it up
  }).on('error', function(error) {
    // handle errors
  });
```

次のコードスニペットは、ストリームをチャンクモードで使用し、pipe を使用してラージバイナリファイルをデータベースからファイルにストリーミングします。

```
var fs = require('fs');
var ostrm = fs.createWriteStream(outFilePath);

db.document.read(largeFileUri).stream('chunked').pipe
(ostrm);
```

大量のデータを処理しない場合は、Promise パターンのほうが便利です。詳細については、「Promise 結果処理パターン」（23 ページ）を参照してください。

1.5.7 データベースへのストリーミング

大きな入力データセットを処理する可能性があるほとんどの Node.js メソッドは、ReadableStream を使用してデータを渡す処理をサポートしています。例えば、DatabaseClient.documents.write に渡すドキュメントディスクリプタの content プロパティには、オブジェクト、文字列、Buffer、または Readable ストリームを指定できます。ファイルなどのソースからデータをストリーミングする場合、必要なものはこのインターフェイスだけです。

例えば次の呼び出しは、Readable ストリームを使用して、ファイルから画像をデータベースにストリーミングします。

```
db.documents.write({
  uri: '/my/image.png',
  contentType: 'image/png',
  content: fs.createReadStream(pathToImage)
})
```

ストリームをその場で設定したり、詳細に制御したりする必要がある場合は、documents および graphs インターフェイスの createWriteStream メソッドを使用できます。例えば、次に示すように DatabaseClient.documents.write の代わりに DatabaseClient.documents.createWriteStream を使用すれば、write の呼び出しを制御してドキュメントを集めることができます。

```
var ws = db.documents.createWriteStream({
  uri: '/my/data.json',
  contentType: 'application/json',
});
// Resulting doc contains {"key":"value"}
ws.write('"{key"', 'utf8');
ws.write(': "value"}', 'utf8');
ws.end();
```

writeable ストリームインターフェイスを使用して、ドキュメントとセマンティックグラフをロードできます。詳細については、Node.js API リファレンスの「documents.createWriteStream」および「graphs.createWriteStream」を参照してください。

1.5.8 エラー処理

コールバックまたは Promise パターンを使用するときに、成功したコールバックでのエラーは、次のエラーコールバックで処理されます。このようなエラーをトラップする場合、Promise チェーンの末尾に（またはコールバックパターンの場合は結果のハンドラの後に）catch 節を含める必要があります。呼び出しを try-catch ブロックで囲むだけでは、このようなエラーはトラップされません。

例えば、従来のコールバックパターンでは、DatabaseClient.documents.write を呼び出した場合、次のように catch で終える必要があります。onSuccess コールバックが例外をスローした場合、onError 関数が実行します。

```
db.documents.write(...)
  .result(function onSuccess(response) {...})
  .catch(function onError(err) {...});
```

同様に、Promise パターンを使用してリクエストを連結している場合、類似のハンドラでチェーンを終了させる必要があります。

```
db.documents.write(...).result()
  .then(function onSuccess1(response) {...})
  .then(function onSuccess2(response) {...})
  .catch(function onError(err) {...});
```

1.6 データベースクライアントの作成

アプリケーションによる MarkLogic サーバーとのすべてのインタラクションは、marklogic.DatabaseClient オブジェクトを通じて行われます。各データベースクライアントは、REST API インスタンスおよび特定のデータベースへの接続について、1 人のユーザーごとに接続を管理します。アプリケーションでは、異なる REST API インスタンスや異なるデータベースに接続したり、異なるユーザーとして接続したりする目的で、複数のデータベースクライアントを作成できます。

注： マルチステートメントトランザクションおよび複数のデータベースを使用する場合、マルチステートメントトランザクションの一部として操作を実行するデータベースコンテキストは、トランザクションが作成されたのと同じでなければなりません。同じ制限がマルチステートメントトランザクションのコミットまたはロールバックにも適用されます。

データベースクライアントを作成するには、接続の詳細を表すパラメータを指定して `marklogic.createDatabaseClient` を呼び出します。例えば、次のコードは、インスタンスに関連付けられたデフォルトデータベースおよびダイジェスト認証を使用して、デフォルトのホストおよびポート（`localhost:8000`）でリスンする REST API インスタンスにアタッチされたデータベースクライアントを作成します。接続は、ユーザー「me」、パスワード「mypwd」で認証されます。

```
const ml = require('marklogic');
const db = ml.createDatabaseClient({user:'me',
password:'mypwd'});
```

証明書ベースの認証も Kerberos も使用していない場合、接続の詳細にはユーザー名とパスワードを含める必要があります。追加のプロパティを含められます。次の表は、`createDatabaseClient` に渡す接続オブジェクトに含めることができる主要プロパティを示したものです。

プロパティ名	デフォルト値	説明
`host`	`localhost`	設定済みの REST API インスタンスを持つ MarkLogic サーバーホスト。
`port`	`8000`	REST API インスタンスがリスンするポート。
`database`	REST インスタンスに関連付けられているデフォルトデータベース	ドキュメントの操作およびクエリを実行する対象となるデータベース。REST API インスタンスに関連付けられているデフォルトデータベース以外のデータベースを指定する場合は、xdmp-eval-in 権限が必要です。詳細については、「異なるデータベースに対するリクエストの評価」（17 ページ）を参照してください。
`authType`	`digest`	接続を確立するのに使用する認証方法。使用可能な値は、`basic`、`digest`、`digestbasic`、`application-level`、または `kerberos-ticket` です。これは、REST API インスタンスで設定した認証方法と一致している必要があります。詳細については、『Security Guide』を参照してください。

プロパティ名	デフォルト値	説明
ssl	false	SSL 接続を確立するかどうか。詳細については、『Security Guide』の「Configuring SSL on App Servers」を参照してください。true に設定した場合は、接続オブジェクトに追加の SSL プロパティを含めることができます。また、そのようなプロパティはエージェントに渡されます。そのようなプロパティのリストについては、http://nodejs.org/api/https.html#https_https_request_options_callback を参照してください。
agent	最大 10 個の空きソケット、合計 50 個のソケットを 60 秒ごとにキープアライブ	接続プールエージェント。

詳細については、Node.js API リファレンスの「marklogic.createDatabaseClient」および「REST API インスタンスの管理」（281 ページ）を参照してください。

1.7 認証および接続のセキュリティ

このセクションでは、データベースクライアントの作成および MarkLogic の接続の確立時に認証および SSL を設定する方法について説明します。このセクションでは、次の内容を取り上げます。

- SSL による MarkLogic への接続
- 証明書ベースの認証の使用
- Kerberos 認証の使用

1.7.1 SSL による MarkLogic への接続

データベースクライアントオブジェクトで MarkLogic への安全な接続を使用するために、ssl プロパティをいずれかの認証メソッドと組み合わせることができます。次に例を示します。

```
ml.createDatabaseClient({
  user: 'me',
  password: 'mypassword',
  authType: 'digest',
  ssl: true
})
```

アプリケーションサーバーは SSL を有効にしている必要があります。詳細については、『Security Guide』の「Configuring SSL on App Servers」を参照してください。

Node.js クライアント API は、最初に接続を確立するときに、MarkLogic が正当な証明書を送信していることを確認できる必要があります。証明書が、VeriSign などの実績ある認証局以外の認証局によって署名されている場合、データベースクライアント設定にその認証局の証明書を含める必要があります。認証局を指定するには、ca プロパティを使用します。次に例を示します。

```
ml.createDatabaseClient({
  authType: 'certificate',
  ssl: true
  ca: fs.readFileSync('ca.crt')
})
```

詳細については、『Security Guide』の「Procedures for Obtaining a Signed Certificate」を参照してください。

1.7.2 証明書ベースの認証の使用

証明書ベースの認証を使用するときに、クライアントアプリケーションは、認証局により署名された証明書を、証明書のプライベートキーとともに取得します。証明書には、パブリックキーと、接続を確立するために必要な他の情報が含まれます。

詳細については、次のトピックを参照してください。

- クライアント証明書の取得
- アプリケーションサーバーの設定
- 例：データベースクライアント設定

注： SSL を使用して MarkLogic に接続するときには、Node.js クライアント API で証明書ベースの認証だけを使用できます。詳細については、「SSL による MarkLogic への接続」（28 ページ）を参照してください。

1.7.2.1 クライアント証明書の取得

実績ある認証局によって署名されたクライアント証明書か、自己署名証明書のどちらかを使用できます。次のいずれかのオプションを選択します。

- Verisign などの実績ある認証局からクライアント証明書を取得します。
- 自己証明書を作成します。

実績ある認証局により署名されたクライアント証明書を取得するには、OpenSSL ソフトウェアまたは同様のツールを使用して CSR（証明書署名リクエスト）を作成し、CSR を認証局に送信します。詳細については、http://openssl.org と `openssl req` コマンドの man ページを参照してください。

自己証明書を作成するには、MarkLogic に独自の認証局をインストールし、続いてその認証局を使用してクライアント証明書に自己署名します。詳細については、『Security Guide』の「Creating a Certificate Authority」を参照してください。

クライアント証明書と自己署名による関連キーを取得するには、`xdmp.x509CertificateGenerate` サーバーサイド JavaScript 関数または `xdmp:x509-certificate-generate` XQuery 関数を使用してください。`private-key` パラメータを null に設定し、`credentialId` オプションを自分の認証局に対応するように設定します。次に例を示します。

```
const x509Config = ...;
const cert = xdmp.x509CertificateGenerate(
    x509Config, null, {credentialId:
xdmp.credentialId('ca-cred')});
```

1.7.2.2 アプリケーションサーバーの設定

アプリケーションサーバーも、証明書ベースの認証と SSL を使用するように設定する必要があります。詳細については、『Security Guide』の「Configuring an App Server for External Authentication」および「Procedures for Enabling SSL on App Servers」を参照してください。SSL 用にアプリケーションサーバーを設定する場合は、次の手順も実行します。詳細については、『Security Guide』の「Enabling SSL for an App Server」を参照してください。

1. 「ssl require client certificate」を true に設定します。

2. 「SSL Client Certificate Authorities」の下で「Show」をクリックし、サーバーのクライアント証明書の署名に使用できる認証局を選択します。

1.7.2.3 例：データベースクライアント設定

例えば、「client.crt」というファイルに証明書があり、「clientpriv.pem」というファイルにプライベートキーがある場合、次のようなデータベースクライアント設定で証明書とプライベートキーを使用できます。

```
ml.createDatabaseClient({
  authType: 'certificate',
  cert: fs.readFileSync('client.crt'),
  key: fs.readFileSync('clientpriv.pem'),
  ssl: true
})
```

高度なセキュリティには、パスフレーズで保護できる単一の PKCS12 ファイルに、クライアント証明書とプライベートキーを組み合わせることもできます。詳細については、http://openssl.org と「openssl pkcs12」コマンドの man ページを参照してください。

例えば、「credentials.pfx」という PKCS12 ファイルがある場合、次のようにこのファイルと、データベースクライアント設定のパスフレーズを使用できます。

```
ml.createDatabaseClient({
  authType: 'certificate',
  pfx: fs.readFileSync('credentials.pfx'),
  key: 'yourPassphrase',
  ssl: true
})
```

ベーシックまたはダイジェスト認証で証明書を使用して、これらのメソッドのセキュリティを高めることもできます。例えば、次のコードは、ダイジェスト認証で証明書を使用します。

```
ml.createDatabaseClient({
  user: 'me',
  password: 'mypassword',
  authType: 'digest',
  cert: fs.readFileSync('client.crt'),
  key: fs.readFileSync('clientpriv.pem'),
  ssl: true
})
```

1.7.3 Kerberos 認証の使用

Kerberos 認証を使用するように MarkLogic インストールおよびクライアントアプリケーション環境を設定するには、次の手順を使用します。

- Kerberos を使用するための MarkLogic の設定
- Kerberos を使用するためのクライアントホストの設定
- Kerberos を使用するデータベースクライアントの作成

1.7.3.1 Kerberos を使用するための MarkLogic の設定

Kerberos 認証を使用する前に、外部セキュリティを使用するように MarkLogic を設定する必要があります。Kerberos を使用するようにインストール環境がまだ設定されていない場合、少なくとも次の手順を実行する必要があります。

1. Kerberos 外部セキュリティ設定オブジェクトを作成します。詳細については、『Security Guide』の「Creating an External Authentication Configuration Object」を参照してください。

2. Kerberos キータブファイルを作成し、それを MarkLogic インストール環境にインストールします。詳細については、『Security Guide』の「Creating a Kerberos keytab File」を参照してください。

3. 外部名と関連付けられた 1 人あるいは複数のユーザーを作成します。詳細については、『Security Guide』の「Assigning an External Name to a User」を参照してください。

4. 「kerberos-ticket」認証を使用するようにアプリケーションサーバーを設定します。詳細については、『Security Guide』の「Configuring an App Server for External Authentication」を参照してください。

詳細については、『Security Guide』の「External Security」を参照してください。

1.7.3.2 Kerberos を使用するためのクライアントホストの設定

クライアントでは、Node.js クライアント API が Kerberos Key Distribution Center から TGT（Ticket-Granting Ticket）にアクセスできる必要があります。この API は TGT を使用して Kerberos サービスチケットを取得します。

これらの手順に従って、クライアントアプリケーションで TGT を利用できるようにします。

1. まだインストールされていない場合は、MIT Kerberos をクライアント環境にインストールします。http://www.kerberos.org/software/index.html からこのソフトウェアをダウンロードできます。

2. これが MIT Kerberos の新規インストールである場合、`krb5.conf` ファイルを編集して、インストール環境を設定します。Linux では、デフォルトで、このファイルは `/etc/` にあります。詳細については、https://web.mit.edu/kerberos/krb5-1.15/doc/admin/conf_files/krb5_conf.html を参照してください。

3. クライアントホストで `kinit` を使用して、Kerberos Key Distribution Center で TGT を作成しキャッシュします。`kinit` に与えられるプリンシパルは、「Kerberos を使用するための MarkLogic の設定」の手順を実行するときに MarkLogic ユーザーに関連付けたプリンシパルである必要があります。

詳細については、次のトピックを参照してください。

- https://web.mit.edu/kerberos/krb5-1.15/doc/user/user_commands/kinit.html
- http://web.mit.edu/kerberos/krb5-current/doc/user/tkt_mgmt.html#obtaining-tickets-with-kinit

1.7.3.3 Kerberos を使用するデータベースクライアントの作成

クライアントアプリケーションで、データベースクライアントを作成するときに、`authType` プロパティを「kerberos」に設定します。

例えば、ポート 8000 でローカルホストに接続しており、したがってホストとポートを明示的に指定する必要がないと想定した場合、次の呼び出しで、Kerberos 認証を使用して localhost:8000 に接続するデータベースクライアントオブジェクトが作成されます。

```
ml.createDatabaseClient({authType: 'kerberos'});
```

1.8 このガイドの例の使用方法

Node.js クライアント API を使用した MarkLogic サーバーに対するすべてのリクエストは、`DatabaseClient` オブジェクトを通じてやり取りされます。このため、どの例でも、このオブジェクトを作成することから始まります。`DatabaseClient` を作成するには、ホスト、ポート、ユーザー、およびパスワードなどの MarkLogic サーバー接続情報を指定する必要があります。

このガイドのほとんどの例で、`marklogic.createDatabaseClient` で使用するのに適した接続オブジェクトをエクスポートする `my-connection.js` という名前のモジュールを要求（`require`）することで接続の詳細を抽象化しています。このカプセル化は、利便性のみを目的としたものです。各自のアプリケーションでこのようにする必要はありません。

例えば、このガイドでは、それぞれの例の最初付近に次のステートメントが示されています。

```
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);
```

例を使用するには、まず次のコンテンツが含まれた my-connection.js という名前のファイルを作成する必要があります。このファイルは、このガイドの例をコピーして作成するすべてのスクリプトと同じ場所に配置する必要があります。

```
module.exports = {
  connInfo: {
    host: 'localhost',
    port: 8000,
    user: your-ml-username,
    password: your-ml-user-password
  }
};
```

接続の詳細は、各自の環境に合わせて変更してください。少なくとも user および password プロパティを変更する必要があります。ほとんどの例では、ユーザーが rest-reader または rest-writer ロールまたは同等の権限を持っている必要があります。ただし、一部の操作では、追加の権限が必要になることもあります。詳細については、「セキュリティ要件」（15 ページ）を参照してください。

my-connection.js を作成しない場合は、別の方法で接続の詳細を提供できるように、例の中の marklogic.createDatabaseClient の呼び出しを変更します。

2.0 ドキュメントの操作

この章では、Node.js クライアント API を使用してドキュメントとメタデータを作成、読み取り、更新、および削除する方法に関する次のトピックについて説明します。

2.1 ドキュメント操作の概要

Node.js クライアント API は、ドキュメントとドキュメントメタデータを作成、読み取り、更新、および削除する関数を公開しています。

ほとんどのドキュメント操作関数は `DatabaseClient.documents` インターフェイスを通じて提供されます。例えば、次のコードスニペットは、データベースクライアントオブジェクトを作成し、`documents.read()` メソッドを呼び出してドキュメントを読み取ります。

```
var ml = require('marklogic');
var db =
ml.createDatabaseClient({'user':'me','password':'mypwd'});

db.documents.read('/doc/example.json')....;
```

`DatabaseClient` インターフェイスには、`DatabaseClient.read` および `DatabaseClient.createCollection` など、JavaScript オブジェクトをデータベースドキュメントにバインドするための読み取りおよび書き込み操作が用意されています。通常、このような操作は `DatabaseClient.documents` の同等のメソッドよりもシンプルですが、機能的には劣ります。例えば、`DatabaseClient.read` を使用して、トランザクション ID を指定したり、ドキュメントのメタデータを読み取ったりすることはできません。

DatabaseClient.documents の複数のインターフェイスが、URI やドキュメントコンテンツなど、データをカプセル化するドキュメントディスクリプタを受け付けたり返したりします。詳細については、「入力ドキュメントディスクリプタ」（41 ページ）および「ドキュメントディスクリプタ」（21 ページ）を参照してください。

データをデータベースにロードする場合、DatabaseClient.documents.write メソッドは最大レベルの制御性と豊富な機能セットを提供します。ただし、そのレベルの制御性が必要ない場合は、他のいずれかのインターフェイスのほうが使いやすい可能性もあります。例えば、JavaScript ドメインオブジェクトをデータベースに保存するだけの場合は、DatabaseClient.createCollection を使用すれば、ドキュメントディスクリプタやドキュメント URI を作成しなくても、そのような処理を実行できます。

デフォルトでは、データベースとやり取りするそれぞれの Node.js クライアント API 呼び出しは、完全なトランザクション操作を表します。例えば、DatabaseClient.Documents.write の単一の呼び出しを使用して、複数のドキュメントを更新する場合、すべての更新は同じトランザクションの一部として適用され、操作がサーバー上で完了したらトランザクションはコミットされます。マルチステートメントトランザクションを使用して、クライアントサイドの複数の操作を単一のトランザクションに展開させることができます。詳細については、「トランザクションの管理」（227 ページ）を参照してください。

次の表は、データベースへの書き込みに関連する一般的なタスクと、そのタスクを実行するのに最適なメソッドを示したものです。インターフェイスの完全なリストについては、Node.js API リファレンスを参照してください。

目的	使用するメソッド
JavaScript オブジェクトのコレクションを JSON ドキュメントとしてデータベースに保存する。	DatabaseClient.createCollection 詳細については、「オブジェクトとドキュメントのコレクションの管理」（60 ページ）を参照してください。
DatabaseClient.createCollection を使用して作成した JavaScript オブジェクトのコレクションを更新する。	DatabaseClient.writeCollection 詳細については、「オブジェクトとドキュメントのコレクションの管理」（60 ページ）を参照してください。

目的	使用するメソッド
URI を指定してドキュメントのコレクションを挿入または更新する。	`DatabaseClient.writeCollection` 詳細については、「オブジェクトとドキュメントのコレクションの管理」（60 ページ）を参照してください。
付随するコンテンツがある状態またはない状態でドキュメントメタデータを挿入または更新する。	`DatabaseClient.documents.write` 詳細については、「単一ドキュメントのメタデータの挿入または更新」（46 ページ）を参照してください。
マルチステートメントトランザクションのコンテキストでドキュメントやメタデータを挿入または更新する。	`DatabaseClient.documents.write` 詳細については、「データベースへのドキュメントのロード」（39 ページ）を参照してください。
ドキュメントのロード時にコンテンツ変換を適用する。	`DatabaseClient.documents.write` 詳細については、「データベースへのドキュメントのロード」（39 ページ）および「読み込み時のコンテンツの変換」（48 ページ）を参照してください。
ドキュメント全体を置き換えるのではなく、ドキュメントの一部またはそのメタデータを更新する。	`DatabaseClient.documents.patch` 詳細については、「ドキュメントコンテンツまたはメタデータへのパッチの適用」（79 ページ）を参照してください。

次の表は、データベースからのデータの読み取りに関連する一般的なタスクと、各タスクに最適な関数を示したものです。インターフェイスの完全なリストについては、Node.js API リファレンスを参照してください。

目的	使用するメソッド
1 つあるいは複数のドキュメントのコンテンツを URI を指定して読み取る。	`DatabaseClient.read`
`DatabaseClient.createCollection` を使用して以前にデータベースに保存した JavaScript オブジェクトのコレクションをリストアする。	`DatabaseClient.documents.query` 詳細については、「ドキュメントとメタデータのクエリ」（131 ページ）および「オブジェクトとドキュメントのコレクションの管理」（60 ページ）を参照してください。
1 つあるいは複数のドキュメントやメタデータを URI を指定して読み取る。	`DatabaseClient.documents.read` 詳細については、「データベースからのドキュメントの読み取り」（49 ページ）を参照してください。
1 つあるいは複数のドキュメントやメタデータのコンテンツを URI を指定して読み取って、読み取り変換を適用する。	`DatabaseClient.documents.read` 詳細については、「データベースからのドキュメントの読み取り」（49 ページ）および「取得時のコンテンツの変換」（56 ページ）を参照してください。
マルチステートメントトランザクションのコンテキストで 1 つあるいは複数のドキュメントやメタデータを URI を指定して読み取る。	`DatabaseClient.documents.read` 詳細については、「データベースからのドキュメントの読み取り」（49 ページ）を参照してください。
クエリにマッチするドキュメントやメタデータを読み取る。	`DatabaseClient.documents.query` 詳細については、「ドキュメントとメタデータのクエリ」（131 ページ）を参照してください。

目的	使用するメソッド
レキシコンおよびレンジインデックス内の値をクエリおよび分析する。詳細については、「レキシコンおよびレンジインデックスのクエリ」（174 ページ）を参照してください。	`DatabaseClient.values.read`
データベースからセマンティックグラフを読み取る。詳細については、Node.js API リファレンスを参照してください。	`DatabaseClient.graphs.read`

2.2 データベースへのドキュメントのロード

ドキュメントコンテンツおよびメタデータをデータベースに挿入するには、`DatabaseClient.documents.write` または `DatabaseClient.documents.createWriteStream` メソッドを使用します。ストリームインターフェイスは、主にバイナリなどのサイズの大きいドキュメントの書き込みを目的としています。

- 概要
- 入力ドキュメントディスクリプタ
- 呼び出しの規則
- 例：単一のドキュメントのロード
- 例：複数のドキュメントのロード
- 単一ドキュメントのメタデータの挿入または更新
- ドキュメント URI の自動生成
- 読み込み時のコンテンツの変換

2.2.1 概要

ドキュメント全体やメタデータを挿入または更新するには、`DatabaseClient.documents.write` を使用します。ドキュメントの一部またはそのメタデータだけを更新するには、`DatabaseClient.documents.patch` を使用します。詳細については、「ドキュメントコンテンツまたはメタデータへのパッチの適用」（79 ページ）を参照してください。

`write` 関数へのプライマリ入力は、1 つあるいは複数のドキュメントディスクリプタです。各ディスクリプタは、1 つのドキュメント URI と、書き込むコンテンツやメタデータをカプセル化します。詳細については、「入力ドキュメントディスクリプタ」（41 ページ）を参照してください。

例えば次の呼び出しは、URI が /doc/example.json の単一のドキュメントを書き込みます。

```
var db = marklogic.createDatabaseClient(...);
db.documents.write(
  { uri: '/doc/example.json',
    contentType: 'application/json',
    content: { some: 'data' }
  }
);
```

複数のドキュメントを書き込むには、複数のディスクリプタを渡します。例えば、次の呼び出しは 2 つのドキュメントを書き込みます。

```
db.documents.write(
    { uri: '/doc/example1.json',
      contentType: 'application/json',
      content: { data: 'one' }
    },
    { uri: '/doc/example2.json',
      contentType: 'application/json',
      content: { data: 'two' }
    },
);
```

ディスクリプタは、個別のパラメータとして渡したり、配列やカプセル化呼び出しオブジェクトとして渡すことができます。詳細については、「呼び出しの規則」（41 ページ）を参照してください。

戻り値に対して result() 関数を呼び出すと、書き込み操作の成功または失敗に基づいて操作を実行できます。詳細については、「サポートされている結果処理手法」（22 ページ）を参照してください。

例えば、次のコードスニペットは、書き込みが失敗した場合にエラーメッセージをコンソールに出力します。

```
db.documents.write(
  { uri: '/doc/example.json',
    contentType: 'application/json',
    content: { some: 'data' }
  }
).result(null, function(error) {
    console.log(
})
```

2.2.2 入力ドキュメントディスクリプタ

書き込む各ドキュメントは、ドキュメントディスクリプタで記述します。ドキュメントディスクリプタには、URI とコンテンツまたはメタデータのいずれか、あるいはコンテンツとメタデータの両方が含まれている必要があります。詳細については、「ドキュメントディスクリプタ」（21 ページ）を参照してください。

例えば次のコードは、URI が `/doc/example.json` のドキュメントのドキュメントディスクリプタです。ドキュメントコンテンツは、1 つのプロパティを含む JavaScript オブジェクトとして表します。

```
{ uri: '/doc/example.json', content: {'key': 'value'} }
```

ドキュメントディスクリプタの `content` プロパティには、オブジェクト、文字列、`Buffer` または `ReadableStream` を指定できます。

特定の単一のドキュメントに適用する場合、メタデータはドキュメントディスクリプタのプロパティとして表します。複数のドキュメントに適用する場合、メタデータは呼び出しオブジェクトのプロパティとして表します。詳細については、「単一ドキュメントのメタデータの挿入または更新」（46 ページ）を参照してください。

例えば、次のドキュメントディスクリプタには、URI が `/doc/example.json` のドキュメントのコレクションとドキュメントクオリティメタデータが含まれています。

```
{ uri: '/doc/example.json',
  content: {'key': 'value'},
  collections: [ 'collection1', 'collection2' ],
  quality: 2
}
```

2.2.3 呼び出しの規則

`DatabaseClient.documents.write` には、少なくとも 1 つのドキュメントディスクリプタを渡す必要があります。また、変換名やトランザクション ID などの追加のプロパティを含めることもできます。`documents.write` に渡すパラメータは、次のいずれかの形式にできます。

- 1 つあるいは複数のドキュメントディスクリプタ：
  `db.documents.write(desc1, desc2,...)`
- 1 つあるいは複数のドキュメントディスクリプタの配列：
  `db.documents.write([desc1, desc2, ...])`
- ドキュメントディスクリプタ配列と追加のオプションプロパティをカプセル化する呼び出しオブジェクト：`db.documents.write({documents: [desc1, desc2, ...], txid: ..., ...})`

次の各呼び出しは同じように機能します。

```
// passing document descriptors as parameters
db.documents.write(
  {uri: '/doc/example1.json', content: {...}},
  {uri: '/doc/example2.json', content: {...}}
);

// passing document descriptors in an array
db.documents.write([
  {uri: '/doc/example1.json', content: {...}},
  {uri: '/doc/example1.json', content: {...}}
]);

// passing document descriptors in a call object
db.documents.write({
  documents: [
    {uri: '/doc/example1.json', content: {...}},
    {uri: '/doc/example2.json', content: {...}}
  ],
  additional optional properties
});
```

追加のオプションプロパティには、変換の指定内容、トランザクション ID、テンポラルコレクション名を含めることができます。詳細については、Node.js API リファレンスを参照してください。このようなプロパティは、常に呼び出しオブジェクトのプロパティとして指定できます。

例えば、次の呼び出しには、呼び出しオブジェクトの追加プロパティとしてトランザクション ID（`txid`）が含まれています。

```
// passing a transaction id as a call object property
db.documents.write({
  documents: [
    {uri: '/doc/example1.json', content: {...}},
    {uri: '/doc/example2.json', content: {...}}
  ],
  txid: '1234567890'
});
```

ドキュメントディスクリプタが 1 つしかない場合は、呼び出しオブジェクトを使用する代わりに、追加のオプションプロパティをドキュメントディスクリプタのプロパティとして渡すことができます。例えば、次の呼び出しには、1 つのドキュメントディスクリプタの内部にトランザクション ID が含まれています。

```
// passing a transaction id as a document descriptor
property
db.documents.write(
  { uri: '/doc/example1.json',
    content: {...},
    txid: '1234567890'
  }
);
```

2.2.4 例：単一のドキュメントのロード

この例では、DatabaseClient.documents.write を使用して単一のドキュメントをデータベースに挿入します。

ロードするドキュメントはドキュメントディスクリプタで指定します。次のドキュメントディスクリプタは、URI が /doc/example.json の JSON ドキュメントを表しています。ここではドキュメントコンテンツが JavaScript オブジェクトとして表現されていますが、文字列、Buffer、ReadableStream にすることもできます。

```
{ uri: '/doc/example.json',
  contentType: 'application/json',
  content: { some: 'data' }
})
```

以下のコードは、データベースクライアントを作成して DatabaseClient.documents.write を呼び出すことで、ドキュメントをロードします。この例では、result 関数を呼び出してエラーハンドラを渡して書き込みの失敗をチェックします。また、書き込みが成功した場合は何も行われないため、result() に対する最初のパラメータとして null が渡されます。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.documents.write(
    { uri: '/doc/example.json',
      contentType: 'application/json',
      content: { some: 'data' }
    })
  .result(null, function(error) {
      console.log(JSON.stringify(error));
    });
```

その他の例については、node-client-api ソースディレクトリの examples/before-load.js および examples/write-remove.js を参照してください。

メタデータを含めるには、メタデータプロパティをドキュメントディスクリプタに追加します。例えば、ドキュメントをコレクションに追加するには、collections プロパティをディスクリプタに追加します。

```
db.documents.write(
    { uri: '/doc/example.json',
      contentType: 'application/json',
      content: { some: 'data' },
      collections: [ 'collection1', 'collection2' ]
    })
```

呼び出しオブジェクトを使用することで、トランザクション ID や書き込み変換などのオプションの追加パラメータを含めることができます。詳細については、「呼び出しの規則」（41 ページ）を参照してください。

2.2.5 例：複数のドキュメントのロード

この例は、「例：単一のドキュメントのロード」（43 ページ）をベースにしており、DatabaseClient.documents.write を使用して複数のドキュメントをデータベースに挿入します。

MarkLogic サーバーに対する単一のリクエストで複数のドキュメントを挿入または更新するには、複数のドキュメントディスクリプタを DatabaseClient.documents.write に渡します。

次のコードは、URI が /doc/example1.json および /doc/example2.json の 2 つのドキュメントをデータベースに挿入します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.documents.write(
    { uri: '/doc/example1.json',
      contentType: 'application/json',
      content: { data: 'one' }
    },
    { uri: '/doc/example2.json',
      contentType: 'application/json',
      content: { data: 'two' }
    }
```

```
).result(null, function(error) {
    console.log(JSON.stringify(error));
});
```

複数のドキュメントを書き込むと、書き込まれた各ドキュメントのディスクリプタが含まれたオブジェクトを返します。このディスクリプタには、URI、コンテンツの解釈に使用された MIME タイプ、書き込みにより更新されたもの（コンテンツ、メタデータ、またはその両方）が含まれます。

例えば、上記の呼び出しの戻り値は次のようになります。

```
{ documents: [
  { uri: '/doc/example1.json',
    mime-type: 'application/json',
    category: ['metadata','content']
  },{
    uri: '/doc/example2.json',
    mime-type: 'application/json',
    category: ['metadata','content']
  }
]}
```

メタデータを明示的に指定しなかったにもかかわらず、`category` プロパティは、コンテンツとメタデータの両方が更新されたことを示していることに注目してください。これは、システムのデフォルトのメタデータ値がドキュメントに暗黙的に割り当てられているためです。

複数のドキュメントをロードするときにドキュメントのメタデータを含めるには、ドキュメント固有のメタデータをそのドキュメントのディスクリプタに含めます。複数のドキュメントに適用されるメタデータを指定するには、パラメータリストまたはドキュメントのプロパティにメタデータディスクリプタを含めます。

例えば、2 つのドキュメントをコレクション「examples」に追加するには、次に示すようにメタデータディスクリプタをドキュメントディスクリプタの前に追加します。一連のディスクリプタは、それらが記述されている順序で処理されるため、ディスクリプタの順序は重要です。メタデータディスクリプタは、パラメータリストまたは `documents` 配列で、そのディスクリプタよりも後に記述されているドキュメントディスクリプタにのみ影響します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.documents.write({
  documents: [
```

```
    { contentType: 'application/json',
      collections: [ 'examples' ]
    },
    { uri: '/doc/example1.json',
      contentType: 'application/json',
      content: { data: 'one' }
    },
    { uri: '/doc/example2.json',
      contentType: 'application/json',
      content: { data: 'two' }
    }
  ]
}).result(null, function(error) {
      console.log(JSON.stringify(error));
});
```

2.2.6 単一ドキュメントのメタデータの挿入または更新

特定の単一のドキュメントのメタデータを挿入または更新するには、`DatabaseClient.documents.write` に渡すドキュメントディスクリプタに 1 つあるいは複数のメタデータプロパティを含めます。複数のドキュメントに対して同じメタデータを挿入または更新するには、複数のドキュメントの書き込みにメタデータディスクリプタを含めます。詳細については、「メタデータを扱う」（73 ページ）を参照してください。

注： パーミッションを設定する場合には、少なくとも 1 つの更新パーミッションを含める必要があります。

メタデータは、更新時に置換されます。マージされるわけではありません。例えば、ドキュメントディスクリプタに `collections` プロパティが含まれている場合は、`DatabaseClient.documents.write` を呼び出すと、ドキュメントの既存のすべてのコレクション関連付けが置き換えられます。

次の例は、URI が `/doc/example.json` のドキュメントを挿入し、それをコレクション「examples」と「metadata-examples」に追加します。ドキュメントがすでに存在していて他のコレクションの一部である場合、そのドキュメントはそれらのコレクションから削除されます。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.documents.write(
    { uri: '/doc/example.json',
      collections: ['examples', 'metadata-examples'],
      contentType: 'application/json',
```

```
      content: { some: 'data' }
    })
  .result(null, function(error) {
      console.log(JSON.stringify(error));
    });
```

ドキュメントのメタデータのみを挿入または更新するには、content プロパティを省略します。例えば次のコードは、ドキュメントコンテンツを変更することなく、クオリティを 2 に設定し、コレクションを「some-collection」に設定します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.documents.write(
    { uri: '/doc/example.json',
      collections: ['some-collection'],
      quality: 2,
    })
  .result(null, function(error) {
      console.log(JSON.stringify(error));
    });
```

2.2.7 ドキュメント URI の自動生成

以降で説明するように、ドキュメントディスクリプタの uri プロパティを extension プロパティで置き換えることで、挿入時にドキュメント URI を自動的に生成できます。

> **注：** この機能は、新しいドキュメントを作成する場合にのみ使用できます。既存のドキュメントを更新するには、そのドキュメントの URI がわかっている必要があります。

この機能を使用するには、次の特性を持つドキュメントディスクリプタを作成します。

- uri プロパティを省略する。
- 生成された URI 拡張子（「xml」または「json」など）を指定する extension プロパティを含める。ドット（.）プレフィックスは含めないでください。つまり、「.json」ではなく「json」と指定します。
- 必要に応じて、生成された URI のデータベースディレクトリプレフィックスを指定する directory プロパティを含める。ディレクトリプレフィックスは、末尾がスラッシュ（/）でなければなりません。

次の例は、形式が /my/directory/< *自動生成* >.json の URI のデータベースにドキュメントを挿入します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.documents.write(
    { extension: 'json',
      directory: '/my/directory/',
      content: { some: 'data' },
      contentType: 'application/json'
    }
).result(
    function(response) {
      console.log('Loaded ' + response.documents[0].uri);
    },
    function(error) {
      console.log(JSON.stringify(error));
    }
);
```

上記のスクリプトを実行し、正常に完了すると、次のような出力が返されます。

```
Loaded /my/directory/16764526972136717799.json
```

2.2.8 読み込み時のコンテンツの変換

カスタムの書き込み変換を適用すると、読み込み時にコンテンツを変換できます。変換はサーバーサイド XQuery、JavaScript、XSLT のいずれかで、REST API インスタンスに関連付けられている modules データベースにインストールします。また、変換は config.transforms 関数を使用してインストールできます。このトピックでは、読み込みの際に変換を適用する方法について説明します。詳細と例については、「コンテンツ変換機能を扱う」（245 ページ）を参照してください。

ドキュメントの作成時または更新時に変換を適用するには、transform プロパティが含まれた呼び出しオブジェクトを持つ documents.write を呼び出します。transform プロパティは、変換名と、変換で想定されるあらゆるパラメータをカプセル化します。transform プロパティは次のような形式になります。

```
transform: [transformName, {param1: value, param2: value,
...}]
```

例えば、次のコードスニペットは、my-transform という名前でインストールされた変換を適用し、2 つのパラメータの値を渡します。

```
db.documents.write({
  documents: [
    {uri: '/doc/example1.json', content: {...}},
    {uri: '/doc/example2.json', content: {...}}
  ],
  transform: [
    'my-transform',
    { my-first-param: 'value',
      my-second-param: 42
    }
  ]
});
```

ドキュメントを 1 つだけ挿入または更新する場合は、transform プロパティをドキュメントディスクリプタに埋め込むこともできます。次に例を示します。

```
db.documents.write({
  uri: '/doc/example1.json',
  content: {...}},
  transform: [
    'my-transform',
    { my-first-param: 'value',
      my-second-param: 42
    }
  ]
});
```

2.3 データベースからのドキュメントの読み取り

データベースから 1 つあるいは複数のドキュメントやメタデータを読み取るには、DatabaseClient.documents.read を使用します。このセクションでは、次の内容を取り上げます。

- URI によるドキュメントのコンテンツの取得
- ドキュメントに関するメタデータの取得
- 例：コンテンツとメタデータの取得
- 取得時のコンテンツの変換

2.3.1 URI によるドキュメントのコンテンツの取得

URI を使用してデータベースからドキュメントのコンテンツを取得するには、`DatabaseClient.documents.read` または `DatabaseClient.read` を使用します。また、クエリにマッチするドキュメントのコンテンツまたはメタデータを取得することもできます。詳細については、「ドキュメントとメタデータのクエリ」（131 ページ）を参照してください。

`DatabaseClient.read` と `DatabaseClient.Documents.read` のどちらでも URI を指定してドキュメントを読み取れますが、両者は機能および複雑さの点で異なります。メタデータの読み取り、マルチステートメントトランザクションの使用、読み取り変換の適用、コンテンツタイプなどの情報へのアクセスが必要な場合は、`DatabaseClient.Documents.read` を使用します。`DatabaseClient.read` は、入力として URI のリストのみを受け付け、要求されたドキュメントのコンテンツのみを返します。

2 つの関数は異なる出力を返します。`Documents.read` は、コンテンツだけではなく、ドキュメントディスクリプタの配列を返します。DatabaseClient.documents.read は、入力 URI リストと同じ順序で各ドキュメントのコンテンツが含まれている配列のみを返します。どちらの関数の出力も、コールバック、Promise、または Stream を使用して処理できます。詳細については、「サポートされている結果処理手法」（22 ページ）を参照してください。

例えば、次のコードは、`DatabaseClient.Documents.read` を使用して URI が `/doc/example1.json` のドキュメントを読み取り、`result` メソッドによって返された Promise を使用して出力を処理します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.documents.read('/doc/example1.json')
   .result( function(documents) {
      documents.forEach(function(document) {
        console.log(JSON.stringify(document));
      });
    }, function(error) {
      console.log(JSON.stringify(error, null, 2));
    });
```

result によって返される完全なディスクリプタは次のようになります。返された配列には、返された各ドキュメントのドキュメントディスクリプタ項目が含まれています。この場合、ドキュメントは 1 つだけです。

```
{
  "partType":"attachment",
  "uri":"/doc/example1.json",
  "category":"content",
  "format":"json",
  "contentType":"application/json",
  "contentLength":"14",
  "content":{"data":"one"}
}
```

DatabaseClient.read を使用して同じドキュメントを読み取ると、次のような出力が返されます。コンテンツのみで、ディスクリプタではない点に注目してください。

```
db.read('/doc/example1.json').result(...);
==>
{"data":"one"}
```

複数の URI を渡すことで、複数のドキュメントを読み取ることができます。次の例は、2 つのドキュメントを読み取り、Stream パターンを使用して結果を処理します。データハンドラは、入力としてドキュメントディスクリプタを受け取ります。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.documents.read('/doc/example1.json',
'/doc/example2.json')
  .stream().on('data', function(document) {
    console.log('Read ' + document.uri);
  }).
  on('end', function() {
    console.log('finished');
  }).
  on('error', function(error) {
    console.log(JSON.stringify(error));
    done();
  });
```

呼び出しオブジェクトパラメータを Documents.read に渡し、ドキュメントのどの部分を返すかを指定する categories プロパティを含めることで、メタデータをリクエストできます。詳細については、「ドキュメントに関するメタデータの取得」（52 ページ）を参照してください。

Documents.read を呼び出すときに、読み取り変換を使用して、レスポンスが作成される前にサーバーサイド変換をコンテンツに適用できます。詳細については、「取得時のコンテンツの変換」（56 ページ）を参照してください。

2.3.2 ドキュメントに関するメタデータの取得

URI でドキュメントを読み取るときにメタデータを取得するには、categories プロパティを含む DatabaseClient.documents.read に呼び出しオブジェクトを渡します。メタデータは、すべて（カテゴリ値 metadata）あるいは一部（カテゴリ値 collections、permissions、properties、metadataValues、および quality）を取得できます。コンテンツとメタデータの両方を取得するには、カテゴリ値 content を含めます。

> **注：** metadataValues メタデータカテゴリは単純なキー / 値メタデータを表し、メタデータフィールドと呼ばれることがあります。詳細については、『Administrator's Guide』の「Metadata Fields」を参照してください。

例えば、次のコードはドキュメント /doc/example.json に関するすべてのメタデータを取得しますが、コンテンツは取得しません。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.documents.read({
  uris: ['/doc/example.json'],
  categories: ['metadata']
}).result(
    function(documents) {
      for (var i in documents)
        console.log('Read metadata for ' +
documents[i].uri);
    },
    function(error) {
      console.log(JSON.stringify(error));
    }
);
```

結果は、リクエストしたドキュメントのすべてのメタデータプロパティを含むドキュメントディスクリプタになります。メタデータのカテゴリおよび形式の詳細については、「メタデータを扱う」（73 ページ）を参照してください。

```
[{
  "partType":"attachment",
  "uri":"/doc/example.json",
  "category":"metadata",
  "format":"json",
  "contentType":"application/json",
  "contentLength":"168",
  "collections":[],
  "permissions":[
    {"role-name":"rest-writer","capabilities":["update"]},
    {"role-name":"rest-reader","capabilities":["read"]}
  ],
  "properties":{},
  "quality":0
}]
```

次の例は、2 つの JSON ドキュメントに関するコンテンツとコレクションのメタデータを取得します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.documents.read({
  uris: ['/doc/example1.json', '/doc/example2.json'],
  categories: ['collections', 'content']
}).stream().
  on('data', function(document) {
    console.log('Collections for ' + document.uri + ': '
      + JSON.stringify(document.collections));
  }).
  on('end', function() {
    console.log('finished');
  }).
  on('error', function(error) {
    console.log(JSON.stringify(error));
  });
```

結果は、collections プロパティと content プロパティを含む、各ドキュメントのドキュメントディスクリプタになります。

```
{
  "partType" : "attachment",
  "uri" : "/doc/example2.json",
  "category" : "content",
  "format" : "json",
  "contentType" : "application/json",
  "contentLength" : "14",
  "collections" : ["collection1", "collection2"],
  "content" : {"data":"two"}
}
```

2.3.3 例：コンテンツとメタデータの取得

この例では、ドキュメントのコンテンツとメタデータの読み取り方法を示します。

以下のスクリプトは、「examples」コレクションにあり、ドキュメントクオリティが 2 のドキュメント例をデータベースに書き込みます。次に、読み取りドキュメントディスクリプタの categories プロパティに 'content' と 'metadata' の両方を含めて、ドキュメントとメタデータを単一の操作で読み取ります。また、'collections' または 'permissions' などの特定のメタデータプロパティを指定することもできます。

例を実行するには、次のスクリプトをファイルにコピーし、node コマンドを使用して実行します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');

var db = marklogic.createDatabaseClient(my.connInfo);

// (1) Seed the database with an example document that has
custom metadata
db.documents.write({
  uri: '/read/example.json',
  contentType: 'application/json',
  collections: ['examples'],
  metadataValues: {key1: 'val1', key2: 2},
  quality: 2,
  content: { some: 'data' }
}).result().then(function(response) {
  // (2) Read back the content and metadata
  return db.documents.read({
    uris: [response.documents[0].uri],
```

```
    categories: ['content', 'metadata']
  }).result();
}).then(function(documents) {
  // Emit the read results
  console.log('CONTENT: ' +
              JSON.stringify(documents[0].content));
  console.log('COLLECTIONS: ' +
              JSON.stringify(documents[0].collections));
  console.log('PERMISSIONS: ' +
             JSON.stringify(documents[0].permissions, null,
2));
  console.log('PROPERTIES: ' +
             JSON.stringify(documents[0].properties, null,
2));
  console.log('QUALITY: ' +
              JSON.stringify(documents[0].quality, null,
2));
  console.log("METADATAVALUES: " +
              JSON.stringify(documents[0].metadataValues,
null, 2));

});
```

このスクリプトにより、次のような出力が生成されます。

```
CONTENT: {"some": "data"}
COLLECTIONS: ["examples"]
PERMISSIONS: [
  {
    "role-name": "rest-writer",
    "capabilities": [
      "update"
    ]
  },
  {
    "role-name": "rest-reader",
    "capabilities": [
      "read"
    ]
  }
]
PROPERTIES: {}
QUALITY: 2
METADATAVALUES: {
  "key2": 2,
  "key1": "val1"
}
```

2.3.4 取得時のコンテンツの変換

ドキュメントには、コンテンツをリクエスタに返す前に、カスタムのサーバーサイド変換を適用できます。変換は JavaScript モジュール、XQuery モジュール、XSLT スタイルシートのいずれかであり、`DatabaseClient.config.transforms.write` 関数を使用して REST API インスタンスにインストールします。詳細については、「コンテンツ変換機能を扱う」（245 ページ）を参照してください。

ドキュメントの取得時に自動的に適用されるデフォルトの読み取り変換を設定できます。また、ほとんどの読み取り操作では、渡される呼び出しオブジェクトに `transform` プロパティを含めることで、リクエストごとの変換も指定できます。デフォルトの変換とリクエストごとの変換の両方が存在する場合、変換は連結され、デフォルトの変換が最初に実行されます。そのため次の図に示すように、デフォルトの変換の出力がリクエストごとの変換の入力になります。

デフォルトの変換を設定するには、REST API インスタンスの `document-transform-out` 設定パラメータを設定します。インスタンス全体に適用される設定パラメータは、`DatabaseClient.config.serverprops.write` 関数を使用して設定します。詳細については、「インスタンスプロパティの設定」（283 ページ）を参照してください。

リクエストごとの変換を指定するには、`DatabaseClient.documents.read` といった読み取り操作の呼び出しオブジェクトの形式を使用して `transform` プロパティを含めます。`transform` プロパティの値は配列です。最初の項目は変換名で、オプションの追加項目では変換が想定するパラメータの名前と値を指定します。つまり、`transform` プロパティは次のような形式になります。

```
transform: [transformName, {param1: value, param2: value,
...}]
```

次の例は、「example」という名前でインストールされていて、「reviewer」という名前の単一のパラメータを想定する変換を適用します。完全な例については、「例：読み取り、書き込み、およびクエリの変換機能」（250 ページ）を参照してください。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.documents.read({
    uris: ['/doc/example.json'],
```

```
    transform: ['example', {reviewer: 'me'}]
  }).result(
    function(documents) {
      for (var i in documents)
        console.log('Document ' + documents[i].uri + ': ');
        console.log(JSON.stringify(documents[i].content));
    }
);
```

2.4 データベースからのコンテンツの削除

Node.js クライアント API を使用すると、URI、コレクション、またはディレクトリを指定してドキュメントを削除できます。

- URI によるドキュメントの削除
- 一連のドキュメントの削除
- すべてのドキュメントの削除

2.4.1 URI によるドキュメントの削除

URI で 1 つあるいは複数のドキュメントを削除するには、`DatabaseClient.remove` または `DatabaseClient.documents.remove` を使用します。いずれの関数も、URI でドキュメントを削除できますが、`DatabaseClient.remove` のほうがより単純で制限の高いインターフェイスを提供します。コレクションまたはディレクトリ別に複数のドキュメントを削除することもできます。詳細については、「一連のドキュメントの削除」（58 ページ）を参照してください。

ドキュメントを削除すると、関連付けられたメタデータも削除されます。抽出済みメタデータを別個の XHTML ドキュメントとして格納しているバイナリドキュメントを削除すると、プロパティドキュメントも削除されます。詳細については、「バイナリドキュメントを扱う」（69 ページ）を参照してください。

以下の例は、`/doc/example1.json` と `/docs/example2.json` のドキュメントを削除します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.documents.remove('/doc/example1.json','/doc/example2.js
on').result(
    function(response) {
      console.log(JSON.stringify(response));
    }
);
```

レスポンスには、ドキュメント URI と成功インジケータが含まれます。例えば、上記のプログラムでは次の出力が生成されます。

```
{ "uris":["/doc/example1.json", "/doc/example2.json"],
  "removed":true }
```

`DatabaseClient.remove` でドキュメントを削除するときに、レスポンスには URI だけが含まれます。次に例を示します。

```
db.remove('/doc/example1.json')
==> ['/doc/example1.json']

db.remove('/doc/example1.json','/doc/example2.json')
==> ['/doc/example1.json','/doc/example2.json']
```

その他の例については、GitHub で入手できる `node-client-api` ソース（http://github.com/marklogic/node-client-api）の例とテストを参照してください。

存在しないドキュメントを削除しようとした場合でも、存在するドキュメントを正常に削除した場合と同じ出力が生成されます。

複数のドキュメントを削除するときに、個々のパラメータ（呼び出しに他のパラメータがない場合）、URI の配列、または値が URI である `uris` プロパティを持つオブジェクトとして URI を指定できます。複数のドキュメントを削除するときに、いずれかのドキュメントの削除でエラーが発生した場合、バッチ全体が失敗します。

トランザクション ID やテンポラルコレクション情報など、追加のパラメータを `DatabaseClient.documents.remove` に指定できます。詳細については、Node.js API リファレンスを参照してください。

2.4.2 一連のドキュメントの削除

`DatabaseClient.documents.removeAll` を使用すると、コレクション内またはデータベースディレクトリ内のすべてのドキュメントを削除できます。

コレクション別にドキュメントを削除するには、次の形式を使用します。

```
db.documents.removeAll({collection:..., other-properties...}
```

ディレクトリ別にドキュメントを削除するには、次の形式を使用します。

```
db.documents.removeAll({directory:..., other-properties...}
```

オプションの *other-properties* には、トランザクション ID を含めることができます。詳細については、Node.js API リファレンスを参照してください。

コレクション内またはディレクトリ内のすべてのドキュメントを削除するには、`rest-writer` ロールまたは同等の権限が必要です。

> **注：** ディレクトリ別にドキュメントを削除するには、ディレクトリ名の末尾のスラッシュ（「/」）を含める必要があります。

次の例は、コレクション「/countries」内のすべてのドキュメントを削除します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.documents.removeAll({collection: '/countries'}).result(
    function(response) {
      console.log(JSON.stringify(response));
    }
);

==> {"exists":false,"collection":"/countries"}
```

次の例は、ディレクトリ「/doc/」内のすべてのドキュメントを削除します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.documents.removeAll({directory: '/doc/'}).result(
    function(response) {
      console.log(JSON.stringify(response));
    }
);

==> {"exists":false,"directory":"/doc/"}
```

`DatabaseClient.documents.removeAll` に渡す呼び出しオブジェクトに `txid` プロパティを含めて、削除を実行するトランザクションを指定することもできます。次に例を示します。

```
db.documents.removeAll({directory: '/doc/', txid:
'1234567890'})
```

その他の例については、GitHub で入手できる `node-client-api` ソース（http://github.com/marklogic/node-client-api）の例とテストを参照してください。

2.4.3 すべてのドキュメントの削除

データベース内のすべてのドキュメントを削除するには、`DatabaseClient.documents.removeAll` を呼び出して、呼び出しオブジェクトに all プロパティを含め、値を true にします。次に例を示します。

```
db.documents.removeAll({all: true})
```

データベース内のすべてのドキュメントを削除するには、`rest-admin` または同等の権限が必要です。

この方法でデータベースを消去する際には確認は行われず、その他の安全策も提供されません。バックアップを作成しておくことをお勧めします。

次の例は、データベース内のすべてのドキュメントを削除します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.documents.removeAll({all: true}).result(
    function(response) {
      console.log(JSON.stringify(response));
    }
);

==> {"exists":false,"allDocuments":true}
```

`DatabaseClient.documents.removeAll` に渡す呼び出しオブジェクトに `txid` プロパティを含めて、削除を実行するトランザクションを指定することもできます。次に例を示します。

```
db.documents.removeAll({all: true, txid: '1234567890'})
```

2.5 オブジェクトとドキュメントのコレクションの管理

次の操作を使用すると、データベース内の JavaScript オブジェクトのコレクションを簡単に管理できます。オブジェクトは JSON としてシリアライズ可能でなければなりません。

- `DatabaseClient.createCollection` : JavaScript オブジェクトのコレクションを、自動生成された URI を持つ JSON ドキュメントとしてデータベースに格納します。オブジェクトは JSON としてシリアライズ可能でなければなりません。

- `DatabaseClient.read` : URI を指定して 1 つあるいは複数の JavaScript オブジェクトをデータベースから読み取ります。`DatabaseClient.documents.read` とは異なり、このメソッドはドキュメントディスクリプタを返すのではなく、ドキュメントのコンテンツだけを返します。ドキュメントは、入力 URI と同じ順序で返されます。
- `DatabaseClient.documents.query` : コレクション内のすべてのドキュメントを検索してオブジェクトをリストアするか、コレクションを検索します。
- `DatabaseClient.writeCollection` : URI およびコレクション名を指定してオブジェクトまたは他のドキュメントを更新します。コレクション内のオブジェクトを更新するには、`creatCollection` ではなく、このメソッドを使用してください。これは、`createCollection` が常に各オブジェクトの新しいドキュメントを作成するためです。
- `DatabaseClient.removeCollection` : コレクション名でオブジェクトまたは他のドキュメントを削除します。

`createCollection` の呼び出しは加算的です。つまり、同じコレクションに対して `createCollection` を複数回呼び出しても、コレクション内にすでにあるドキュメント（オブジェクト）はコレクション内に残ります。ただし、`createCollection` を呼び出すたびに各オブジェクトの新しいドキュメントが生成されます。つまり、同じオブジェクトに対して 2 回呼び出すと、以前のオブジェクトが上書きされるのではなく、新しいオブジェクトが作成されることになります。コレクション内のオブジェクト / ドキュメントを更新するには、`DatabaseClient.writeCollection` を使用します。

ドキュメントを詳細に制御する必要がある場合は、`DatabaseClient.documents.write` を使用します。例えば、`createCollection` を使用した場合、ドキュメント URI を制御したり、権限やドキュメントのプロパティなどのメタデータを含めたり、トランザクション ID を指定したりすることはできません。

`DatabaseClient.documents.query` でドキュメントを検索する方法の詳細については、「ドキュメントとメタデータのクエリ」（131 ページ）を参照してください。

以下のスクリプト例は、次の処理を実行します。

- 一連のオブジェクトからコレクションを作成する。
- すべてのオブジェクトを読み取る。
- `kind='cat'` のオブジェクトのみを検索する。
- コレクションを削除する。

例を実行するには、次のスクリプトをファイルにコピーし、node コマンドで実行します。「このガイドの例の使用方法」（33 ページ）で説明しているように、データベース接続情報は my-connection.js にカプセル化されます。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');

var db = marklogic.createDatabaseClient(my.connInfo);
var qb = marklogic.queryBuilder;

// The collection of objects to persist
var pets = [
  { name: 'fluffy', kind: 'cat' },
  { name: 'fido', kind: 'dog' },
  { name: 'flipper', kind: 'fish' },
  { name: 'flo', kind: 'rodent' }
];
var collName = 'pets';

// (1) Write the objects to the database
db.createCollection(collName, pets).result()
  .then(function(uris) {
    console.log('Saved ' + uris.length + ' objects with
URIs:');
    console.log(uris);

    // (2) Read back all objects in the collection
    return db.documents.query(
      qb.where(qb.collection(collName))
    ).result();
  }, function(error) {
    console.log(JSON.stringify(error));
  }).then( function(documents) {
    console.log('\nFound ' + documents.length + '
documents:');
    documents.forEach( function(document) {
      console.log(document.content);
    });

    // (3) Find the cats in the collection
    return db.documents.query(
      qb.where(qb.collection(collName), qb.value('kind',
'cat'))
    ).result();
  }).then( function(documents) {
    console.log('\nFound the following cats:');
```

```
    documents.forEach( function(document) {
      console.log('  ' + document.content.name);
    });

    // (4) Remove the collection from the database
    db.removeCollection(collName);
  });
```

スクリプトを実行すると、次のような出力が生成されます。

```
Saved 4 objects with URIs:
[ '/717293155828968327.json',
  '/5648624202818659648.json',
  '/4552049485172923004.json',
  '/16796864305170577329.json' ]

Found 4 documents:
{ name: 'fido', kind: 'dog' }
{ name: 'flipper', kind: 'fish' }
{ name: 'flo', kind: 'rodent' }
{ name: 'fluffy', kind: 'cat' }

Found the following cats:
  fluffy
```

2.6 簡易的なドキュメントチェックの実行

データベース内にドキュメントが存在しているかどうかテストしたり、ドキュメントを取得せずにドキュメント ID を取得したりするには（コンテンツのバージョン管理が有効な場合）、`DatabaseClient.documents.probe` または `DatabaseClient.probe` を使用します。

次の例は、ドキュメント `/doc/example.json` が存在するかどうかを調べます。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.documents.probe('/doc/example.json').result(
    function(response) {
      if (response.exists) {
        console.log(response.uri + ' exists');
      } else {
        console.log(response.uri + 'does not exist');
      }
    }
);
```

戻り値は、ブール型の値である exists プロパティを含むドキュメントディスクリプタです。REST インスタンスでコンテンツのバージョン管理が有効になっている場合は、レスポンスには versionId プロパティも含まれます。例えば、コンテンツのバージョン管理が有効な場合、上記の例は次の出力を生成します。

```
{
  contentType: "application/json",
  versionId: "14115045710437450",
  format: "json",
  uri: "/doc/example.json",
  exists: true
}
```

コンテンツのバージョン管理の詳細については、「オプティミスティックロックを使用した条件付き更新」（64 ページ）を参照してください。

2.7 オプティミスティックロックを使用した条件付き更新

オプティミスティック（楽観的）ロックを使用するアプリケーションでは、ドキュメントが存在しない場合にのみドキュメントを作成します。また、そのアプリケーションによるドキュメントの最後の変更以降、そのドキュメントが変更されていない場合にのみドキュメントを更新または削除します。ただし、オプティミスティックロックは実際にドキュメントをロックするわけではありません。

オプティミスティックロックは、整合性が重要であっても競合がまれにしか発生しない環境に有用です。不必要なマルチステートメントトランザクションを排除して、サーバー負荷を最小限に抑えることができます。

このセクションでは、次の内容を取り上げます。

- オプティミスティックロックについて
- オプティミスティックロックを有効にする
- バージョン ID を取得する
- 条件付き更新を適用する

2.7.1 オプティミスティックロックについて

例えば、ドキュメントを読み取って修正を行い、その変更内容でデータベースのドキュメントを更新するアプリケーションについて考えてみましょう。ドキュメントの整合性を保証する従来のアプローチでは、読み取り、修正、および更新をマルチステートメントトランザクションで実行します。この場合、ドキュメントが読み取られてから更新がコミットされるまでドキュメントがロックされます。しかしながら、このペシミスティック（悲観的）ロックは、ドキュメントへのアクセスをブロックし、アプリケーションサーバーのオーバーヘッドを増大させます。

オプティミスティックロックでは、アプリケーションはドキュメントが読み取られてから更新されるまでの間、ドキュメントのロックを保持しません。その代わりに、アプリケーションはドキュメントの読み取り時にドキュメントの状態を保存し、変更されているかどうかを更新時にチェックします。ドキュメントが読み取られてから更新されるまでの間にドキュメントが変更されている場合は、更新が失敗します。これが条件付き更新です。

オプティミスティックロックは、整合性が重要であっても競合がまれにしか発生しない環境に有用です。不必要なマルチステートメントトランザクションを排除して、サーバー負荷を最小限に抑えることができます。

Node.js クライアント API は、コンテンツのバージョン管理を使用してオプティミスティックロックを実装します。REST API インスタンスでコンテンツのバージョン管理を有効にすると、MarkLogic サーバーは各ドキュメントが作成または更新されたときにそれに不透明型のバージョン ID を関連付けます。バージョン ID は、ドキュメントを更新するたびに変わります。バージョン ID はドキュメントを読み取ったときに返されます。この ID を更新または削除操作に戻して、コミットする前に変更をテストできます。

> **注：** コンテンツのバージョン管理を有効にしても、ドキュメントのバージョン管理は実装されません。MarkLogic サーバーは、ドキュメントの複数のバージョンを保持したり、発生した変更のトラッキングを行いません。バージョン ID は、変更が発生したことを検出する目的でのみ使用できます。

アプリケーションでオプティミスティックロックを使用するには、次の手順に従う必要があります。

1. REST API インスタンスで、オプティミスティックロックを有効にする。
2. 条件付き更新を行うドキュメントのバージョン ID を取得する。
3. 更新操作にバージョンを含めることで、条件付き更新を適用する。

オプティミスティックロックは、REST API インスタンスプロパティ `update-policy` を設定して有効にします。バージョン ID の送受信は、ドキュメントディスクリプタの `versionId` プロパティを介して行います。

2.7.2 オプティミスティックロックを有効にする

オプティミスティックロックを有効にするには、`DatabaseClient.config.serverprops.write` を呼び出して、`update-policy` プロパティを `version-required` または `version-optional` に設定します。例えば、次のコードは、update-policy を version-optional に設定します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.config.serverprops.write({'update-policy':
'version-optional'})
  .result(function(response) {
      console.log(JSON.stringify(response));
  });
```

すべてのドキュメント更新または削除操作でオプティミスティックロックを使用する場合は、プロパティを version-required に設定します。オプティミスティックロックを選択的に使用できるようにするには、プロパティを version-optional に設定します。

以下の表は、このプロパティの各設定がドキュメント操作にどのように影響するのかを示したものです。

設定	効果
merge-metadata	これがデフォルト設定です。存在しないドキュメントを挿入、更新、または削除した場合に、操作は成功します。バージョン ID を指定した場合、それは無視されます。
version-optional	存在しないドキュメントを挿入、更新、または削除した場合に、操作は成功します。バージョン ID を指定した場合は、そのドキュメントが存在し、現在のバージョン ID が指定したバージョン ID とマッチしない場合に操作が失敗します。
version-required	バージョン ID を指定せずにドキュメントを更新または削除したが、そのドキュメントが存在しない場合に、操作は成功します。ドキュメントが存在する場合は、操作が失敗します。バージョン ID を指定した場合は、そのドキュメントが存在し、現在のバージョン ID がヘッダのバージョンとマッチしない場合に操作が失敗します。
overwrite-metadata	merge-metadata とほぼ同じ動作になりますが、リクエスト内のメタデータが既存のメタデータにマージされるのではなく、リクエスト内のメタデータによって既存のメタデータが上書きされます。この設定ではオプティミスティックロックが無効になります。

2.7.3 バージョン ID を取得する

オプティミスティックロックが有効な場合、ドキュメントを読み取ったときにバージョン ID がレスポンスに含められます。バージョン ID を取得できるのは、`update-policy` を設定してオプティミスティックロックを有効にした場合だけです。詳細については、「オプティミスティックロックを有効にする」（65 ページ）を参照してください。

条件付き更新で使用するバージョン ID は、次の方法で取得できます。

- `DatabaseClient.documents.read` を呼び出す。返されたドキュメントディスクリプタの `versionId` プロパティを通じてバージョン ID を取得できます。
- `DatabaseClient.documents.probe` を呼び出す。ドキュメントは取得されませんが、返されたドキュメントディスクリプタの `versionId` プロパティを通じてバージョン ID を取得できます。

バージョン ID を取得する上記の方法を組み合わせて、マッチさせることができます。

次の例は、複数ドキュメントのバージョンを返します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.documents.read('/doc/example1.json',
'/doc/example2.json').
  .stream().on('data', function(document) {
    console.log('Read ' + document.uri +
       ' with version ' + document.versionId);
  }).on('end', function() {
    console.log('finished');
  });
```

2.7.4 条件付き更新を適用する

条件付き更新を適用するには、更新または削除操作に渡すドキュメントディスクリプタに `versionId` プロパティを含めます。オプティミスティックロックが有効になっていない場合、バージョン ID は無視されます。詳細については、「オプティミスティックロックを有効にする」（65 ページ）を参照してください。

更新または削除操作に渡されるドキュメントディスクリプタにバージョン ID が含まれている場合、MarkLogic サーバーは更新または削除をコミットする前にバージョン ID のマッチを確認します。入力したバージョン ID がドキュメントの現在のバージョン ID とマッチしない場合、操作は失敗します。複数のドキュメントの更新では、バッチ内のいずれかのドキュメントの条件付き更新が失敗すると、更新バッチ全体が拒否されます。

次の例は、各入力ドキュメントディスクリプタに `versionId` プロパティを含めて、2 つのドキュメントの条件付き更新を実行します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.documents.write({
  documents: [
    { uri: '/doc/example1.json',
      contentType: 'application/json',
      content: { data: 1 },
      versionId: 14115098125553360
    },
    { uri: '/doc/example2.json',
      contentType: 'application/json',
      content: { data: 2 },
      versionId: 14115098125553350
    }
  ]
}).result(
  function(success) {
    console.log('Loaded the following documents:');
    for (var i in success.documents)
      console.log(success.documents[i].uri);
  }
);
```

同様に、次の例は、データベース内のドキュメントのバージョン ID が、入力ドキュメントディスクリプタのバージョン ID とマッチする場合のみ、ドキュメント `/doc/example.json` を削除します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.documents.remove({
  uris: ['/doc/example.json'],
```

```
    versionId: 14115105931044000}).
  result(
    function(response) {
      console.log(JSON.stringify(response));
  });
```

注： 単一の操作で複数のドキュメントを削除する場合、条件付き削除は使用できません。

バージョン ID のミスマッチにより条件付き更新または削除が失敗した場合、MarkLogic サーバーは次のようなエラーで応答します（オブジェクトのコンテンツは、読みやすさを考慮して整形されています）。

```
{message: "remove document: response with invalid 412
           status (on /doc/example1.json)",
  statusCode:412,
  body:{
    error: {
      status-code: "412",
      status: "Precondition Failed",
      message-code: "RESTAPI-CONTENTWRONGVERSION",
      message: "RESTAPI-CONTENTWRONGVERSION:
(err:FOER0000)
               Content version mismatch: uri
/doc/example.json
               doesn't match if-match: 14115105931044000"
}}}
```

2.8 バイナリドキュメントを扱う

このセクションでは、MarkLogic サーバーで Node.js API を使用してバイナリドキュメントデータを操作する方法について、簡単に概要を説明します。ここでは、次の内容を取り上げます。

- バイナリドキュメントのタイプ
- バイナリコンテンツのストリーミング
- レンジリクエストによるバイナリコンテンツの取得

2.8.1 バイナリドキュメントのタイプ

このセクションでは、バイナリドキュメントタイプの概要について説明します。詳細については、『Application Developer's Guide』の「Working With Binary Documents」を参照してください。

MarkLogic サーバーでは、バイナリドキュメントを次の 3 つの表現で格納できます。

- スモールバイナリドキュメントは、完全な状態でデータベースに格納されます。
- ラージバイナリドキュメントは、データベースに小さな参照フラグメントがある状態でディスクに格納されます。ディスク上のコンテンツは、MarkLogic サーバーによって管理されます。
- 外部バイナリドキュメントは、データベースに小さな参照フラグメントがある状態でディスクに格納されます。ただし、ディスク上のコンテンツは、MarkLogic サーバーによって管理されません。

ドキュメントを挿入または更新すると、MarkLogic はドキュメントのサイズとデータベースの設定に基づいて、バイナリドキュメントの大小を自動的に判定します。

外部のバイナリドキュメントは Node.js クライアント API を使用して作成できませんが、他のドキュメントと同じように取得できます。

2.8.2 バイナリコンテンツのストリーミング

ストリーミング手法を使用してバイナリコンテンツにアクセスすることにより、MarkLogic サーバーのメモリとクライアントアプリケーションに大きなドキュメントが読み込まれないようにできます。

ストリームを入力ソースとして使用して、バイナリおよび他のデータを MarkLogic にストリーミングできます。詳細については、「データベースへのストリーミング」（25 ページ）を参照してください。

データベースから大きなバイナリドキュメント、または外部のバイナリドキュメントを取得する場合、MarkLogic サーバーは次の条件下で、データベースからコンテンツを自動的にストリーミングします。

- リクエストが、一括読み取りではなく、単一のドキュメントに対するものである。
- 返されたバイナリコンテンツのサイズがバイナリサイズの上限のしきい値を超えている。詳細については、『Application Developer's Guide』の「Working With Binary Documents」を参照してください。
- リクエストがコンテンツのみを対象としていてる。つまり、メタデータは要求されません。
- MIME タイプのコンテンツが、Accept ヘッダまたはドキュメントの URI ファイル拡張子から判断できる。
- コンテンツ変換が適用されない。

レンジリクエストを使用して、上記の制約を満たすバイナリドキュメントを少しずつ取得することもできます。詳細については、「レンジリクエストによるバイナリコンテンツの取得」（71 ページ）を参照してください。

「ストリーム結果処理パターン」（24 ページ）で説明しているチャンクストリームパターンなど、ストリーミング結果のハンドラを使用することによって、ドキュメント全体がクライアントアプリケーションのメモリに読み込まれないようにできます。

2.8.3 レンジリクエストによるバイナリコンテンツの取得

db.documents.read だけを使用してバイナリドキュメントを取得するときの目標は、アプリケーションコードがチャンクで処理する場合でも、最終的にドキュメント全体を取得することにあります。それに対し、レンジリクエストを使用してバイナリドキュメントの一部を取得すると、バイナリドキュメントの一部への再試行可能でランダムなアクセスを行えます。

レンジリクエストを使用するには、取得操作は、「バイナリコンテンツのストリーミング」（70 ページ）で説明している条件を満たす必要があります。取得するバイト範囲を指定するには、`DatabaseClient.documents.read` に渡される呼び出しオブジェクトに「range」プロパティを含めます。

次の例では、URI が `/binaries/large.jpg` であるバイナリドキュメントの最初の 500K を要求します。

```
db.documents.read({
    uris: '/binary/song.m4a',
    range: [0,511999]
  })
```

このような呼び出しで返されるドキュメントディスクリプタは、次のようになります。`contentLength` プロパティは、返されたバイト数を示します。この値は、ソースドキュメントの末尾を超える範囲を要求した場合、要求された範囲のサイズよりも小さくなります。

```
result: [{
  content: {
    type': 'Buffer',
    data': [ theData ]
  },
  uri: '/binary/song.m4a',
  category: [ 'content' ],
  format: 'binary',
  contentLength: '10',
  contentType: 'audio/mp4'
}]
```

2.9 テンポラルドキュメントを扱う

ほとんどのドキュメント書き込み操作では、テンポラルドキュメントを扱えます。テンポラル対応のドキュメント挿入および更新は、`documents.create`、`documents.write`、`documents.patch`、`documents.remove` などの操作で次のパラメータ（またはドキュメントディスクリプタプロパティ）を指定することにより有効になります。

- `temporalCollection`：新しいドキュメントが挿入されるテンポラルコレクションの URI、または更新されているドキュメントを含むテンポラルコレクションの名前。
- `temporalDocument`：テンポラルコレクション内のドキュメントの「論理」URI。テンポラルコレクションドキュメント URI。これは、`temporal:statement-set-document-version-uri` XQuery 関数、または `temporal.statementSetDocumentVersionUri` サーバーサイド JavaScript 関数の最初のパラメータに相当します。
- `sourceDocument`：操作対象のドキュメントのテンポラルコレクションドキュメント URI。既存のドキュメントを更新するときにのみ適用できます。このパラメータを使用すれば、ユーザー保守バージョン URI を持つドキュメントの扱いが容易になります。
- `systemTime`：更新または挿入のシステム開始時間。

更新操作中、`sourceDocument` または `temporalDocument` を指定しなければ、`uri` ディスクリプタプロパティはソースドキュメントを示します。`temporalDocument` を指定しても `sourceDocument` を指定しなければ、`temporalDocument` がソースドキュメントを識別します。

`uri` プロパティは常に、出力ドキュメント URI を参照します。MarkLogic がバージョン URI を管理する場合、ドキュメント URI とテンポラルドキュメントコレクション URI には同じ値が含まれます。ユーザーがバージョン URI を管理する場合、これらを別々にできます。

`documents.protect` を使用して、指定した期間、更新、削除、消去などの操作からテンポラルドキュメントを保護します。この方法は、`temporal:document-protect` XQuery 関数または `temporal.documentProtect` サーバーサイド JavaScript 関数の呼び出しと同等です。

詳細については、『Temporal Developer's Guide』と「Node.js Client API JSDoc」を参照してください。

2.10 メタデータを扱う

Node.js クライアント API を使用すると、メタデータを挿入、更新、取得、およびクエリできます。メタデータとは、ドキュメントのプロパティ、コレクション、パーミッション、およびクオリティです。メタデータは、JSON または XML として操作できます。

このセクションでは、次の内容を取り上げます。

- メタデータのカテゴリ
- メタデータの形式
- ドキュメントのプロパティを扱う
- メタデータのマージの無効化

2.10.1 メタデータのカテゴリ

ドキュメントとそのメタデータを操作する場合、通常、メタデータはドキュメントディスクリプタのプロパティとして表現します。例えば、次のディスクリプタは、入力であるか出力であるかにかかわらず、ドキュメント `/doc/example.json` のコレクションメタデータが含まれています。

```
{ uri: '/doc/example.json',
  collections: ['collection1','collection2']
```

操作によっては、読み取りまたは書き込みを行うドキュメントの部分とそのメタデータを指定するための `categories` プロパティをサポートしています。例えば、次のような `DatabaseClient.documents.read` を呼び出して、`/doc/example.json` に関連付けられているコレクションとクオリティだけを読み取ることができます。

```
db.documents.read({
  uris: ['/doc/example.json'],
  categories: ['collections', 'quality']
})
```

次のカテゴリがサポートされています。

- `collections`
- `permissions`
- `properties`
- `quality`
- `metadataValues`
- `metadata`
- `content`

metadataValues カテゴリは単純なキー / 値メタデータプロパティを表し、「メタデータフィールド」と呼ばれることがあります。このカテゴリには、テンポラルドキュメントの特定のプロパティなどのシステム管理対象メタデータと、ユーザー定義のメタデータの両方を含めることができます。metadataValues におけるプロパティの値は常に、MarkLogic では文字列として格納されます。詳細については、『Administrator's Guide』の「Metadata Fields」を参照してください。

metadata カテゴリは、コレクション、パーミッション、プロパティ、およびクオリティの簡略表現です。DatabaseClient.documents.read などの一部の操作は、コンテンツとメタデータを同時に取得または更新できるように content カテゴリもサポートしています。

2.10.2 メタデータの形式

メタデータは、ドキュメントディスクリプタでは次の形式を取ります。メタデータを書き込みまたはクエリ操作に含めるときに、メタデータのすべてのカテゴリを指定する必要はありません。この構造は、入力および出力メタデータの両方に該当します。

```
{
  "collections" : [ string ],
  "permissions" : [
    {
      "role-name" : string,
      "capabilities" : [ string ]
    }
  ],
  "properties" : {
   property-name : property-value
  },
  "quality" : integer,
  "metadataValues": { key: value, key: value, ...}
}
```

次の例は、URI が /doc/example.json のドキュメントのメタデータを読み取って得られた出力のドキュメントディスクリプタです。このドキュメントは 2 つのコレクションに含まれ、2 つのパーミッション、2 つのドキュメントプロパティ、および 2 つの metadataValues のキー / 値ペアを持ちます。

```
{
  "partType": "attachment",
  "uri": "/doc/example.json",
  "category": "metadata",
  "format": "json",
  "contentType": "application/json",
  "collections": [ "collection1", "collection2" ],
  "permissions": [
```

```
      {
        "role-name": "rest-writer",
        "capabilities": [ "update" ]
      },
      {
        "role-name": "rest-reader",
        "capabilities": [ "read" ]
      }
    ],
    "properties": {
      "prop1": "this is my prop",
      "prop2": "this is my other prop"
    },
    "metadataValues": {
      "key1": "value1",
      "key2": 2
    },
    "quality": 0
  }
```

次の例は、メタデータを XML として示しています。すべての要素は名前空間 `http://marklogic.com/rest-api` 内にあります。`<collection/>`、`<permission/>`、またはプロパティの各要素は、含めなくても、複数含めてもかまいません。`<quality/>` 要素は、1 つだけ含めることができます。各プロパティ要素の要素名とコンテンツは、プロパティによって異なります。

```
<metadata xmlns="http://marklogic.com/rest-api">
  <collections>
    <collection>collection-name</collection>
  </collections>
  <permissions>
    <permission>
      <role-name>name</role-name>
      <capability>capability</capability>
    </permission>
  </permissions>
  <properties>
   <property-element/>
  </properties>
  <quality>integer</quality>
  <metadata-values>
    <metadata-value key="key1">value1</metadata-value>
    <metadata-value key="key2">2</metadata-value>
  <metadata-values>
</metadata>
```

2.10.3 ドキュメントのプロパティを扱う

ドキュメントのプロパティは、メタデータの一種です。ドキュメントのプロパティを使用して、ドキュメントのコンテンツを変更することなく、クエリ可能なユーザー定義データをドキュメントに追加できます。詳細については、『Application Developer's Guide』の「Properties Documents and Directories」を参照してください。

ほとんどの場合、アプリケーションはドキュメントのプロパティと一貫した形式で機能しなければなりません。つまり、JSON としてプロパティを挿入または更新した場合は、それらを JSON として取得およびクエリする必要があり、XML としてプロパティを挿入または更新した場合は、それらを XML として取得およびクエリする必要があります。XML 形式と JSON 形式を混在させると、名前空間の一貫性が損なわれます。

ドキュメントのプロパティは、常に XML としてデータベースに格納されます。JSON として挿入したドキュメントのプロパティは、名前空間 `http://marklogic.com/xdmp/json/basic` で内部的に XML 要素に変換され、JSON プロパティ名とマッチする XML ローカル名が付けられます。例えば、JSON で `{ myProperty : 'value' }` として表したドキュメントのプロパティは、次のような XML 表現になります。

```
<rapi:metadata uri="/doc/example.json" ...>
  <prop:properties
xmlns:prop="http://marklogic.com/xdmp/property">
    <myProperty type="string"
        xmlns="http://marklogic.com/xdmp/json/basic">
      value
    </myProperty>
  </prop:properties>
</rapi:metadata>
```

入力および出力でドキュメントのプロパティに一貫して JSON 表現を使用する限り、この内部表現は透過的です。ただし、XML を使用してドキュメントのプロパティをクエリしたり読み取ったりする場合は、名前空間と内部表現に注意する必要があります。同様に、XML を使用してドキュメントのプロパティを名前空間なしまたは独自の名前空間に挿入できますが、プロパティの JSON 表現に名前空間を反映させることはできません。このため、一貫性のある方法でプロパティを操作することをお勧めします。

1 つだけ例外として挙げられるのは、リソースサービスの拡張および変換など、サーバー上のプロパティを使用して機能するコードです。そのようなコードは、常に XML としてドキュメントのプロパティにアクセスします。

JSON を使用して挿入したユーザー定義のドキュメントのプロパティに基づいてインデックスを設定した場合は、設定で `http://marklogic.com/xdmp/json/basic` 名前空間を使用する必要があります。

last-updated などの保護されているシステムプロパティは、アプリケーションで変更できません。そのようなプロパティの JSON 表現は、キー $ml.prop でオブジェクトにラップします。次に例を示します。

```
{ "properties": {
    "$ml.prop": {
      "last-updated": "2013-11-06T10:01:11-08:00"
    }
} }
```

2.10.4 メタデータのマージの無効化

REST クライアント API を使用して大量のドキュメントを一度に読み込む際に、パフォーマンスが大幅に低下する場合は、メタデータのマージを無効にすることでパフォーマンスを多少向上させることができます。このトピックでは、メタデータのマージを無効にすることによるトレードオフと、無効にする方法について説明します。

- メタデータのマージを無効にすることを検討すべき状況
- メタデータのマージを無効にする方法

2.10.4.1 メタデータのマージを無効にすることを検討すべき状況

メタデータのマージを無効にしてもパフォーマンスはわずかしか向上しません。このため、大量のドキュメントを読み込む場合を除き、これによってパフォーマンスが大幅に向上することはありません。パフォーマンスの向上を体感できる可能性があるのは、次のいずれかの状況です。

- 大量のドキュメントを一度に読み込む場合。
- 同じメタデータを共有する大量のドキュメント、またはデフォルトのメタデータを使用する大量のドキュメントを更新する場合。

更新ポリシー version-optional または version-required を使用しながら、メタデータのマージを無効にすることはできません。

複数のドキュメントの書き込みリクエストの場合、リクエストに特定のドキュメントのコンテンツが含まれている限り、メタデータのマージはデフォルトで無効になります。詳細については、『REST Application Developer's Guide』の「Understanding When Metadata is Preserved or Replaced」を参照してください。

メタデータのマージを無効にする影響の詳細については、『REST Application Developer's Guide』の「Understanding Metadata Merging」を参照してください。

2.10.4.2 メタデータのマージを無効にする方法

メタデータのマージは、update-policy インスタンス設定プロパティによって制御します。デフォルト値は merge-metadata です。

メタデータのマージを無効にするには、「インスタンスプロパティの設定」（283 ページ）で説明する手順を使用して update-policy を overwrite-metadata に設定します。次に例を示します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.config.serverprops.write({'update-policy':
'overwrite-metadata'})
  .result(function(response) {
      console.log(JSON.stringify(response));
  });
```

3.0 ドキュメントコンテンツまたはメタデータへのパッチの適用

この章では、db.documents.patch 関数を使用して、ドキュメントのコンテンツまたはメタデータの中で選択した部分だけを更新する処理に関連する次のトピックについて説明します。

3.1 コンテンツとメタデータへのパッチの適用の概要

部分更新とは、ドキュメント全体またはそのメタデータをすべて置き換えるのではなく、ドキュメントまたはメタデータの一部に適用する更新です。例えば、JSON プロパティや XML 要素を挿入したり、JSON プロパティの値を変更したりすることです。コンテンツの部分更新は、JSON および XML ドキュメントにのみ適用できますが、メタデータの部分更新は、あらゆるドキュメントタイプに適用できます。

パッチとは部分更新のディスクリプタで、JSON または XML で記述されています。パッチでは、どのような更新をどこに適用するのかを MarkLogic サーバーに指示します。パッチで実行できるのは、挿入、置換、置換 - 挿入、および削除の 4 つの操作です（置換 - 挿入操作は、ターゲットコンテンツに少なくとも 1 つのマッチがある場合は置換として機能し、マッチがない場合は挿入として機能します）。

部分更新は、次の操作において使用します。

- 既存のドキュメントで、JSON プロパティ、プロパティ値、または配列項目を追加または削除する。
- 配列項目または JSON プロパティの値を追加、置換、または削除する。
- 既存のドキュメントのメタデータのサブセットを追加、置換、または削除する（パーミッションの変更やドキュメントプロパティの挿入など）。
- MarkLogic サーバーで、ビルトイン関数、カスタム関数、またはユーザー定義関数を使用して置換コンテンツまたはメタデータを動的に生成する。詳細については、「MarkLogic サーバーでの置換データの作成」（116 ページ）を参照してください。

単一のパッチで複数の更新を適用できます。また、同一のパッチでコンテンツとメタデータの両方を更新することもできます。

パッチ操作は、JSON プロパティ、XML 要素および属性、JSON 配列項目などのデータ値、および XML 要素または属性内のデータをターゲットとして実行できます。操作のターゲットは、XPath 式または JSONPath 式を使用して指定します。新しいコンテンツまたはメタデータを挿入する場合は、位置を指定して挿入ポイントを定義します。詳細については、『REST Application Developer's Guide』の「How Position Affects the Insertion Point」を参照してください。

ドキュメントコンテンツにパッチを適用する場合は、パッチ形式がドキュメント形式とマッチしている必要があります（XML ドキュメントの場合は XML パッチ、JSON ドキュメントの場合は JSON パッチ）。ドキュメントタイプが異なるコンテンツにパッチを適用することはできません。ただし、メタデータについては、すべてのドキュメントタイプにパッチを適用できます。メタデータのみのパッチは、XML または JSON のどちらであってもかまいません。コンテンツとメタデータの両方を変更するパッチは、ドキュメントのコンテンツタイプとマッチしている必要があります。

Node.js クライアント API は、パッチを作成および適用する次のインターフェイスを提供します。

- `marklogic.patchBuilder`
- `DatabaseClient.documents.patch`

パッチを適用するには、URI と 1 つあるいは複数のパッチ操作を指定して `DatabaseClient.documents.patch` を呼び出します。パッチ操作を作成するには、`marklogic.patchBuilder` を使用します。

次の例では、JSON ドキュメント `/patch/example.json` にパッチを適用して、`theTop` という名前のプロパティの「最後の子」の位置に `child3` という名前のプロパティを挿入します。完全な例については、「例：JSON プロパティの追加」（81 ページ）を参照してください。

```
var pb = marklogic.patchBuilder;
db.documents.patch('/patch/example.json',
  pb.insert('/object-node("theTop")', 'last-child',
    {child3: 'INSERTED'})
);
```

パッチ操作の作成に関するその他の例については、「パッチの例」（98 ページ）を参照してください。

パッチに複数の操作が含まれている場合、各操作がターゲットドキュメントごとに個別に適用されます。つまり、同じパッチ内で、ある操作が別の操作のコンテキストパスや選択パスの結果、またはコンテキストの変更に影響を与えることはありません。パッチ内の各操作は、マッチするすべてのノードに個別に適用されます。パッチ内のいずれかの操作でエラーが発生して失敗した場合は、パッチ全体が失敗します。

部分更新では、コンテンツ変換は直接サポートされていません。ただし、独自の置換コンテンツ生成関数を実装することで、同じ結果を得ることができます。詳細については、「MarkLogic サーバーでの置換データの作成」（116 ページ）を参照してください。

JSON ドキュメントにパッチを適用する前に、『REST Application Developer's Guide』の「Limitations of JSON Path Expressions」で説明されている制限を熟知しておく必要があります。

3.2　例：JSON プロパティの追加

次の例は、`DatabaseClient.documents.patch` 関数とパッチビルダーを使用して、新しいプロパティを JSON ドキュメントに挿入します。このプログラム例は、次の処理を実行します。

1. URI `/patch/example.json` を指定して、ドキュメントをデータベースに挿入します。

2. このドキュメントにパッチを適用し、`theTop` の「最後の子」の位置に `child3` という新しいプロパティを挿入します。親ノードは、XPath 式を使用して指定します。

3. パッチが適用されたドキュメントをデータベースから読み取って、コンソールに表示します。

`db.documents.patch` 関数は、ドキュメント URI と 1 つあるいは複数の操作を受け付けます。この場合は、「パッチビルダーの `insert` メソッドを呼び出してプロパティを挿入する」という操作を 1 つだけ渡します。

```
pb.insert('/object-node("theTop")',    // path to parent
node
          'last-child',                // insertion position
          {child3: 'INSERTED'})        // content to insert
```

書き込み、パッチ、および読み取りの各操作は、Promise パターンを使用して連結します。詳細については、「Promise 結果処理パターン」（23 ページ）を参照してください。

以下にスクリプトの例を示します。最初の書き込み操作は、このドキュメント例をカプセル化し、実行全体で一貫した動作を得る目的でのみこの例に含まれています。通常、パッチのターゲットとなるドキュメントは、データベースに個別にロードされています。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

// (1) Insert the base document into the database
db.documents.write({
  uri: '/patch/example.json',
  contentType: 'application/json',
  content: {
    theTop: {
      child1: { grandchild: 'gc-value' },
      child2: 'c2-value'
  } }
}).result().then(function(response) {
  // (2) Patch the document
  var pb = marklogic.patchBuilder;
  return db.documents.patch(response.documents[0].uri,
    pb.insert('/object-node("theTop")', 'last-child',
              {child3: 'INSERTED'})
  ).result();
}).then(function(response) {
  // (3) Read the patched document
  return db.documents.read(response.uri).result();
}).then(function(response) {
  console.log(response[0].content);
});
```

このプログラム例により、次のようにドキュメントが変換されます。

パッチ適用前	パッチ適用後

{"theTop" : {
  "child1" : {
  "grandchild" : "gc-value"
  },
  "child2" : "c2-value"
}}
	{"theTop" : {
	  "child1" : {
	  "grandchild" : "gc-value"
	  },
	  "child2" : "c2-value",
	  "child3" : "INSERTED"
	}}

正常に完了すると、DatabaseClient.documents.patch はパッチが適用されたドキュメントの URI を返します。例えば、上記の db.documents.patch 呼び出しの結果は次のとおりです。

```
{ uri: '/patch/example.json' }
```

その他の例については、「パッチの例」（98 ページ）を参照してください。

3.3 パッチリファレンス

DatabaseClient.documents.patch の呼び出しには、挿入、置換、置換 - 挿入、削除操作を 1 つあるいは複数含めることができます。これらの操作は、特定の patch 呼び出しの中で複数回発生させることができます。marklogic.patchBuilder インターフェイスを使用して、各操作を作成します。次に例を示します。

```
db.documents.patch(uri,
  pb.insert(path, position, newContent)
)
```

パッチビルダーには、メタデータをターゲットとするパッチ操作を構築するための特殊用途のインターフェイスが含まれます。patchBuilder.collections、patchBuilder.permissions、patchBuilder.properties、および patchBuilder.quality。これらのインターフェイスを使用すると、メタデータの内部表現を知らなくても、コレクション、パーミッション、クオリティ、およびドキュメントプロパティにパッチを適用できます。これらのインターフェイスを使用する方法の例については、「例：メタデータへのパッチの適用」（110 ページ）を参照してください。

メタデータパッチ操作を構築するための特殊なインタフェースもあります。これらのインターフェイスが、メタデータの内部構造のナビゲーションを処理します。

また、patchBuilder.library および patchbuilder.pathLanguage などのパッチ設定ディレクティブを含めることもできます。library ディレクティブは 1 つだけ含めることができます。これはカスタム関数を使用して置換コンテンツを生成する場合のみ必要です。詳細については、「MarkLogic サーバーでの置換データの作成」（116 ページ）を参照してください。

以下の表は、コンテンツの操作を作成するパッチビルダーのメソッドとその説明をまとめたものです。

ビルダーのメソッド	説明
insert	新しいプロパティまたは配列項目を挿入します。
replace	既存のプロパティまたは配列項目を置換します。
replaceInsert	JSON プロパティまたは配列項目を置換します。マッチするプロパティが存在しない場合は、代わりに挿入を実行します。
remove	JSON プロパティまたは配列項目を削除します。
library	`replace` または `replaceInsert` 操作の一部である `apply` 操作で使用可能なカスタムの置換コンテンツ生成関数が含まれている、サーバーサイドの XQuery ライブラリモジュールを指定します。
apply	サーバーサイドの置換コンテンツ生成関数を指定します。パッチ操作には `library` 操作が含まれている必要があり、そのモジュールには指定した関数が含まれている必要があります。
pathLanguage	`select` 式および `context` 式を XPath 式（デフォルト）または JSONPath 式としてパースするパッチを設定します。

以下の表は、メタデータの操作を作成するパッチビルダーのインターフェイスとその説明をまとめたものです。

ビルダーのメソッド	説明
collections	コレクションメタデータを修正するためのパッチ操作を作成します。
permissions	パーミッションメタデータを修正するためのパッチ操作を作成します。
properties	ドキュメントプロパティメタデータを修正するためのパッチ操作を作成します。
quality	クオリティメタデータを修正するためのパッチ操作を作成します。

以下の表は、設定ディレクティブを作成するパッチビルダーのメソッドとその説明をまとめたものです。

ビルダーのメソッド	説明
library	`replace` または `replaceInsert` 操作の一部である `apply` 操作で使用可能なカスタムの置換コンテンツ生成関数が含まれている、サーバーサイドの XQuery ライブラリモジュールを指定します。
apply	サーバーサイドの置換コンテンツ生成関数を指定します。パッチ操作には `library` 操作が含まれている必要があり、そのモジュールには指定した関数が含まれている必要があります。
pathLanguage	`select` 式および `context` 式を XPath 式（デフォルト）または JSONPath 式としてパースするパッチを設定します。

3.3.1 insert

新しい JSON プロパティまたは配列項目を挿入する操作を作成するには、`patchbuild.insert` を使用します。挿入操作は、次の形式の呼び出しで作成します。

```
marklogic.patchBuilder.insert(
  context,
  position,
  content,
  cardinality)
```

次の表は、patchBuilder.insert 関数のパラメータとその説明をまとめたものです。

パラメータ	必須かどうか	説明
context	必須	操作対象の既存の JSON プロパティまたは配列要素を選択する XPath または JSONPath 式。式では、複数の項目を選択できます。 context 式のマッチが見つからない場合は、エラーが出力されずに操作が無視されます。 パス式は、インデックスを定義するために使用可能な XPath（またはそれと同等の JSONPath）のサブセットに限定されます。詳細については、『REST Application Developer's Guide』の「Path Expressions Usable in Index Definitions」を参照してください。
position	必須	コンテキストの挿入位置。JSON プロパティまたは context によって選択された値との相対的な位置です。pos-selector は、"before"、"after"、または "last-child" のいずれかでなければなりません。詳細については、「位置が挿入ポイントに与える影響」（96 ページ）を参照してください。
content	必須	挿入する新しいコンテンツ。JSON オブジェクト、配列、またはアトミック値として表現します。
cardinality	任意	position に対する必須のマッチ回数。想定された回数をマッチ回数が満たしていない場合は、操作が失敗してエラーが返されます。使用可能な値： • マッチ数がゼロまたは 1 つ："?"（疑問符） • マッチ数が 1 つ："."（ピリオド） • マッチ数がゼロ以上："*"（アスタリスク） • マッチ数が 1 以上："+"（プラス） デフォルト："*"（回数の要件は常に満たされます）。

3.3.2 replace

既存の JSON プロパティ値または配列項目を置換する操作を作成するには、`patchBuilder.replace` を使用します。マッチする JSON プロパティまたは配列項目が存在しない場合は、エラーが出力されずに操作が無視されます。次の形式の呼び出しを使用して `replace` 操作を作成します。

```
marklogic.patchBuilder.replace(
  select,
  content,
  cardinality,
  apply)
```

`apply` を使用すると、動的コンテンツを生成するためのビルトインまたはカスタム定義のコンテンツ生成関数を指定できます。詳細については、「MarkLogic サーバーでの置換データの作成」（116 ページ）を参照してください。

次の表は、`patchBuilder.replace` 関数のパラメータとその説明をまとめたものです。

パラメータ	必須かどうか	説明
`select`	必須	置換する JSON プロパティまたは配列要素を選択する XPath または JSONPath 式。`select` 式のマッチが見つからない場合は、エラーが出力されずに操作が無視されます。 パス式は、インデックスを定義するのに使用可能な XPath（またはそれと同等の JSONPath）のサブセットに限定されます。詳細については、『REST Application Developer's Guide』の「Path Expressions Usable in Index Definitions」を参照してください。 選択した項目をパッチ内のその他の操作のターゲットにすることはできません。選択した項目の祖先を、同じパッチ内の `delete`、`replace`、または `replace-insert` 操作によって変更することはできません。

パラメータ	必須かどうか	説明
content	任意	置換値。このパラメータを省略した場合は、apply パラメータでコンテンツ生成関数を指定する必要があります。 select のターゲットがプロパティで、content がオブジェクトの場合は、ターゲットプロパティ全体が置換されます。それ以外の場合は、値のみが置換されます。
cardinality	任意	select に対する必須のマッチ回数。想定された回数をマッチ回数が満たしていない場合は、操作が失敗してエラーが返されます。使用可能な値： • マッチ数がゼロまたは 1 つ："?"（疑問符） • マッチ数が 1 つ："."（ピリオド） • マッチ数がゼロ以上："*"（アスタリスク） • マッチ数が 1 以上："+"（プラス） デフォルト："*"（回数の要件は常に満たされます）。
apply	任意	置換コンテンツ生成関数のローカル名。関数を指定しない場合は、操作に content パラメータが含まれている必要があります。 カスタム関数を指定した場合は、関数の実装を含む XQuery ライブラリモジュールを表す replace-library 操作がこのパッチに含まれている必要があります。そのような操作を作成するには、patchBuilder.library 関数を使用します。 詳細については、「MarkLogic サーバーでの置換データの作成」（116 ページ）を参照してください。

3.3.3 replaceInsert

プロパティ値または配列項目を置換（プロパティ値または配列項目が存在する場合）または挿入（プロパティ値または配列項目が存在しない場合）する操作を作成するには、patchBuilder.replaceInsert を使用します。次の形式の呼び出しを使用して、replace-insert 操作を作成します。

```
replaceInsert(
  select,
```

```
        context,
        position,
        content,
        cardinality,
        apply)
```

apply を使用して動的コンテンツを生成するためのビルトインまたはカスタムのコンテンツ生成関数を指定する場合は、content を省略できます。詳細については、「MarkLogic サーバーでの置換データの作成」（116 ページ）を参照してください。

ターゲットの選択に使用できるパス式には制限があるため、現在のところこの操作が配列項目で最も役立ちます。詳細については、『REST Application Developer's Guide』の「Limitations of JSON Path Expressions」を参照してください。

次の表は、replace-insert 操作のパラメータとその説明をまとめたものです。

パラメータ	必須かどうか	説明
select	必須	置換する JSON プロパティまたは配列項目を選択する XPath または JSONPath 式。select 式のマッチが見つからない場合は、context と position を使用して挿入が試行されます。また、context のマッチが見つからない場合は、何も処理は行われません。 パス式は、インデックスを定義するのに使用可能な XPath（またはそれと同等の JSONPath）のサブセットに限定されます。詳細については、『REST Application Developer's Guide』の「Path Expressions Usable in Index Definitions」を参照してください。 選択した項目をパッチ内のその他の操作のターゲットにすることはできません。選択した項目の祖先を、同じパッチ内の delete、replace、または replace-insert 操作によって変更することはできません。

パラメータ	必須かどうか	説明
content	任意	選択した値を置き換えるのに使用するコンテンツ。content が存在しない場合は、apply を使用してコンテンツ生成関数を指定する必要があります。 select のターゲットがプロパティで、content がオブジェクトの場合は、ターゲットプロパティ全体が置換されます。それ以外の場合は、値のみが置換されます。
context	必須	操作対象の既存のプロパティまたは配列要素を選択する XPath または JSONPath 式。式では、複数の項目を選択できます。 select または context 式のマッチが見つからない場合は、エラーが出力されずに操作が無視されます。 パス式は、インデックスを定義するのに使用可能な XPath（またはそれと同等の JSONPath）のサブセットに限定されます。詳細については、『REST Application Developer's Guide』の「Path Expressions Usable in Index Definitions」を参照してください。 選択したノードの祖先を、同じパッチ内の delete、replace、または replace-insert 操作によって変更することはできません。
position	任意	select のマッチが見つからない場合の、コンテキストの挿入位置です。キー / バリューペアまたは context によって選択された値との相対的な位置です。*pos-selector* は、"before"、"after"、または "last-child" のいずれかでなければなりません。詳細については、「位置が挿入ポイントに与える影響」（96 ページ）を参照してください。 デフォルト：last-child。

パラメータ	必須かどうか	説明
cardinality	任意	position に対する必須のマッチ回数。想定された回数をマッチ回数が満たしていない場合は、操作が失敗してエラーが返されます。使用可能な値： • マッチ数がゼロまたは 1 つ："?"（疑問符） • マッチ数が 1 つ："."（ピリオド） • マッチ数がゼロ以上："*"（アスタリスク） • マッチ数が 1 以上："+"（プラス） デフォルト：*（回数の要件が常に満たされる）。
apply	任意	置換コンテンツ生成関数のローカル名。関数を指定しない場合は、操作に content パラメータが含まれている必要があります。 カスタム関数を指定した場合は、関数の実装を含む XQuery ライブラリモジュールを表す replace-library 操作がこのパッチに含まれている必要があります。そのような操作を作成するには、patchBuilder.library 関数を使用します。 詳細については、「MarkLogic サーバーでの置換データの作成」（116 ページ）を参照してください。

3.3.4 remove

JSON プロパティまたは配列要素を削除する操作を作成するには、patchBuilder.remove を使用します。patchBuilder.remove 関数の呼び出しの形式は、次のとおりです。

```
marklogic.patchBuilder.remove(
  select,
  cardinality)
```

次の表は、patchBuilder.remove のパラメータとその説明をまとめたものです。

コンポーネント	必須かどうか	説明
select	必須	削除する JSON プロパティまたは配列要素を選択する XPath または JSONPath 式。select 式のマッチが見つからない場合は、エラーが出力されずに操作が無視されます。 パス式は、インデックスを定義するのに使用可能な XPath（またはそれと同等の JSONPath）のサブセットに限定されます。詳細については、『REST Application Developer's Guide』の「Path Expressions Usable in Index Definitions」を参照してください。 選択した項目をパッチ内のその他の操作のターゲットにすることはできません。選択した項目の祖先を、同じパッチ内の delete、replace、または replace-insert 操作によって変更することはできません。
cardinality	任意	select に対する必須のマッチ回数。想定された回数をマッチ回数が満たしていない場合は、操作が失敗してエラーが返されます。使用可能な値： • マッチ数がゼロまたは 1 つ："?"（疑問符） • マッチ数が 1 つ："."（ピリオド） • マッチ数がゼロ以上："*"（アスタリスク） • マッチ数が 1 以上："+"（プラス） デフォルト："*"（回数の要件は常に満たされます）。

3.3.5 apply

replace または replaceInsert 操作の置換コンテンツを生成するのに使用するサーバーサイド関数を指定する設定ディレクティブを作成するには、patchBuilder.apply を使用します。指定する関数は、db.documents.patch の同じ呼び出しに含まれている library ディレクティブによって指定されたサーバーサイドモジュールに配置されていなければなりません。

patchBuilder.apply 関数の呼び出しの形式は、次のとおりです。

```
marklogic.patchBuilder.apply(functionName)
```

詳細については、「MarkLogic サーバーでの置換データの作成」（116 ページ）を参照してください。

3.3.6 library

置換コンテンツを生成するための 1 つあるいは複数の関数が含まれたサーバーサイドライブラリモジュールを指定する設定ディレクティブを作成するには、patchBuilder.library を使用します。このモジュールは、「カスタム置換コンストラクタの記述」（121 ページ）で説明されている規則に準拠している必要があります。ライブラリで関数を使用するには、生成したライブラリディレクティブを db.documents.patch の呼び出しに含めて、replace または replaceInsert ビルダー関数を呼び出すときに patchBuilder.apply からの出力を含めます。

patchBuilder.library 関数の呼び出しの形式は、次のとおりです。

```
marklogic.patchBuilder.library(moduleName)
```

詳細については、「MarkLogic サーバーでの置換データの作成」（116 ページ）を参照してください。

3.3.7 pathLanguage

デフォルトでは、パッチ内のすべてのパス式が XPath を使用して表現されます。以前のリリースとの互換性を確保するために JSONPath もサポートされていますが、使用することは推奨されていません。JSONPath を使用しない限り、pathLanguage ディレクティブを使用する必要はありません。

パッチ操作でコンテキストを表して、式を選択するパス言語を指定する設定ディレクティブを作成するには、patchBuilder.pathLanguage を使用します。patchBuilder.pathLanguage 関数の呼び出しは、次のいずれかの形式でなければなりません。

```
marklogic.patchBuilder.pathLanguage('xpath')

marklogic.patchBuilder.pathLanguage('jsonpath')
```

結果のディレクティブを db.documents.patch の呼び出しに含めます。

DatabaseClient.documents.patch 呼び出しのすべての操作で、同じパス言語を使用する必要があります。

3.3.8 collections

コレクションメタデータのパッチ操作を作成するには、patchBuilderCollections メソッドを使用します。これらのメソッドにアクセスするには、patchBuiler.collections を使用します。次のメソッドを使用して、コレクションを追加または削除できます。

```
marklogic.patchBuilder.collections.addCollection(collName)

marklogic.patchBuilder.collections.removeCollection
(collName)
```

例については、「例：メタデータへのパッチの適用」（110 ページ）を参照してください。

3.3.9 permissions

パーミッションメタデータのパッチ操作を作成するには、patchBuilderPermissions メソッドを使用します。これらのメソッドにアクセスするには、patchBuilder.permissions を使用します。このインターフェイスを使用して、パーミッションの追加や削除、またはロールの機能の置換を行えます。

```
marklogic.patchBuilder.permissions.addPermission(role,
capabilities)

marklogic.patchBuilder.permissions.removePermission(role)

marklogic.patchBuilder.permissions.replacePermission(role,
capabilities)
```

ここで *role* はロール名を含む文字列であり、*capabilities* は「read」、「insert」、「update」、「execute」の値を取り得る、単一の文字列または文字列の配列です。

replacePermission はロールに関連付けられたすべての機能を置換することに注意してください。単一の機能を選んで置換する場合には、使用できません。むしろ、ロールおよび関連した機能を単一のユニットとして操作する必要があります。

例については、「例：メタデータへのパッチの適用」（110 ページ）を参照してください。

3.3.10 properties

ドキュメントプロパティメタデータのパッチ操作を作成するには、patchBuilderProperties メソッドを使用します。これらのメソッドにアクセスするには、patchBuilder.properties を使用します。プロパティの追加や削除、またはプロパティの値の置換を行えます。

```
marklogic.patchBuilder.properties.addProperty(name, value)

marklogic.patchBuilder.properties.removeProperty(name)

marklogic.patchBuilder.properties.replaceProeprty(name,
value)
```

例については、「例：メタデータへのパッチの適用」（110 ページ）を参照してください。

3.3.11 quality

クオリティメタデータの値を設定するパッチ操作を作成するには、`patchBuilderQuality` メソッドを使用します。これらのメソッドにアクセスするには、`patchBuilder.quality` を使用します。

```
marklogic.patchBuilder.quality.set(value)
```

例については、「例：メタデータへのパッチの適用」（110 ページ）を参照してください。

3.4 パッチ操作のコンテキストの定義

コンテンツまたはメタデータを挿入、置換、または削除する場合は、操作対象の JSON または XML コンポーネントを MarkLogic サーバーに通知するのに十分なコンテキストがパッチ定義に含まれている必要があります。例えば、削除対象となる JSON プロパティや XML 要素、新しいプロパティや要素の挿入位置、置換対象となる JSON プロパティや XML 要素、値などが含まれている必要があります。

ビルダーを使用してパッチを作成する場合は、`DocumentPatchBuilder.insertFragment()` や `DocumentPatchBuilder.replaceValue()` などのビルダーメソッドの *contextPath* および *selectPath* パラメータを使用してコンテキストを指定します。RAW モードの XML または JSON からパッチを作成する場合は、`context` および `select` の各 XML 属性または JSON プロパティを使用して操作コンテキストを指定します。

操作コンテキストを定義するには、XPath 式を使用します。XPath 式は、パスレンジインデックスを定義するのに使用できる XPath のサブセットに制限されます。詳細については、『REST Application Developer's Guide』の「Path Expressions Usable in Index Definitions」を参照してください。

挿入操作には、新しいコンテンツの挿入位置を、選択したノードの前または後など、コンテキストとの相対的な位置として定義する位置パラメータ / プロパティが追加で用意されています。詳細については、「位置が挿入ポイントに与える影響」（96 ページ）を参照してください。

3.5 位置が挿入ポイントに与える影響

insert および replace-insert パッチ操作には、コンテキスト式と組み合わせたときの挿入位置を定義する位置パラメータが含まれています。このセクションでは、位置が挿入ポイントに与える影響の詳細について説明します。

このトピックでは、JSON ドキュメントでのパッチ適用について取り上げます。XML の詳細については、『REST Application Developer's Guide』の「Specifying Position in XML」を参照してください。

PatchBuilder 操作の位置パラメータ（または、RAW モードのパッチの position プロパティ）は、新しいコンテンツを、コンテキスト XPath 式によって選択された項目から相対的に見てどこに挿入するのかを指定します。位置は、次のいずれかの値で指定できます。

- before：context によって選択された項目の前に挿入します。
- after：context によって選択された項目の後ろに挿入します。
- last-child：ターゲット配列またはオブジェクトの最後の子の位置に挿入します。context によって選択される値は、オブジェクトまたは配列ノードでなければなりません。

insert 操作または replace-insert 操作のどちらでも使用方法は同じです。

次の表は、context と position の組み合わせの例と、新しいコンテンツの結果の挿入ポイントを示したものです。挿入ポイントは *** で示しています。

コンテキスト	位置	挿入ポイントの例
/theTop/node("child1") プロパティ	after	{"theTop" : { "child1" : ["val1", "val2"], *** "child2" : [{"one": "val1"}, {"two": "val2"}] } }

コンテキスト	位置	挿入ポイントの例
/theTop/child1[1] 配列項目	before	{"theTop" : { "child1" : [*** "val1", "val2"], "child2" : [{"one": "val1"}, {"two": "val2"}] } }
/theTop/array-node ("child1") 配列	last- child	{"theTop" : { "child1" : ["val1", "val2" ***], "child2" : [{"one": "val1"}, {"two": "val2"}] } }
/node("theTop") オブジェクト	last- child	{"theTop" : { "child1" : ["val1", "val2"], "child2" : [{"one": "val1"}, {"two": "val2"}] *** } }
/theTop/child1 値	last- child	選択した値がオブジェクトまたは配列でないためエラーになります。例えば、ターゲットドキュメントのコンテンツが次の場合はエラーになります。 {"theTop" : { "child1" : "val1", "child2" : [...] } }

JSON での XPath の使用方法の詳細については、『Application Developer's Guide』の「Traversing JSON Documents Using XPath」を参照してください。

3.6 パッチの例

このセクションでは、パッチを作成してメタデータと JSON ドキュメントに適用する例について説明します。ここでは、次の内容を取り上げます。

- 例の実行準備
- 例：Insert
- 例：Replace
- 例：ReplaceInsert
- 例：Remove
- 例：メタデータへのパッチの適用

XML ドキュメントにパッチを適用するには、RAW モードのパッチを作成する必要があります。詳細については、「XML ドキュメントへのパッチの適用」（115 ページ）および「ビルダーを使用せずにパッチを作成する」（113 ページ）を参照してください。

3.6.1 例の実行準備

データベース接続情報を除き、この例には必要な情報がすべて含まれています。
各例は、スクリプト例と同じ場所に配置された `my-connection.js` という名前のモジュールに接続情報がカプセル化されていることを前提としています。このモジュールには、次のようなコンテンツが含まれている必要があります。

```
module.exports = {
  connInfo: {
    host: 'localhost',
    port: 8000,
    user: your-ml-username,
    password: your-ml-user-password
  }
};
```

詳細については、「このガイドの例の使用方法」（33 ページ）を参照してください。

3.6.2 例：Insert

この例では、ベースの JSON ドキュメントに次の変更を加えることで、挿入操作の実行方法を示します。

- 名前なしルートオブジェクトの新しいプロパティを挿入する（`INSERTED1`）。
- 兄弟プロパティと相対的なサブオブジェクトの位置に新しいプロパティを挿入する（`INSERTED2`）。

- 特定の値を持つ項目との相対的な位置に新しい配列項目を挿入する（INSERTED3）。
- 配列の特定の位置との相対的な位置に新しい配列項目を挿入する（INSERTED4）。
- 配列の末尾に新しい項目を挿入する（INSERTED5）。
- 新しいプロパティを「最後の子」の位置に挿入する（INSERTED6）。

この例では、まずベースドキュメントをデータベースに挿入し、パッチを作成して適用します。その後、変更されたドキュメントがデータベースから取得され、コンソールに表示されます。この例では Promise パターンを使用してこれらの操作を順に実行します。詳細については、「Promise 結果処理パターン」（23 ページ）を参照してください。

この例を実行するには、次のコードをファイルにコピーして、node コマンドで指定します。次に例を示します。node insert.js とします。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

// (1) Insert the base document into the database
db.documents.write({
  uri: '/patch/insert.json',
  contentType: 'application/json',
  content: {
    theTop: {
      child1: { grandchild: 'gcv' },
      child2: [ 'c2v1', 'c2v2' ],
      child3: [ {c3k1: 'c3v1'}, {c3k2: 'c3v2'} ]
  } }
}).result().then(function(response) {
  // (2) Patch the document
  var pb = marklogic.patchBuilder;
  return db.documents.patch(response.documents[0].uri,
    // insert a sibling of theTop
    pb.insert('/theTop', 'before',
              {INSERTED1: [ 'i1v1', 'i1v2']}),

    // insert a property in child1's value
    pb.insert('/theTop/child1/grandchild', 'before',
              {INSERTED2: 'i2v'}),

    // insert an array item in child2 by value
    pb.insert('/theTop/child2[.= "c2v1"]', 'after',
              'INSERTED3'),
```

```
        // insert an array item in child2 by position
        pb.insert('/theTop/child2[2]', 'after',
                  'INSERTED4'),

        // insert an object in child3
        pb.insert('/theTop/array-node("child3")', 'last-child',
                  {INSERTED5 : 'i5v'}),

        // insert a new child of theTop
        pb.insert('/theTop', 'last-child',
                  {INSERTED6: 'i6v'})
      ).result();
    }).then(function(response) {
      // (3) Emit the resulting document
      return db.documents.read(response.uri).result();
    }).then(function(documents) {
      console.log(JSON.stringify(documents[0].content));
    });
```

次の表は、パッチ適用後のターゲットドキュメントの変化を示したものです。

更新前	更新後

{"theTop": {
"child1": {"grandchild":
"gcv"},
"child2": ["c2v1",
"c2v2"],
"child3": [
{"c3k1": "c3v1"},
{"c3k2": "c3v2"}
]
} }
	{ "INSERTED1": ["i1v1", "i1v2"
	],
	"theTop": {
	"child1": {
	"INSERTED2": "i2v",
	"grandchild": "gcv"
	},
	"child2": [
	"c2v1",
	"INSERTED3",
	"c2v2",
	"INSERTED4"
	],
	"child3": [
	{ "c3k1": "c3v1" },
	{ "c3k2": "c3v2" }
	{ "INSERTED5": "i5v" }
	],
	"INSERTED6": "i6v"
	} }

位置として last-child を使用して挿入を行う場合は、挿入コンテキストの XPath 式では、格納しているオブジェクトまたは配列を参照する必要がある点に注意してください。/a/b などの XPath 式を作成すると、式はコンテナではなく値（または値のシーケンス）を参照します。コンテナを参照するには、node() および array-node() 演算子を使用します。以下に示すように、child3 の下に INSERTED5 を作成する操作で array-node("child3") を使用するのはこのためです。

```
pb.insert('/theTop/array-node("child3")', 'last-child',
                {INSERTED5 : 'i5v'})
```

挿入コンテキスト式を /theTop/child3 に変更すると、子を追加する配列オブジェクトではなく、配列内の値のシーケンスを参照するようになります。

JSON での XPath の使用方法の詳細については、『Application Developer's Guide』の「Traversing JSON Documents Using XPath」を参照してください。

3.6.3　例：Replace

この例では、JSON ドキュメントに次の変更を加えることで、置換操作の実行方法を示します。

- 単純な値（child1）、オブジェクト値（child2）、配列値（child3）でプロパティの値を置換する。
- 位置または値で配列項目の値を置換する（child4）。
- 配列内のオブジェクトの値をプロパティ名で置換する（child5）。
- オブジェクトである配列項目をプロパティ名で置換する（child5）。

この例では、まずベースドキュメントをデータベースに挿入し、パッチを作成して適用します。その後、変更されたドキュメントがデータベースから取得され、コンソールに表示されます。この例では Promise パターンを使用してこれらの操作を順に実行します。詳細については、「Promise 結果処理パターン」（23 ページ）を参照してください。

この例を実行するには、次のコードをファイルにコピーして、node コマンドで指定します。次に例を示します。node replace.js とします。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

// (1) Insert the base document into the database
db.documents.write({
  uri: '/patch/replace.json',
  contentType: 'application/json',
```

```
    content: {
      theTop: {
        child1: 'c1v',
        child2: { gc: 'gcv' },
        child3: [ 'c3v1', 'c3v2' ],
        child4: [ 'c4v1', 'c4v2' ],
        child5: [ {gc1: 'gc1v'}, {gc2:'gc2v'}]
    } }
  }).result().then(function(response) {
    // (2) Patch the document
    var pb = marklogic.patchBuilder;
    return db.documents.patch(response.documents[0].uri,
      // replace the simple value of a property
      pb.replace('/theTop/child1', 'REPLACED1'),

      // replace a property value that is an object
      pb.replace('/theTop/child2', {REPLACE2: 'gc2'}),

      // replace the value of a property that is an array
      pb.replace('/theTop/array-node("child3")',
                 ['REPLACED3a', 'REPLACED3b']),

      // replace an array item by position
      pb.replace('/theTop/child4[1]', 'REPLACED4a'),

      // replace an array item by value
      pb.replace('/theTop/child4[.="c4v2"]', 'REPLACED4b'),

      // replace the value of a property in an array item
      pb.replace('/theTop/child5/gc1', 'REPLACED5a'),

    // replace an object-valued array item by property name
    pb.replace('/theTop/child5[gc2]', {REPLACED5b: '5bv'})

  ).result();
  }).then(function(response) {
    // (3) Emit the resulting document
    return db.documents.read(response.uri).result();
  }).then(function(documents) {
    console.log(JSON.stringify(documents[0].content, null,
  2));
  });
```

次の表は、パッチ適用後のターゲットドキュメントの変化を示したものです。

更新前	更新後

{"theTop": {
  "child1": "c1v",
  "child2": {gc: "gcv"},
  "child3": ["c3v1",
"c3v2"],
  "child4": ["c4v1",
"c4v2"],
  "child5": [
  {gc1: "gc1v"},
  {gc2: "gc2v"}
  ]
} }
	{"theTop": {
	  "child1": "REPLACED1",
	  "child2": {"REPLACED2": "gc2"},
	  "child3": ["REPLACED3a",
	"REPLACED3b"],
	  "child4": ["REPLACED4a",
	"REPLACED4b"],
	  "child5": [
	  { "gc1": "REPLACED5a" },
	  { "REPLACED5b": "5bv" }
	  ]
	} }

コンテナを選択することと、そのコンテンツを選択することの違いを理解することは重要です。例えば、child5 に適用された 2 つの置換操作について考えてみましょう。XPath 式 /theTop/child5/gc1 は、gc1 という名前のプロパティ値に対応します。このため、置換操作によって次のように変更されます。

```
pb.replace('/theTop/child5/gc1', 'REPLACED5a')

{"gc1" : "gc1v"} ==> {"gc1" : "REPLACED5a"}
```

それに対して、XPath 式 /theTop/child5[gc2] は、gc2 という名前のプロパティが含まれているオブジェクトノードを選択します。このため、置換操作によってオブジェクト全体が新しい値で置換され、結果として次のように変更されます。

```
pb.replace('/theTop/child5[gc2]', {REPLACED5b: '5bv'})

{"gc2" : "gc2v"} ==> {"REPLACED5b": "5bv"}
```

同様に、child3 での置換操作について考えてみましょう。XPath 式 /theTop/array-node("child3") は、child3 という名前の配列ノードを選択するため、操作によって配列全体が新しい値で置換されます。次に例を示します。

```
pb.replace('/theTop/array-node("child3")',
              ['REPLACED3a', 'REPLACED3b'])

"child3": ["c3v1", "c3v2"] ==> "child3": ["REPLACED3a",
"REPLACED3b"]
```

それに対して、/theTop/child3 などの XPath 式は、配列（'c3v1', 'c3v2'）内の値を選択するため、この選択式での置換操作により、配列の各値が同じコンテンツで置換されます。次に例を示します。

```
pb.replace('/theTop/child3', 'REPLACED')

"child3": ["c3v1", "c2v2"] ==> "child3": ["REPLACED",
"REPLACED"]
```

JSON での XPath の使用方法の詳細については、『Application Developer's Guide』の「Traversing JSON Documents Using XPath」を参照してください。

3.6.4　例：ReplaceInsert

この例では、replace-insert パッチ操作の実行方法を示します。この例では、次の更新操作を実行します。

- 配列項目を位置によって置換または挿入する（child1）。
- 配列項目を値によって置換または挿入する（child2）。
- オブジェクト値配列項目を、含まれているプロパティ名によって置換または挿入する（child3）。

この例では、まずベースドキュメントをデータベースに挿入し、パッチを作成して適用します。その後、変更されたドキュメントがデータベースから取得され、コンソールに表示されます。この例では Promise パターンを使用してこれらの操作を順に実行します。詳細については、「Promise 結果処理パターン」（23 ページ）を参照してください。

この例を実行するには、次のコードをファイルにコピーして、node コマンドで指定します。次に例を示します。node replace-insert.js とします。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

// (1) Insert the base document into the database
db.documents.write({
  uri: '/patch/replace-insert.json',
  contentType: 'application/json',
  content: {
    theTop: {
      child1: [ 'c1v1', 'c1v2' ],
      child2: [ 'c2v1', 'c2v2' ],
      child3: [ { c3a: 'c3v1' }, { c3b: 'c3v2' } ]
  } }
```

```
}).result().then(function(response) {
  // (2) Patch the document
  var pb = marklogic.patchBuilder;
  return db.documents.patch(response.documents[0].uri,
    // Replace the value of an array item by position, or
    // insert a new one in the target position.
    pb.replaceInsert('/theTop/child1[1]',
              '/theTop/array-node("child1")', 'last-child',
              'REPLACED1'),
    pb.replaceInsert('/theTop/child1[3]',
'/theTop/child1[2]', 'after',
              'INSERTED1'),

    // Replace an array item that has a specific value, or
    // insert a new item with that value at the end of the
array
    pb.replaceInsert('/theTop/child2[.= "c2v1"]',
              '/theTop/node("child2")', 'last-child',
              'REPLACED2'),
    pb.replaceInsert('/theTop/child2[.= "INSERTED2"]',
              '/theTop/array-node("child2")', 'last-child',
              'INSERTED2'),

   // Replace the value of an object that is an array item,
or
    // insert an equivalent object at the end of the array
    pb.replaceInsert('/theTop/child3[c3a]',
              '/theTop/node("child3")', 'last-child',
              { REPLACED3: 'c3rv'}),
    pb.replaceInsert('/theTop/child3[INSERTED3]',
              '/theTop/node("child3")', 'last-child',
              { INSERTED3: 'c3iv'})
  ).result();
}).then(function(response) {
  // (3) Emit the resulting document
  return db.documents.read(response.uri).result();
}).then(function(documents) {
  console.log(JSON.stringify(documents[0].content, null,
2));
}, function(error) { console.log(error); throw error; }
);
```

次の表は、パッチ適用後のターゲットドキュメントの変化を示したものです。

更新前	更新後

{"theTop": {
  "child1": ["c1v1", "c1v2"],
  "child2": ["c2v1", "c2v2"],
  "child3": [
  { "c3a": "c3v1" },
  { "c3b": "c3v2" }
  ]
} }
	{ "theTop": {
	  "child1": [
	  "REPLACED1",
	  "c1v2",
	  "INSERTED1"
	  ],
	  "child2": [
	  "REPLACED2",
	  "c2v2",
	  "INSERTED2"
	  ],
	  "child3": [
	  { "REPLACED3": "c3rv" },
	  { "c3b": "c3v2" },
	  { "INSERTED3": "c3iv" }
	  ]
	} }

前述したように、`select` パスは置換するコンテンツを示します。配列項目を操作する場合は、通常、絶対パスを使用する必要があります。例えば、次のパッチ操作について考えてみましょう。

```
pb.replaceInsert('/theTop/child1[1]',
                 '/theTop/array-node("child1")', 'last-child',
                 'REPLACED1')
```

この操作の目的は、/theTop/child1 の配列値の最初の項目の値が存在する場合に、それを置換することです。配列が空の場合は、新しい値を挿入します。つまり、次のいずれかの変換が実行されます。

```
{"theTop": {"child1": ["c1v1", "c1v2"], ...}
    ==> {"theTop": {"child1": ["REPLACED1", "c1v2"], ...}

{"theTop": {"child1": [], ...}
    ==> {"theTop": {"child1": ["REPLACED1"], ...}
```

select 式 /theTop/child1[1] は、配列項目値をターゲットにする必要があります。ただし context 式では、/theTop/array-node("child1") を参照することにより、格納する配列ノードをターゲットにする必要があります。この場合、select 式を context 式と相対的にすることはできません。

replace-insert で配列項目値全体をターゲットにすることはできますが、プロパティの値だけをターゲットにすることはできません。例えば、JSON の次の配列について考えてみましょう。

```
"child3": [
  { "c3a": "c3v1" },
  { "c3b": "c3v2" }
]
```

例で実行したように、replace-insert をオブジェクト値項目 { "c3a": "c3v1" } 全体で使用できます。ただし、オブジェクト内のプロパティの値だけをターゲットにすることはできません ("c3v1")。プロパティ値の置換は、配列に新しいオブジェクトを挿入する処理とは根本的に異なります。格納するオブジェクトとは異なり、プロパティは、それを削除して新しい値を挿入することでのみ置換できます。

3.6.5 例：Remove

この例では、パッチ削除（delete）操作を使用して JSON ドキュメントに次の変更を加える方法を示します。

- タイプに基づいてプロパティを削除する。
- 位置、値、またはプロパティ名で配列項目を削除する。
- 配列内のすべての項目を削除する。

この例を実行するには、次のコードをファイルにコピーして、node コマンドで指定します。次に例を示します。node remove.js とします。

```
var marklogic = require('marklogic');
var my = require('../my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

// (1) Insert the base document into the database
db.documents.write({
  uri: '/patch/remove.json',
  contentType: 'application/json',
  content: {
    props: {
      anyType: [1, 2],
      objOrLiteral: 'anything',
      arrayVal: [3, 4]
    },
```

```
    arrayItems: {
      byPos: ['DELETE', 'PRESERVE'],
      byVal: ['DELETE', 'PRESERVE'],
      byPropName: [ {DELETE: 5}, {PRESERVE: 6} ],
      all: ['DELETE1', 'DELETE2']
    }
  }
}).result().then(function(response) {
  // (2) Patch the document
  var pb = marklogic.patchBuilder;
  return db.documents.patch(response.documents[0].uri,
    // Remove a property with any value type
    pb.remove('/props/node("anyType")'),

    // Remove a property with atomic or object value type
    pb.remove('/props/objOrLiteral'),

    // Remove a property with array value type
    pb.remove('/props/array-node("arrayVal")'),

    // Remove all items in an array
    pb.remove('/arrayItems/all'),

    // Remove an array item by position
    pb.remove('/arrayItems/byPos[1]'),

    // Remove an array item by value
    pb.remove('/arrayItems/byVal[.= "DELETE"]'),

    // Remove an object-valued array item by property name
    pb.remove('/arrayItems/byPropName["DELETE"]')
  ).result();
}).then(function(response) {
  // (3) Read the patched document
  return db.documents.read(response.uri).result();
}).then(function(documents) {
  console.log(JSON.stringify(documents[0].content, null,
2));
}, function(error) {
  console.log(error); throw error;
});
```

次の表は、パッチ適用後のターゲットドキュメントの変化を示したものです。

更新前	更新後

{ "props": {
"anyType": [1, 2],
"objOrLiteral": "anything",
"arrayVal": [3, 4]
},
"arrayItems": {
"byPos": ["DELETE",
"PRESERVE"],
"byVal": ["DELETE",
"PRESERVE"],
"byPropName": [
{"DELETE": 5},
{"PRESERVE": 6}
],
"all": ["DELETE1",
"DELETE2"]
}
}
	{ "props": { },
	"arrayItems": {
	"byPos": ["PRESERVE"],
	"byVal": ["PRESERVE"],
	"byPropName": [
	{ "PRESERVE": 6 }
	],
	"all": [ ]
	}
	}

プロパティを削除する場合は、指定されたノードの手順を使用してターゲットプロパティを指定するか、値のタイプを把握している必要がある点に注意してください。プログラム例の次の 3 つの操作について考えてみましょう。

```
pb.remove('/props/node("anyType")'),
pb.remove('/props/objOrLiteral'),
pb.remove('/props/array-node("arrayVal")')
```

最初の操作は、タイプに依存しない XPath 式 `/props/node("anyType")` を使用します。この式は、値のタイプにかかわらず、`anyType` という名前のあらゆるノードを選択します。2 番目の操作は、格納する配列ノードではなく、配列項目値を選択する XPath 式 `/props/objOrLiteral` を使用します。つまり、この操作を元のドキュメントに適用すると、`arrayVal` プロパティではなく、配列のコンテンツが削除されます。

```
pb.remove('/props/arrayVal')
==> "arrayVal": [ ]
```

3 番目の形式 `/props/array-node("arrayVal")` は、`arrayVal` プロパティを削除しますが、配列タイプを持つプロパティでのみ機能します。そのため、タイプにかかわらず名前によってプロパティを削除する必要がある場合は、形式が `/path/to/parent/node("propName")` の XPath 式を使用してください。

JSON ドキュメントモデルと、XPath で JSON ドキュメントをトラバースする方法の詳細については、『Application Developer's Guide』の「Working With JSON」を参照してください。

3.6.6 例：メタデータへのパッチの適用

この例では、ドキュメントメタデータにパッチを適用する方法を示します。この例では、次のパッチ操作を実行します。

- ドキュメントをコレクションに追加する。
- ドキュメントをコレクションから削除する。
- パーミッションを変更する。
- メタデータプロパティを追加する。
- クオリティを設定する。

この例では、`patchBuilder` のメタデータパッチ適用ヘルパーインターフェイス（`patchBuilderCollections`、`patchBuilderPermissions`、`patchBuilderProperties`、および `patchBuilder` クオリティ）を使用します。メタデータに直接パッチを適用することもできますが、ヘルパーインターフェイスを使用すれば、MarkLogic 内部でのメタデータの格納に関するレイアウトの詳細を理解する必要がなくなります。

この例を実行するには、次のコードをファイルにコピーして、`node` コマンドで指定します。次に例を示します。`node metadata.js` とします。`db.docments.patch` に渡すパラメータには、`categories` プロパティが含まれている必要があります。このプロパティによって、コンテンツではなくメタデータにパッチを適用することを示します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

// (1) Ensure the example document doesn't already exist
db.documents.remove('/patch/metadata-example.json');

// (2) Insert the base document into the database
db.documents.write({
  uri: '/patch/metadata-example.json',
  categories: ['content', 'metadata'],
```

```
    contentType: 'application/json',
    content: { key: 'value' },
    collections: [ 'initial1', 'initial2' ],
    permissions: [ {
      'role-name': 'app-user',
      capabilities: ['read']
    } ],
    properties: {
      myProp1: 'some-value',
      myProp2: 'some-other-value'
    }
  }).result().then(function(response) {
    // (3) Patch the document
    var pb = marklogic.patchBuilder;
    return db.documents.patch({
      uri: response.documents[0].uri,
      categories: [ 'metadata' ],
      operations: [
        // Add a collection
        pb.collections.add('INSERTED'),

        // Remove a collection
        pb.collections.remove('initial1'),

        // Add a capability to a role
        pb.permissions.replace(
            {'role-name': 'app-user',
             capabilities: ['read','update']}),

        // Add a property
        pb.properties.add('myProp3', 'INSERTED'),

        // Change the quality
        pb.quality.set(2)
      ]
    }).result();
  }).then(function(response) {
    // (4) Emit the resulting metadata
    return db.documents.read({
      uris: [ response.uri ],
      categories: [ 'metadata']
    }).result();
  }).then(function(documents) {
    console.log('collections: ' +
              JSON.stringify(documents[0].collections, null,
  2));
```

```
  console.log('permissions: ' +
              JSON.stringify(documents[0].permissions, null,
2));
  console.log('properties: ' +
               JSON.stringify(documents[0].properties, null,
2));
  console.log('quality: ' +
                JSON.stringify(documents[0].quality, null,
2));
}, function(error) { console.log(error); throw error; });
```

このスクリプト例の出力は、次のようになります。

```
collections: [
  "initial2",
  "INSERTED"
]
permissions: [
  {
    "role-name": "app-user",
    "capabilities": [
      "read",
      "update"
    ]
  },
  {
    "role-name": "rest-writer",
    "capabilities": [
      "update"
    ]
  },
  {
    "role-name": "rest-reader",
    "capabilities": [
      "read"
    ]
  }
]
properties: {
  "myProp1": "some-value",
  "myProp2": "some-other-value",
  "myProp3": "INSERTED"
}
quality: 2
```

app-user ロールに「更新」機能を追加するパッチ操作は、「update」を新しい値として capabilities 配列に挿入するのではなく、パーミッション全体を置き換えることに注意してください。各パーミッションを 1 つの単位として操作する必要があります。role-name や capabilities など、選択したコンポーネントを変更することはできません。

```
// Add a capability to a role
pb.permissions.replace(
          {'role-name': 'app-user',
           capabilities: ['read','update']})
```

また、rest-writer および rest-reader ロールを明示的に指定していない場合でも、これらのロールはドキュメントに割り当てられる点に注意してください。Node.js、Java、または REST クライアント API を使用して作成したすべてのドキュメントには、これらのロールがデフォルトで割り当てられます。詳細については、「セキュリティ要件」（15 ページ）を参照してください。

3.7　ビルダーを使用せずにパッチを作成する

この章の例では、marklogic.PatchBuilder を使用して、JSON によるパッチを作成しています。ただし、ビルダーを使用せずに RAW モードのパッチを作成して、それを db.documents.patch に JavaScript オブジェクトまたは文字列として渡すこともできます。

XML ドキュメントのコンテンツにパッチを適用する場合は、RAW モードのパッチを使用する必要があります。詳細については、「XML ドキュメントへのパッチの適用」（115 ページ）を参照してください。

RAW モードの XML および JSON パッチのシンタックスの詳細については、『REST Application Developer's Guide』の「Partially Updating Document Content or Metadata」を参照してください。

db.documents.patch の次の呼び出しは、配列要素を挿入する RAW モードの JSON パッチを適用します。

```
db.documents.patch(response.documents[0].uri,
    { patch: [ {
        insert: [ {
          context: '/theTop/child[2]',
          position: 'after',
          content: 'three'
    } ] } ] }
);
```

RAW モードのパッチでは、更新の各タイプ（挿入、置換、置換 - 挿入、削除）はオブジェクトの配列であり、各オブジェクトがそのタイプの 1 つの操作を表しています。パッチビルダー操作の呼び出しの出力は、そのように対応する操作を表します。

例えば、上記の RAW モードのパッチの `insert` 操作は、次のパッチビルダーの呼び出しと同等です。

```
pb.insert('/theTop/child[2]', 'after', 'three')
```

パッチビルダーの関数は、次のように RAW モードのパッチ操作に対応します。

PatchBuilder 関数	同等の RAW モードのパッチ

`insert(`
`  context, position,`
`  content, cardinality)`
	`"insert": [ ...,`
	`  { "context": ...,`
	`  "position": ...,`
	`  "content": ...,`
	`  "cardinality": ...`
	`  }, ...`
	`]`

`replace(select, content,`
`cardinality)`
	`"replace": [ ...,`
	`  { "select": ...,`
	`  "content": ...`
	`  "cardinality": ...`
	`  }, ...`
	`]`

`replaceInsert(`
`  select, context,`
`  position, content,`
`  cardinality)`
	`"replace-insert": [ ...,`
	`  { "select": ...,`
	`  "context": ...,`
	`  "position": ...,`
	`  "content": ...`
	`  "cardinality": ...`
	`  }, ...`
	`]`

`remove(select, cardinality)`	`"delete": [ ...,`
	`  { "select": ...,`
	`  "cardinality": ...`
	`  }, ...`
	`]`

3.8　XML ドキュメントへのパッチの適用

XML ドキュメントのコンテンツにパッチを適用する場合は、RAW モードの XML パッチを使用する必要があります。パッチビルダーが作成するのは JSON パッチ操作だけであり、JSON パッチは JSON ドキュメントにのみ適用できます。

RAW モードの XML パッチは、文字列として `db.documents.patch` に渡すことができます。RAW モードの XML パッチのシンタックスの詳細については、『REST Application Developer's Guide』の「XML Patch Reference」を参照してください。

次の例は、新しい要素を別の要素の子として挿入する RAW モードの XML パッチを適用します。パッチは、`db.documents.patch` の 2 番目のパラメータで文字列として渡されます。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

// (1) Insert the base document into the database
db.documents.write({
  uri: '/patch/raw-patch2.xml',
  contentType: 'application/xml',
  content: '<parent><child>data</child></parent>'
}).result().then(function(response) {
  // (2) Patch the document
  return db.documents.patch(
    response.documents[0].uri,
    '<rapi:patch
xmlns:rapi="http://marklogic.com/rest-api">' +
    '  <rapi:insert context="/parent"
position="last-child">' +
    '    <new-child>INSERTED</new-child>' +
    '  </rapi:insert>' +
    '</rapi:patch>'
  ).result();
}).then(function(response) {
  // (3) Emit the resulting document
  return db.documents.read(response.uri).result();
}).then(function(documents) {
  console.log(documents[0].content);
}, function(error) { console.log(error); throw error; });
```

次の表は、パッチによって適用されたドキュメント変換を示したものです。

更新前	更新後
<parent> <child>data</data> </parent>	<parent> <child>data</data> <new-child>INSERTED</new-child> </parent>

Node.js のその他の例については、「例：カスタム置換コンストラクタ」（124 ページ）を参照してください。詳細については、『REST Application Developer's Guide』の「"Partially Updating Document Content or Metadata" on page 62」を参照してください。

3.9　MarkLogic サーバーでの置換データの作成

ビルトインまたはカスタム置換コンストラクタ関数を使用すると、パッチ操作のコンテンツを MarkLogic サーバー上で動的に生成できます。ビルトイン関数は、単純な算術操作と文字列操作をサポートしています。例えば、ビルトイン関数を使用して、数値データの現在値を 1 つずつ増やしたり、文字列を連結したりできます。複雑な操作を実行する場合は、カスタム関数を作成してインストールします。

ここでは、次の内容を取り上げます。

- 置換コンストラクタ関数の概要
- ビルトイン置換コンストラクタの使用
- 置換コンストラクタにパラメータを渡す
- カスタム置換コンストラクタの使用
- カスタム置換コンストラクタの記述
- カスタム置換ライブラリのインストールまたは更新
- カスタム置換ライブラリのアンインストール
- 例：カスタム置換コンストラクタ
- 追加の操作

3.9.1 置換コンストラクタ関数の概要

置換コンストラクタ関数は、パッチの置換または置換 - 挿入操作のコンテンツを生成するサーバーサイド関数です。

置換コンストラクタ関数は、`PatchBuilder.replace` および `PatchBuilder.replaceInsert` 関数または RAW モードのパッチの `replace` および `replace-insert` 操作を使用してパッチ操作を作成するときに使用できます。置換コンストラクタ関数呼び出しの仕様は、アプリケーションによって提供される置換コンテンツの代わりとなります。置換コンストラクタ関数呼び出しは MarkLogic サーバー上で評価され、通常は、現在の値と相対的な新しいコンテンツを生成します。

例えば、ビルトインの `PatchBuilder.multiplyBy` 操作を使用して、プロパティの現在の値を 10% 増加できます。次の置換操作は、XPath 式 `/inventory/price` で選択されたすべての値を、現在の値の 1.1 倍にします。`multiplyBy` の結果は、新しいコンテンツではなく `PatchBuilder.replace` に渡される点に注意してください。

```
pb.replace('/inventory/price', pb.multiplyBy(1.1))
```

ビルトインの置換コンストラクタは、`PatchBuilder` のメソッドとして使用できます。詳細については、「ビルトイン置換コンストラクタの使用」（118 ページ）を参照してください。

`PatchBuilder.library` および `PatchBuilder.apply` を呼び出すことで、カスタム置換コンストラクタも使用できます。この場合、`PatchBuilder.library` を使用して、置換コンストラクタ関数が含まれているサーバーサイド XQuery ライブラリモジュールを指定し、`PatchBuilder.apply` を使用してそのモジュール内の関数を呼び出すパッチ操作を作成します。

例えば、次のコードスニペットは、XPath 式 `/inventory/price` で選択されたすべての値の価格を 2 倍にするパッチ置換操作を作成します。カスタム `dbl` 関数は、modules データベースにインストールされている、URI が `/ext/marklogic/patch/apply/my-lib.xqy` の XQuery ライブラリモジュールによって実装されます。`dbl` 関数は引数値をとらないため、`PatchBuilder.apply` には追加のコンテンツが提供されません。

```
pb.library('my-lib'),
pb.apply('dbl')
```

詳細については、「カスタム置換コンストラクタの記述」（121 ページ）を参照してください。

RAW モードのパッチで置換ジェネレータ関数を使用する方法の詳細については、『REST Application Developer's Guide』の「Constructing Replacement Data on the Server」を参照してください。

3.9.2　ビルトイン置換コンストラクタの使用

Node.js クライアント API には、`replace` または `replaceInsert` パッチ操作に対応したコンテンツを動的に生成する目的で使用できる複数のビルトインサーバーサイド関数が用意されています。例えば、ビルトイン関数を使用して、データ項目の現在の値を 1 つずつ増やせます。

ビルトイン算術関数は、XQuery の `+`、`-`、`*`、および `div` 演算子と同等であり、同じデータ型にキャスト可能な値を受け付けます。つまり、数値、日付、日時、期間、グレゴリオ暦（`xs:gMonth`、`xs:gYearMonth` など）の値を受け付けます。オペランドタイプの組み合わせは、XQuery と同様にサポートされています。詳細については、http://www.w3.org/TR/xquery/#mapping を参照してください。その他のすべての関数は、文字列にキャスト可能な値を想定します。

`PatchBuilder` のインターフェイスには、各ビルトイン関数に対応するメソッドが含まれています。パッチビルダーではなく RAW モードのパッチを使用する場合は、『REST Application Developer's Guide』の「Constructing Replacement Data on the Server」を参照してください。

以下の表は、使用可能なビルトインの置換コンストラクタ関数を示したものです。この表では、*`$current`* は置換操作のターゲットの現在の値を表し、*`$arg`* と *`$argN`* はパッチによって渡された引数値を表します。詳細については、Node.js API リファレンスを参照してください。「apply 操作名」列は、`patchBuilder.apply` で使用する同等の操作の名前を示します。

PatchBuilder 関数	apply 操作名	引数の数	効果
`add`	`ml.add`	1	`$current + $arg`
`subtract`	`ml.subtract`	1	`$current - $arg`
`multiplyBy`	`ml.multiply`	1	`$current * $arg`
`divideBy`	`ml.divide`	1	`$current div $arg`
`concatBefore`	`ml.concat-before`	1	`fn:concat($arg, $current)`

PatchBuilder 関数	apply 操作名	引数の数	効果
concatAfter	ml.concat-after	1	fn:concat($current, $arg)
concatBetween	ml.concat-between	2	fn:concat($arg1, $current, $arg2)
substringBefore	ml.substring-before	1	fn:substring-before ($current, $arg)
substringAfter	ml.substring-after	1	fn:substring-after($current, $arg)
replaceRegex	ml.replace-regex	2 または 3	fn:replace($current, $arg1, $arg2, $arg3)

3.9.3　置換コンストラクタにパラメータを渡す

パッチビルダーを使用してビルトインまたはカスタムの置換コンストラクタの呼び出しを作成する場合は、必要な引数を patchBuilder メソッドに渡します。

例えば、patchBuilder.concatBetween は、入力として指定した 2 つの文字列について、選択した各値を連結します。そのため、concatBetween メソッドは、引数として 2 つの入力文字列を取ります。次の例は、選択したデータ項目の現在の値の前後に文字列「fore」と「aft」を連結します。

```
pb.replace('/some/path/expr',
           pb.concatBetween('fore', 'aft'))
```

RAW モードのパッチでは、操作の content プロパティに入力引数値を指定します。詳細については、『REST Application Developer's Guide』の「Using a Replacement Constructor Function」を参照してください。

入力引数値のデータ型を明示的に指定するには、patchBuilder.apply および patchBuilder.datatype を使用します。選択できるデータ型は、XML スキーマでサポートされているものです。詳細については、http://www.w3.org/TR/xmlschema-2/#built-in-datatypes を参照してください。データ型名の名前空間プレフィックスは省略してください。

例えば、次の呼び出しでは、patchBuilder.multiplyBy を呼び出すのと同等の RAW モードのパッチ操作の入力値のデータ型として明示的に xs:long を指定しています。apply で使用可能なビルトイン関数名のリストについては、「ビルトイン置換コンストラクタの使用」（118 ページ）の表を参照してください。

```
pb.replace('/some/path/expr',
  p.apply('ml.add', p.datatype('long'),
'92233720368547758O7'))
```

3.9.4　カスタム置換コンストラクタの使用

カスタム置換コンストラクタを使用するには、コンストラクタの実装を含む XQuery ライブラリモジュールを REST API インスタンスの modules データベースにインストールしておく必要があります。詳細については、「カスタム置換ライブラリのインストールまたは更新」（122 ページ）を参照してください。

カスタム置換コンストラクタ関数のライブラリをパッチ操作で使用できるようにするには、PatchBuilder.library を呼び出して得られた結果をパッチに含めます。library 操作では、MarkLogic サーバー上で実装を特定する方法を API に通知します。

例えば、次の library 呼び出しは、パッチで使用されるあらゆるカスタム置換コンストラクタ関数が、modules データベース URI /ext/marklogic/patch/apply/my-replace-lib.xqy の XQuery モジュールにあることを示します。

```
pb.patch('/some/doc.json',
  pb.library('my-replace-lib.xqy'),
  ...)
```

このようなライブラリは、パッチごとに 1 つだけ使用できます。ただし、同じライブラリの関数を複数使用することはできます。

ライブラリ内の関数の呼び出しを作成するには、PatchBuilder.apply を使用します。例えば、my-replae-lib.xqy に dbl という関数が含まれている場合は、次の呼び出しの結果をパッチに含めることで使用できます。

```
pb.patch('/some/doc.json',
  pb.library('my-replace-lib.xqy'),
  pb.apply('dbl')
  /* additional patch operations */)
```

関数が入力引数を取る場合は、それらを apply 呼び出しに含めます。

```
pb.apply('doSomething', 'arg1Val', 'arg2Val')
```

指定する関数には、ユーザーの責任において想定される型の値を渡してください。型のチェックは行われません。

3.9.5 カスタム置換コンストラクタの記述

このセクションでは、カスタム置換コンストラクタを実装する要件について説明します。実装の例については、「例：カスタム置換コンストラクタ」（124 ページ）を参照してください。

`replace` および `replace-insert` 操作に対応するコンテンツを生成する独自の関数は、XQuery を使用して作成できます。カスタム置換ジェネレータ関数には、次の XQuery インターフェイスが用意されています。

```
declare function module-ns:func-name(
  $current as node()?,
  $args as item()*
) as node()*
```

置換（または置換 - 挿入）操作のターゲットノードは、`$current` で提供されます。関数を置換 - 挿入操作の代わりに挿入操作として呼び出した場合、`$current` は空です。

操作によって提供される引数のリストは、`$args` を通じて渡されます。引数値の検証は、ユーザーが実行してください。パッチ操作によって提供された引数が JSON 配列または XML `<rapi:value/>` 要素のシーケンスの場合、$args は各値に `fn:data` 関数を適用した結果になります。パッチ操作によって明示的なデータ型が指定されている場合、関数を呼び出す前にキャストが適用されます。

関数は、`fn:error` と `RESTAPI-SRVEXERR` を使用してエラーを報告しなければなりません。詳細については、「拡張機能および変換機能でのエラーの報告」（262 ページ）を参照してください。

パッチビルダーを使用してモジュールへの参照を作成するには、次の名前空間およびインストール URI の規則に従う必要があります。

- モジュールは名前空間 `http://marklogic.com/patch/apply/`*`yourModuleName`* に配置されていなければなりません。
- モジュールは、URI が `/ext/marklogic/patch/apply/`*`yourModuleName`* の形式の modules データベースにインストールする必要があります。`db.config.patch.replace.write` を使用してモジュールをインストールした場合は、自動的にこのようになります。

例えば、ライブラリモジュールに次のモジュール名前空間宣言が含まれていて、

```
module namespace my-lib =
"http://marklogic.com/patch/apply/my-lib";
```

次の URI の modules データベースにインストールした場合、

```
/ext/marklogic/patch/apply/my-lib.xqy
```

次のパッチによって `my-lib.xqy` 内の関数 `dbl` が正常に適用されます。

```
pb.patch('/some/doc.json',
  pb.library('my-lib.xqy'),
  pb.replace('/some/path/expr', pb.apply('dbl')),
  ...)
```

この簡略的な規則は、RAW モードのパッチには該当しません。RAW モードのパッチの場合は、命名規則に従っている場合でも、完全な名前空間とモジュールパスを `replace-library` ディレクティブで明示的に指定する必要があります。

3.9.6 カスタム置換ライブラリのインストールまたは更新

カスタム置換コンストラクタ関数をインストールまたは更新するには、目的の関数を XQuery ライブラリモジュールに配置して、モジュールと依存ライブラリを REST API インスタンスに関連付けられている modules データベースにインストールします。

モジュールのインストールには、`config.patch.replace.write` を使用します。この関数により、`PatchBuilder` が想定する modules データベース URI 規則を使用してモジュールがインストールされます。モジュールが必須の名前空間規則を使用していることを必ず確認してください。詳細については、「カスタム置換コンストラクタの記述」（121 ページ）を参照してください。

例えば次のスクリプトは、URI が `/ext/marklogic/patch/apply/my-lib.xqy` の modules データベースにモジュールをインストールします。この実装は、パス名が `./my-lib.xqy` のファイルから読み取られます。このモジュールは、`app-user` ロールが割り当てられているユーザーが実行できます。

```
var fs = require('fs');
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.config.patch.replace.write('my-lib.xqy',
  [{'role-name': 'app-user', capabilities: ['execute']}],
  fs.createReadStream('./my-lib.xqy')
).result(function(response) {
  console.log('Installed module ' + response.path);
```

```
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

ここでは、URI のモジュール名部分（'my-lib.xqy'）だけが渡されていることに注目してください。必要な URI の残りの部分は自動的に作成されます。

modules データベースへの実装を記述においてパーミッションを指定していない場合、このモジュールは rest-admin ロールが割り当てられているユーザーのみが実行できます。

完全な例については、「例：カスタム置換コンストラクタ」（124 ページ）を参照してください。

ライブラリモジュールが依存ライブラリを必要とする場合は、extlibs インターフェイスを使用してインストールできます。extlibs インターフェイスにより、ディレクトリレベルとファイルレベルの両方で modules データベースアセットを管理できます。詳細については、「modules データベースでのアセットの管理」（275 ページ）を参照してください。

db.config.patch.replace.write の呼び出しは、db.config.extlibs.write を呼び出して、パスパラメータを PatchBuilder 規則に従った値に設定することと同じです。例えば次の呼び出しは、上記の db.config.patch.replace.write を使用した場合と同じようにインストールを実行します。

```
db.config.extlibs.write({
  path: '/marklogic/patch/apply/my-lib.xqy',
  permissions: [
    {'role-name': 'app-user', capabilities: ['execute']}
  ],
  contentType: 'application/xquery',
  source: fs.createReadStream('./my-lib.xqy')
})
```

3.9.7 カスタム置換ライブラリのアンインストール

カスタム置換コンストラクタ関数を含むモジュールを削除するには、db.config.patch.replace.remove を使用します。例えば、次の呼び出しは、db.config.patch.replace.write を使用してインストールされているモジュール my-lib.xqy を削除します。

```
db.config.patch.replace.remove('my-lib.xqy');
```

3.9.8 例：カスタム置換コンストラクタ

この例では、カスタム置換コンストラクタ関数を含む XQuery ライブラリモジュールをインストールおよび使用する方法について説明します。

この例では、dbl および min という名前の 2 つのカスタム置換コンストラクタを含む XQuery ライブラリモジュールをインストールします。dbl 関数は、値が元の入力の 2 倍である新しいノードを作成します。これは追加の引数を受け付けません。min 関数は、値が現在の値の最小値である新しいノードを作成し、値は追加の引数として渡されます。

わかりやすくするため、この例では、実稼動の実装で必要とされる入力データ検証機能のほとんどを省略しています。例えば、min 関数は JSON 数値ノードと XML 要素を入力として受け付けますが、ブール型、テキスト、または日付の入力は受け付けません。また、min は追加の入力引数ですが、検証を一切実行していません。

置換コンテンツジェネレータをインストールすると、パッチが適用されてオレンジの値が 2 倍になり（10 から 20）、梨の最低価格が選択されます。

次の手順に従って、例で使用するファイルをセットアップします。

1. 次の XQuery モジュールを my-lib.xqy という名前のファイルにコピーします。このファイルには、カスタム置換コンストラクタの実装が含まれています。

```
xquery version "1.0-ml";

module namespace my-lib =
"http://marklogic.com/patch/apply/my-lib";

(: Double the value of a node :)
declare function my-lib:dbl(
  $current as node()?,
  $args as item()*
) as node()*
{
  if ($current/data() castable as xs:decimal)
  then
    let $new-value := xs:decimal($current) * 2
    return
      typeswitch($current)
      case number-node()        (: JSON :)
        return number-node {$new-value}
      case element()            (: XML :)
        return element {fn:node-name($current)}
{$new-value}
      default return fn:error((), "RESTAPI-SRVEXERR",
```

```
          ("400", "Bad Request",
             fn:concat("Not an element or number node: ",
                       xdmp:path($current))
             ))
  else fn:error((), "RESTAPI-SRVEXERR",
      ("400", "Bad Request", fn:concat("Non-decimal data:
", $current)
    ))
};

(: Find the minimum value in a sequence of value composed
of :)
(: the current node and a set of input values.:)
declare function my-lib:min(
  $current as node()?,
  $args as item()*
) as node()*
{
  if ($current/data() castable as xs:decimal)
  then
    let $new-value := fn:min(($current, $args))
    return
      typeswitch($current)
      case element()            (: XML :)
        return element {fn:node-name($current)}
{$new-value}
      case number-node()        (: JSON :)
        return number-node {$new-value}
      default return fn:error((), "RESTAPI-SRVEXERR",
        ("400", "Bad Request",
           fn:concat("Not an element or number node: ",
                     xdmp:path($current))
        ))
  else fn:error((), "RESTAPI-SRVEXERR", ("400", "Bad
Request",
    fn:concat("Non-decimal data: ", $current)))
};
```

2. 次のスクリプトを `install-udf.js` という名前のファイルにコピーします。このスクリプトにより、上記の XQuery モジュールがインストールされます。この説明では、このモジュールは、含まれている関数を実行するパーミッションをロール `app-user` が持っているものとして、インストールされています。

```
var fs = require('fs');
var marklogic = require('marklogic');
var my = require('./my-connection.js');
```

```
var db = marklogic.createDatabaseClient(my.connInfo);

db.config.patch.replace.write('my-lib.xqy',
  [ {'role-name': 'app-user', capabilities: ['execute']} ],
  fs.createReadStream('./my-lib.xqy')
).result(function(response) {
  console.log('Installed module ' + response.path);
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

3. 次のスクリプトを udf.js という名前のファイルにコピーします。このスクリプトはベースドキュメントをデータベースに挿入して、dbl および min 置換コンテンツコンストラクタを使用するパッチを適用します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

// (1) Insert the base document into the database
db.documents.write({
  uri: '/patch/udfs.json',
  contentType: 'application/json',
  content: {
    inventory: [
      {name: 'orange', price: 10},
      {name: 'apple', price: 15},
      {name: 'pear', price: 20}
  ] }
}).result().then(function(response) {
  // (2) Patch the document
  var pb = marklogic.patchBuilder;
  return db.documents.patch(response.documents[0].uri,
    pb.library('my-lib.xqy'),
    pb.replace('/inventory[name eq "orange"]/price',
pb.apply('dbl')),
    pb.replace('/inventory[name eq "pear"]/price',
               pb.apply('min', 18, 21))
  ).result();
}).then(function(response) {
  // (3) Emit the resulting document
  return db.documents.read(response.uri).result();
}).then(function(documents) {
  console.log(JSON.stringify(documents[0].content, null,
2));
}, function(error) { console.log(error); throw error; });
```

次の手順に従ってこの例を実行します。この手順では、置換コンテンツコンストラクタモジュールをインストールし、ベースドキュメントをデータベースに挿入して、ドキュメントにパッチを適用し、更新ドキュメントのコンテンツを表示します。

1. 置換コンテンツコンストラクタモジュールをインストールします。

```
node install-udf.js
```

2. ドキュメントを挿入し、dbl および min 関数を使用してパッチを適用します。

```
node udf.js
```

次のような出力が表示されます。

```
{
  "inventory": [
    {
      "name": "orange",
      "price": 20
    },
    {
      "name": "apple",
      "price": 15
    },
    {
      "name": "pear",
      "price": 18
    }
  ]
}
```

次のパッチ操作の dbl 関数によって、オレンジの価格が 2 倍（10 から 20）になります。

```
pb.replace('/inventory[name eq "orange"]/price',
pb.apply('dbl'))
```

次のパッチ操作の min 関数によって、梨の価格が 20 から 18 に引き下げられます。

```
pb.replace('/inventory[name eq "pear"]/price',
           pb.apply('min', 18, 21))
```

各パッチ操作で使用されている XPath 式がターゲットドキュメント内の数値ノード（price）を選択し、置換コンテンツコンストラクタ関数が新しい数値ノードを作成します。

```
typeswitch($current)
  case element()            (: XML :)
    return element {fn:node-name($current)} {$new-value}
  case number-node()        (: JSON :)
    return number-node {$new-value}
```

新しい値のみを返すことはできません。置換ノード全体を返す必要があります。JSON ドキュメントモデルおよび JSON ノードコンストラクタの詳細については、『Application Developer's Guide』の「Working With JSON」を参照してください。

ノードタイプ $current に対して typeswitch を使用して、置換コンストラクタの例を XML 入力で機能させることもできます。以前にインストールした dbl および min 置換コンテンツコンストラクタを使用して、同等の処理を XML ドキュメントに適用するには、次のスクリプトを実行します。「XML ドキュメントへのパッチの適用」（115 ページ）で説明しているように、XML ドキュメントを操作する場合は、ビルダーではなく RAW モードのパッチを使用する必要があります。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

// (1) Insert the base document into the database
db.documents.write({
  uri: '/patch/udf.xml',
  contentType: 'application/xml',
  content:
    '<inventory>' +
      '<item>' +
        '<name>orange</name>' +
        '<price>10</price>' +
      '</item>' +
      '<item>' +
        '<name>apple</name>' +
        '<price>15</price>' +
      '</item>' +
      '<item>' +
        '<name>pear</name>' +
        '<price>20</price>' +
      '</item>' +
    '</inventory>'
```

```
}).result().then(function(response) {
  // (2) Patch the document
  return db.documents.patch(
    response.documents[0].uri,
    '<rapi:patch
xmlns:rapi="http://marklogic.com/rest-api">' +
      '<rapi:replace-library ' +
          'at="/ext/marklogic/patch/apply/my-lib.xqy" ' +
         'ns="http://marklogic.com/patch/apply/my-lib" />'
+
      '<rapi:replace ' +
          'select="/inventory/item[name eq
\'orange\']/price" ' +
          'apply="dbl" />' +
      '<rapi:replace ' +
          'select="/inventory/item[name eq
\'pear\']/price" ' +
          'apply="min">' +
        '<rapi:value>18</rapi:value>' +
        '<rapi:value>21</rapi:value>' +
      '</rapi:replace>' +
    '</rapi:patch>'
  ).result();
}).then(function(response) {
  // (3) Emit the resulting document
  return db.documents.read(response.uri).result();
}).then(function(documents) {
  console.log(documents[0].content);
}, function(error) { console.log(error); throw error; });
```

RAW モードのパッチでは、モジュールのパスとモジュールの名前空間を `replace-library` ディレクティブに明示的に指定する必要があります。

```
<rapi:replace-library
at="/ext/marklogic/patch/apply/my-lib.xqy"
                       ns="http://marklogic.com/patch/apply
/my-lib" />'
```

「カスタム置換ライブラリのインストールまたは更新」（122 ページ）で説明しているインストールパス規則に従っている限り、`PatchBuilder` を使用して JSON パッチを作成すると、このような情報は `PatchBuilder.library` の呼び出しによって自動的に入力されます。

3.9.9 追加の操作

db.config.patch.replace インターフェイスは、置換コンテンツコンストラクタ関数を含むライブラリモジュールを管理するために、次のような追加メソッドも提供しています。

- `db.config.patch.replace.read` - 置換ライブラリの実装を取得します。これは、`db.config.patch.replace.write` を使用してインストールされたモジュールを返します。
- `db.config.patch.replace.list` - インストールされているすべての置換ライブラリモジュールのリストを取得します。

詳細については、『Node.js Client API Reference』を参照してください。

4.0 ドキュメントとメタデータのクエリ

この章では、Node.js クライアント API を使用したデータベースコンテンツとメタデータのクエリに関する次のトピックについて説明します。

4.1 クエリインターフェイスの概要

Node.js クライアント API には、さまざまなタイプのクエリを使用してドキュメントを検索したり、レキシコンをクエリしたりできるインターフェイスが用意されています。次のインターフェイスがクエリ操作をサポートしています。

メソッド	説明
`marklogic.queryBuilder`	`DatabaseClient.documents.query` で使用する文字列クエリ、QBE、または構造化クエリを作成します。詳細については、queryBuilder インターフェイスについてを参照してください。

メソッド	説明
`DatabaseClient.documents.query`	文字列クエリ、構造化クエリ、複合クエリ、または QBE（Query By Example）にマッチするドキュメントを検索し、検索結果の概要、マッチするドキュメント、またはその両方を返します。詳細については、「Query By Example を使用した検索」（153 ページ）、「文字列クエリを使用した検索」（140 ページ）、または「構造化クエリを使用した検索」（158 ページ）を参照してください。
`DatabaseClient.queryCollection`	コレクションで永続的な JavaScript オブジェクトを検索して、マッチするオブジェクトを返します。
`DatabaseClient.valuesBuilder`	`DatabaseClient.values.read` で使用する値クエリを作成します。詳細については、「レキシコンおよびレンジインデックスのクエリ」（174 ページ）を参照してください。
`DatabaseClient.values.read`	レキシコンまたはレンジインデックスで値またはタプル（共起）をクエリします。詳細については、「レキシコンおよびレンジインデックスのクエリ」（174 ページ）を参照してください。
`DatabaseClient.documents.suggest`	レキシコン内で文字列をマッチさせて、検索語の入力候補を提供します。詳細については、「検索語入力候補の生成」（198 ページ）を参照してください。
`DatabaseClient.config.query`	検索結果の変換機能、スニペッタ、および文字列クエリパーサーなど、modules データベースに格納されているクエリ関連のカスタマイズを管理します。

4.2 検索の概念の概要

このセクションでは、検索の概念に関する概要と、Node.js クライアント API で公開されている機能について説明します。検索の概念の詳細については、『Search Developer's Guide』を参照してください。

MarkLogic サーバーデータベースは 2 種類の方法でクエリできます。1 つはドキュメントコンテンツとメタデータを検索する方法で、もう 1 つはコンテンツから作成した値およびワードレキシコンをクエリする方法です。このトピックでは、コンテンツとメタデータを検索する方法について説明します。レキシコンについては、「レキシコンおよびレンジインデックスのクエリ」（174 ページ）を参照してください。

このセクションでは、次の内容を取り上げます。

- 検索の概要
- クエリスタイル
- クエリのタイプ
- インデックス付け

4.2.1 検索の概要

検索の実行は、次の基本的なフェーズで構成されています。

1. 目的の結果セットを定義する一連の基準を作成する。
2. 返す結果の数またはソート順序などの属性を定義して結果セットをより細かく指定する。
3. データベースまたはレキシコンを検索する。

Node.js クライアント API には、クエリの定義および詳細設定の詳細な構造を抽象化する `marklogic.queryBuilder` インターフェイスが用意されています。詳細については、「queryBuilder インターフェイスについて」（137 ページ）を参照してください。クエリ操作を実行するには、`DatabaseClient.documents.query` を使用します。

MarkLogic サーバーは、マッチするフレーズ、特定の値、値の範囲、位置情報の領域など、さまざまな検索基準をサポートしています。これらのクエリタイプおよびその他のクエリタイプについては、「クエリのタイプ」（135 ページ）で説明します。検索基準は、いずれかのクエリスタイルを使用して表現できます。詳細については、「クエリスタイル」（134 ページ）を参照してください。

クエリの結果の詳細設定としては、ドキュメント全体、コンテンツスニペット、ファセット情報、集計結果を返すかどうかなどがあります。また、独自のスニペット化アルゴリズムや検索結果のカスタム変換を定義することもできます。詳細については、「クエリ結果の詳細設定」（189 ページ）を参照してください。

`marklogic.valuesBuilder` および `DatabaseClient.values.read` を使用して、ドキュメントから作成したレキシコンを分析することもできます。詳細については、「レキシコンおよびレンジインデックスのクエリ」（174 ページ）を参照してください。

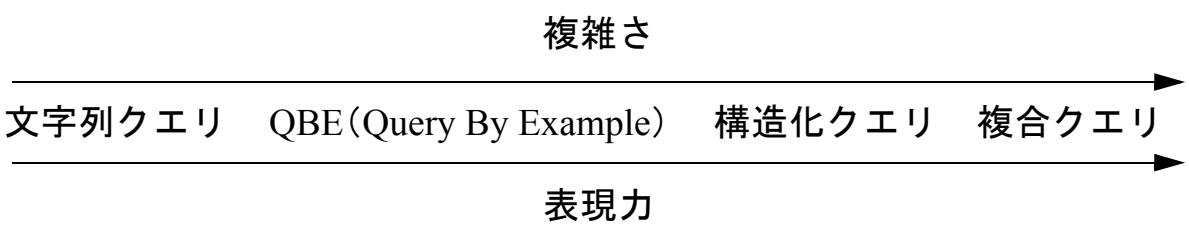
4.2.2 クエリスタイル

Node.js クライアント API を使用してドキュメントコンテンツやメタデータを検索する場合は、次のクエリスタイルを使用して検索基準を表現できます。シンタックスや表現力は、スタイルごとに異なります。

クエリスタイル	説明
QBE（Query By Example）	マッチの対象となるドキュメントの構造をモデル化してドキュメントを検索します。詳細については、「Query By Example を使用した検索」（153 ページ）および `queryBuilder.byExample` を参照してください。
文字列クエリ	ユーザーが検索ボックスに入力するような Google スタイルのクエリ文字列を使用して、ドキュメントとメタデータを検索します。例えば、「cat AND dog」という形式のクエリは、語句「cat」と「dog」が含まれているすべてのドキュメントとマッチします。詳細については、「文字列クエリを使用した検索」（140 ページ）および `queryBuilder.parsedFrom` を参照してください。
構造化クエリ	豊富なサブクエリタイプから複雑なクエリを作成してドキュメントとメタデータを検索します。詳細については、「構造化クエリを使用した検索」（158 ページ）を参照してください。
複合クエリ	ユーザーが他のクエリスタイルとクエリオプションを組み合わせることができるクエリオブジェクトを使用して、ドキュメントとメタデータを検索します。複合クエリは、手動でクエリを作成するユーザー向けの高度な機能です。詳細については、「複合クエリを使用した検索」（172 ページ）を参照してください。

すべてのクエリスタイルが豊富な検索機能をサポートしていますが、一般に文字列クエリよりも QBE、QBE よりも構造化クエリが表現力に富みます。複合クエリはスーパーセットであるため他のどれよりも豊富な表現が可能です。文字列クエリと QBE は簡単に使うことができ、幅広い検索ニーズに対応できるように設計されています。ただし、これらは構造化クエリおよび複合クエリと同じレベルの検索制御性は提供しません。

次の図は、このトレードオフの概要を示したものです。



文字列クエリと構造化クエリの基準は、単一のクエリ操作として組み合わせることができます。QBE は、他の 2 つのクエリスタイルと組み合わせることはできません。

詳細については、『Search Developer's Guide』の「Overview of Search Features in MarkLogic Server」を参照してください。

4.2.3 クエリのタイプ

クエリは、検索基準をカプセル化します。どのクエリスタイル（文字列、QBE、または構造化）を使用する場合でも、基準はこのセクションで説明する 1 つあるいは複数のクエリタイプに分類されます。

次のクエリタイプが、検索の基本となります。これでマッチさせるコンテンツを記述します。

- レンジ：関係式を満たす値とマッチさせます。「5 未満」や「true と等しくない」のような条件を表すことができます。レンジクエリでは、レンジインデックスが利用されている必要があります。
- 値：特定の JSON プロパティや XML 要素にある、文字列や数字のようなリテラル値全体とマッチさせます。デフォルトでは、値クエリは完全一致を使用します。例えば、「mark」の検索は「Mark Twain」にマッチしません。
- ワード：特定の JSON プロパティまたは XML 要素や属性にある単語（ワード）または語句とマッチさせます。値クエリと対照的に、ワードクエリはデフォルトでテキスト値の一部にマッチし、完全一致を使用しません。例えば、指定したコンテキストで「mark」の検索は「Mark Twain」にマッチします。
- ターム：任意の場所に出現する単語または語句とマッチさせます。値クエリと対照的に、タームクエリはデフォルトでテキスト値の一部にマッチし、完全一致を使用しません。例えば、「mark」の検索は「Mark Twain」にマッチします。

追加のクエリタイプを使用すると、基本的なコンテンツクエリを互いに組み合わせたり、制約を追加する基準と組み合わせたりして、複雑なクエリを構築できます。追加のクエリタイプは、次のカテゴリに分類されます。

- 論理構成子：基準間の論理的な関係性を表します。「*x* AND (*y* OR *z*)」のような複合論理式を構築できます。

- ドキュメントセレクタ：コレクション、ディレクトリ、または URI に基づいてドキュメントを選択します。例えば、「コレクション *y* 内のドキュメント内で *x* が出現した場合のみ」のような基準を表すことができます。
- 場所修飾子：マッチが出現した場所に基づいて結果をさらに制限します。例えば、「JSON プロパティ *z* に含まれている *x* のみ」、「*y* から *n* 語以内に *x* が出現する *x* のみ」、「ドキュメントプロパティ内に出現する *x* のみ」などです。

文字列クエリは、追加設定なしでタームクエリと論理構成子をサポートしています。例えば、クエリ文字列「cat AND dog」は、暗黙的に「and」論理構成子で結合された 2 タームのクエリになります。

ただし、パースバインドを使用することで簡単に文字列クエリの表現力を拡張し、追加のクエリタイプを有効にできます。例えば、レンジクエリバインドを使用して識別子「cost」を特定のインデックス付き JSON プロパティに連結すると、形式が「cost GT 10」の文字列クエリを使用できるようになります。詳細については、「文字列クエリを使用した検索」（140 ページ）を参照してください。

QBE では、コンテンツマッチのデフォルトは値クエリです。例えば、`{'my-key': 'desired-value'}` という形式の QBE 検索基準は、暗黙的に値が `'desired-value'` の JSON プロパティ `'my-key'` の値クエリになります。ただし、QBE シンタックスには他のタイプのクエリを作成できる特別なプロパティ名が用意されています。例えば、値クエリの代わりにワードクエリを作成するには、`$word` を使用して `{'my-key': {'$word': 'desired-value'}}` のように記述します。詳細については、「Query By Example を使用した検索」（153 ページ）を参照してください。

構造化クエリの場合、`queryBuilder` インターフェイスにすべてのクエリタイプに対応するビルダーが用意されています。これらのビルダーを互いに組み合わせて使用できます。`queryBuilder.Query` を返すすべての `queryBuilder` メソッドは、上記のいずれかのクエリカテゴリに分類されるクエリまたはサブクエリを作成します。詳細については、「構造化クエリを使用した検索」（158 ページ）を参照してください。

4.2.4 インデックス付け

レンジクエリでは、インデックスが利用されている必要があります。インデックス利用が任意のクエリでも、フィルタリングなし検索を有効にすることでインデックス付けのメリットが得られます。詳細については、『Query Performance and Tuning Guide』の「Fast Pagination and Unfiltered Searches」を参照してください。

レンジインデックスは、管理画面、XQuery Admin API、REST 管理 API を使用して作成できます。また、Configuration Manager または REST パッケージ化 API を使用して、データベースまたはホスト間でインデックスの設定をコピーすることもできます。詳細については、次を参照してください。

- 『Administrator's Guide』の「Range Indexes and Lexicons」
- 『Administrator's Guide』の「Using the Configuration Manager」
- 『MarkLogic REST API Reference』の
  `PUT:/manage/v2/databases/{id|name}/properties`

JSON プロパティに対してインデックスを作成するには、要素レンジインデックスインターフェイスを使用します。インデックス設定に関しては、JSON プロパティと名前空間なしの XML 要素は同等です。

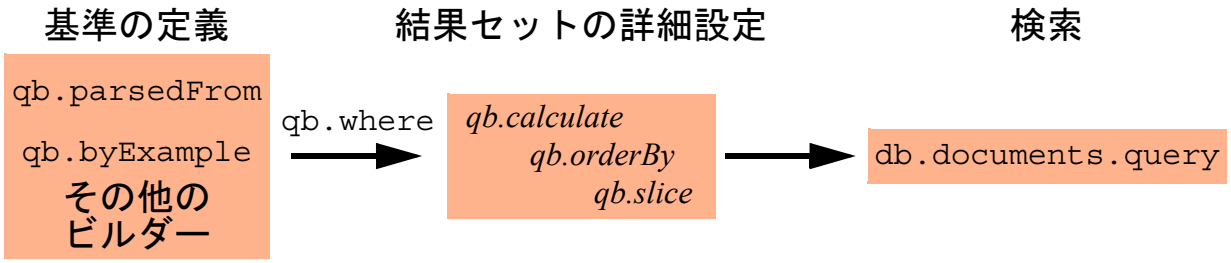
Node.js クライアント API のバインド機能を使用して、文字列クエリで使用可能なインデックスリファレンスを名前にバインドできます。詳細については、「文字列クエリでの制約の使用」（143 ページ）および「検索語入力候補の生成」（198 ページ）を参照してください。レキシコンおよびインデックスに対する値クエリでもインデックスリファレンスを使用します。詳細については、「インデックスリファレンスの作成」（179 ページ）を参照してください。

4.3 queryBuilder インターフェイスについて

`marklogic.queryBuilder` インターフェイスを使用した検索の実行は、次のフェーズで構成されています。

1. 一連の検索基準を作成し、目的の結果セットを定義するクエリを作成する。
2. 返す結果の数またはソート順序などの属性を定義して結果セットをより細かく指定する。
3. データベースを検索する。

次の図は、Node.js クライアント API による、`queryBuilder` と `DatabaseClient.documents.query` を使用した検索の定義および実行方法を示したものです。図の「qb」は `queryBuilder` オブジェクトを表し、「db」は `DatabaseClient` オブジェクトを表しています。斜体の関数はオプションです。



上記の各手順の詳細を次に示します。

1. 文字列クエリ（qb.parsedFrom）、QBE（qb.byExample）、または構造化クエリ（qb.word、qb.range、qb.or などのその他のビルダー）を使用して検索基準を定義します。次に例を示します。

   ```
   qb.parsedFrom("dog")
   ```

   文字列と 1 つあるいは複数の構造化ビルダーを一緒に渡すことができます。これらは AND で結合します。QBE は他のクエリタイプと組み合わせることはできません。

2. 基準を queryBuilder.where に渡してクエリにカプセル化します。これにより、queryBuilder.BuiltQuery オブジェクトが生成されます。このオブジェクトは、結果セットの詳細な設定の有無を問わず、DatabaseClient.documents.query に渡すのに適しています。

   ```
   qb.where(qb.parsedFrom("dog"))
   ```

3. 必要に応じて、結果セットの詳細な設定をクエリに適用します。また、必要とする結果に応じて、次のすべてまたは任意の手順をスキップできます。

   a. queryBuilder.slice を使用して、結果セットからドキュメントのサブセットを選択したり、選択した結果に適用するサーバーサイド変換を指定したりします。デフォルトでは、スライスは最初の 10 個のドキュメントで、変換は行われません。

   b. queryBuilder.orderBy を使用して、ソートキーやソート方向を指定します。

   c. queryBuilder.calculate を使用して、結果セットに対して 1 つあるいは複数の集計を行います。

4. 必要に応じて queryBuilder.withOptions を使用し、より詳細な検索オプションやトランザクション ID の指定、またはクエリデバッグ情報の要求など、詳細な設定を検索に追加します。

5. 最終的な BuiltQuery オブジェクトを DatabaseClient.documents.query 関数に渡して検索を実行します。次に例を示します。

   ```
   db.documents.query(qb.where(qb.parsedFrom("dog")))
   ```

次の表は、使用可能な各クエリスタイルにおける、queryBuilder を使用した同等クエリの作成例です。このクエリは、語句「cat」と「dog」の両方が含まれているドキュメントとマッチします。検索のクエリ作成部分だけが、クエリスタイルごとに異なる点に注目してください。

クエリスタイル	コードスニペット

文字列	`db.documents.query(`
	`  qb.where(`
	`  qb.parsedFrom('cat AND dog')`
	`  ).orderBy(qb.sort('descending')`
	`  .slice(0,5)`
	`)`

QBE	`db.documents.query(`
	`  qb.where(`
	`  qb.byExample({`
	`  $and:[{$word:'cat'},{$word:'dog'}]`
	`  })`
	`  ).orderBy(qb.sort('descending')`
	`  .slice(0,5)`
	`)`

構造化	`db.documents.query(`
	`  qb.where(`
	`  qb.and(qb.term('cat'), qb.term('dog'))`
	`  ).orderBy(qb.sort('descending')`
	`  .slice(0,5)`
	`)`

文字列と構造化の組み合わせ	`db.documents.query(`
	`  qb.where(`
	`  qb.term('cat'),`
	`  qb.parsedFrom('dog')`
	`  ).orderBy(qb.sort('descending')`
	`  .slice(0,5)`
	`)`

詳細については、次のいずれかのトピックを参照してください。

- 「文字列クエリを使用した検索」（140 ページ）
- 「Query By Example を使用した検索」（153 ページ）
- 「構造化クエリを使用した検索」（158 ページ）

4.4 文字列クエリを使用した検索

文字列クエリはシンプルであっても強力なテキスト文字列を使用します。通常このテキスト文字列は、アプリケーションにおいてユーザーが検索ボックスに入力するクエリテキストとなります。このセクションでは、次の内容を取り上げます。

- 文字列クエリの概要
- 例：基本的な文字列クエリ
- 文字列クエリでの制約の使用
- 例：文字列クエリでの制約の使用
- カスタム制約パーサーの使用
- 例：カスタム制約パーサー
- 追加情報

4.4.1 文字列クエリの概要

MarkLogic サーバー Search API のデフォルトの検索文法により、「cat」、「cat AND dog」、または「cat NEAR dog」などのシンプルな検索をすばやく作成できます。そのような文字列クエリは、ユーザーが検索ボックスに入力するクエリテキストを表現しています。

デフォルト文法として、AND、OR、NOT、NEAR などの演算子とグループ化を利用できます。文法の詳細については、『Search Developer's Guide』の「Searching Using String Queries」を参照してください。

Node.js クライアントでは、`queryBuilder.parsedFrom` メソッドで文字列クエリを実行します。例えば、語句「cat」および「dog」が含まれているドキュメントにマッチさせるクエリには、次の queryBuilder 呼び出しを使用します。

```
qb.parsedFrom('cat AND dog')
```

詳細については、「例：基本的な文字列クエリ」（141 ページ）および Node.js API リファレンスを参照してください。

デフォルトで、DatabaseClient.documents.query は、ドキュメントのコンテンツを含んだ、マッチしたドキュメントごとに 1 つのドキュメントディスクリプタの配列を返します。対象となるドキュメントとその数を制御したり、スニペットやファセットを返したり、ドキュメント全体ではなく結果のサマリを返すなど、さまざまな方法で検索をより細かく設定できます。詳細については、「クエリ結果の詳細設定」（189 ページ）を参照してください。

文字列文法では、クエリタームに対する検索制約も利用できます。例えば、形式が *constraintName:value* または *constraintName relationalOp value* のタームを含めることで、マッチの対象を値が制約を満たす場合に制限できます。*ConstraintName* は、クエリに設定する制約の名前です。

例えば、「location」というワード制約を同じ名前の JSON プロパティに対して定義した場合、文字列クエリ「location:oslo」は、ターム「oslo」が `location` プロパティの値の場合のみマッチします。

同様に、レンジ制約を数値プロパティに対して定義し、「votes」という名前にバインドした場合、プロパティ値に対して「votes GT 5」などの関係式を含めることができます。

Node.js クライアントは、クエリで使用可能な名前に制約定義をバインドするパースバインドによる、文字列クエリへの制約をサポートしています。そのようなバインドの定義には、`queryBuilder.parseBindings` 関数を使用します。次に例を示します。

```
qb.parsedFrom(theQueryString, qb.parseBindings(binding
definitions...))
```

詳細については、「文字列クエリでの制約の使用」（143 ページ）および「カスタム制約パーサーの使用」（147 ページ）を参照してください。

4.4.2 例：基本的な文字列クエリ

次のスクリプト例は、データベースに「サンプルデータのロード」（205 ページ）のデータが入力されていることを前提としています。スクリプトは、語句「oslo」が含まれているすべてのドキュメントを検索します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');

var db = marklogic.createDatabaseClient(my.connInfo);
var qb = marklogic.queryBuilder;

db.documents.query(
    qb.where(qb.parsedFrom('oslo'))
).result( function(results) {
  console.log(JSON.stringify(results, null, 2));
});
```

この検索は、ドキュメントディスクリプタの配列を返します。マッチするドキュメントごとに 1 つのディスクリプタとなります。各ディスクリプタには、ドキュメントコンテンツが含まれています。

例えば、ファイル `string-search.js` に上記のスクリプトが含まれている場合、次のコマンドは以下の結果を生成します。検索は、ノルウェーのオスロにいる投稿者に対応する 2 つのドキュメントとマッチします。

```
$ node string-search.js
[
  {
    "uri": "/contributors/contrib1.json",
    "category": "content",
    "format": "json",
    "contentType": "application/json",
    "contentLength": "230",
    "content": {
      "Contributor": {
        "userName": "souser10002@email.com",
        "reputation": 446,
        "displayName": "Lars Fosdal",
        "originalId": "10002",
        "location": "Oslo, Norway",
        "aboutMe": "Software Developer since 1987, mainly
using Delphi.",
        "id": "sou10002"
      }
    }
  },
  {
    "uri": "/contributors/contrib2.json",
    "category": "content",
    "format": "json",
    "contentType": "application/json",
    "contentLength": "202",
    "content": {
      "Contributor": {
        "userName": "souser1000634@email.com",
        "reputation": 272,
        "displayName": "petrumo",
        "originalId": "1000634",
        "location": "Oslo, Norway",
        "aboutMe": "Developer at AspiroTV",
        "id": "sou1000634"
      }
    }
  }
]
```

ドキュメントコンテンツの代わりに検索サマリを返すには、
queryBuilder.withOptions を使用して categories を 'none' に設定します。
次に例を示します。

```
db.documents.query(

qb.where(qb.parsedFrom('oslo')).withOptions({categories:
'none'})
)
```

結果は、マッチ数（2）と各ドキュメント内でマッチするテキストのスニペットが含まれた検索サマリになります。

```
[{
  "snippet-format": "snippet",
  "total": 2,
  "start": 1,
  "page-length": 10,
  "results": [...snippets here...],
  "qtext": "oslo",
  "metrics": {
    "query-resolution-time": "PT0.005347S",
    "facet-resolution-time": "PT0.000067S",
    "snippet-resolution-time": "PT0.001523S",
    "total-time": "PT0.007753S"
  }]
```

また、別の方法で結果を詳細に設定することもできます。詳細については、「クエリ結果の詳細設定」（189 ページ）を参照してください。

4.4.3 文字列クエリでの制約の使用

文字列クエリインターフェイスを使用すると、クエリの一部の解釈方法を定義するパースバインドを作成できます。クエリタームにバインド名がプレフィックスされている場合は、関連付けられている制約がそのタームの検索に適用されるように、名前と検索制約間のバインドを定義できます。パースバインドは、ワード、値、レンジ、コレクション、スコープの各制約に対して作成できます。

例えば、名前「rep」と制約間のバインドを定義すると、「reputation」という名前の JSON プロパティ内のマッチする値に検索を制限できます。この場合、文字列クエリに形式が rep:*value* のタームが含まれている場合、その制約が値の検索に適用されます。そのため、次のタームは「値が 120 の reputation プロパティのすべての出現を検出する」ことを意味します。

```
rep:120
```

詳細については、『Search Developer's Guide』の「Using Relational Operators on Constraints」を参照してください。

> **注：** ここで使用した reputation に対する制約などのレンジ制約では、対応するレンジインデックスが利用されている必要があります。詳細については、「インデックス付け」（136 ページ）を参照してください。

パースバインドを作成して適用するには、次の手順に従います。完全な例については、「例：文字列クエリでの制約の使用」（145 ページ）を参照してください。

1. `queryBuilder.bind` または `queryBuilder.bindDefault` を呼び出して、バインド名の仕様を作成します。例えば次の呼び出しは、名前「rep」のバインド名の仕様を作成します。

   ```
   qb.bind('rep')
   ```

2. いずれかの `queryBuilder` バインドビルダーメソッド（`collection`、`range`、`scope`、`value`、または `word`）を呼び出してバインド名の仕様を渡し、名前（またはデフォルト）と制約間のバインドを作成します。例えば次の呼び出しは、名前 'rep' と JSON プロパティ名 'reputation' に対する値制約間のバインドを作成します。

   ```
   qb.value('reputation', qb.bind('rep'))
   ```

3. `queryBuilder.ParseBindings` を使用して、バインドを `queryBuilder.parseBindings` オブジェクトにバンドルします。次に例を示します。

   ```
   qb.parseBindings(
     qb.value('reputation', qb.bind('rep')), ...more
   bindings..
   )
   ```

4. パースバインドを `queryBuilder.parsedFrom` の 2 番目のパラメータとして渡して、それらを特定のクエリに適用します。次に例を示します。

   ```
   qb.parsedFrom('rep:120',
     qb.parseBindings(
       qb.value('reputation', qb.bind('rep')), ...more
   bindings..
     )
   )
   ```

また、queryBuilder.bindEmptyAs を使用して、クエリ文字列が空のときの動作を定義するバインドを作成することもできます。すべての結果を返す、または一切返さないように指定できます。デフォルトでは、結果は返されません。slice 指定子なしのクエリはマッチするドキュメントを返すため、空のクエリバインドを all-results に設定すると、空のクエリがデータベース内のすべてのドキュメントを取得するようになります。

次の例は、クエリテキストが空の文字列で、空のクエリバインドが all-results を示すため、すべての検索結果を返します。queryBuilder.slice を呼び出すと、クエリは最大 5 つのドキュメントを返します。

```
db.documents.query( qb.where(
  qb.parsedFrom('',
    qb.parseBindings(
     qb.bindEmptyAs('all-results')
  ))
)).slice(0,5)
```

4.4.4 例：文字列クエリでの制約の使用

この例は、カスタムパースバインドのルールを定義し、それらを文字列クエリベースの検索に適用します。この例は、「文字列クエリでの制約の使用」（143 ページ）で説明する機能を示したものです。

この例では、marklogic-samplestack アプリケーションから生成したデータを使用します。シードデータには、次の形式の「contributor」JSON ドキュメントが含まれています。

```
{ "com.marklogic.samplestack.domain.Contributor": {
    "userName": string,
    "reputation": number,
    "displayName": string,
    "originalId": string,
    "location": string,
    "aboutMe":  string,
    "id": string
} }
```

スクリプト例は、次のパースバインドを検索に適用します。

- ターム「rep」は、reputation JSON プロパティの値に対応します。レンジ制約にバインドされているため、「rep > 100」などの関係式で使用できます。この制約は、次のバインド定義によって表現します。

```
qb.range('reputation', qb.datatype('int'), qb.bind('rep'))
```

- 別の制約によってカバーされていない無修飾タームは、aboutMe JSON プロパティのワードクエリとマッチするように制約されます。この制約は、次のバインド定義によって表現します。

```
qb.word('aboutMe', qb.bindDefault())
```

データベース設定には、スカラー型「int」を持つ reputation JSON プロパティの要素レンジインデックスが含まれます。このインデックスは、reputation に対するレンジ制約をサポートするのに必要です。

バインドと設定をこのように組み合わせることにより、次のクエリテキストは「aboutMe」プロパティで「marklogic」が出現するドキュメントとマッチするようになります。ターム「marklogic」は、制約名によって修飾されていないため、無修飾タームです。

```
"marklogic"
```

次のクエリテキストは、「reputation」プロパティの値が 50 よりも大きい場合にドキュメントにマッチします。

```
marklogic AND rep GT 50
```

これらの節を組み合わせて使用することで、aboutMe プロパティに「marklogic」が含まれていて、reputation プロパティが 50 よりも大きいすべてのドキュメントをマッチさせることができます。

```
marklogic AND rep GT 50
```

バインドなしでは、上記のクエリは、任意の場所に語句「marklogic」があるドキュメントにマッチします。サブ式「rep GT 50」はワード「rep」と「50」を比較するため無意味です。

次のスクリプトは、バインドを作成して、それらを上記の検索テキストに適用します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');

var db = marklogic.createDatabaseClient(my.connInfo);
var qb = marklogic.queryBuilder;

db.documents.query( qb.where(
  qb.parsedFrom('marklogic AND rep GT 50',
    qb.parseBindings(
      qb.word('aboutMe', qb.bindDefault()),
      qb.range('reputation', qb.datatype('int'),
qb.bind('rep'))
  ))
)).result(function (documents) {
  console.log(JSON.stringify(documents[0].content, null,
2));
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

marklogic-samplestack シードデータに対してクエリを実行すると、1 人の投稿者がマッチして、次のような出力が生成されます。

```
{
  "Contributor": {
    "userName": "souser1601813@email.com",
    "reputation": 91,
    "displayName": "grechaw",
    "originalId": "1601813",
    "location": "Occidental, CA",
    "aboutMe": "XML (XQuery, Java, XML database) software
engineer at MarkLogic.Hardcore accordion player.",
    "id": "sou1601813"
  }
}
```

4.4.5 カスタム制約パーサーの使用

API は、ワード、値、レンジ、コレクション、およびスコープの各制約パーシングのバインドをサポートしています。これらの制約タイプではアプリケーションのニーズを満たさない場合は、カスタム制約パーサーに対するバインドを作成できます。パーサーは、『Search Developer's Guide』の「Creating a Custom Constraint」の説明に従って実装します。

Node.js クライアントを使用してカスタム制約パーサーを文字列クエリに適用するには、次の手順に従います。

1. カスタム制約パーサーを実装する XQuery モジュールを作成します。構造化クエリに対応したパーサーインターフェイスを使用します。詳細については、『Search Developer's Guide』の「Implementing a Structured Query parse Function」を参照してください。以下に示す命名規則に従う必要があります。

2. `DatabaseClient.config.query.custom.write` を使用して、パーサー XQuery ライブラリモジュールを REST API インスタンスに関連付けられている modules データベースにインストールします。詳細については、「例：カスタム制約パーサー」（148 ページ）を参照してください。

3. `queryBuilder.parseFunction` を使用して、制約名とカスタムパーサー間のパースバインドを作成します。

Node.js クライアント API を使用したカスタム制約の実装では、次の命名規則に従う必要があります。

- パース関数には、`parse` という名前を付ける必要があります。
- 開始および終了ファセット関数が存在する場合は、それぞれ `start-facet` および `finish-facet` にしなければなりません。
- モジュールの名前空間は、`http://marklogic.com/query/custom/`*`yourModuleName`* でなければなりません。*`yourModuleName`* は任意の名前です。

4.4.6 例：カスタム制約パーサー

この例では、Node.js クライアント API でカスタム制約パーサーを実装、インストール、および使用する方法を示します。詳細については、「カスタム制約パーサーの使用」（147 ページ）を参照してください。

この例は、`marklogic-samplestack` シードデータをベースにしています。データは、データベースディレクトリ `/contributors/` にインストールされている投稿者ドキュメントと、データベースディレクトリ `/questions/` にインストールされている質問ドキュメントで構成されています。

この例の制約では、クエリテキストに `cat:c` または `cat:q` という形式のタームを含めることで、検索の対象を投稿者または質問カテゴリに制約できるようにしています。名前「cat」は、queryBuilder パースバインドを使用してカスタム制約にバインドされています。制約パーサーでは、値「c」と「q」をそれぞれ対応する投稿者データと質問データとして定義しています。

この例では次の手順について説明します。

- 制約パーサーの実装
- 制約パーサーのインストール
- 文字列クエリでのカスタム制約の使用

4.4.6.1 制約パーサーの実装

次の XQuery モジュールは、制約パーサーを実装します。ファセット処理関数は提供されていません。パーサーは、呼び出し元が提供したカテゴリ名に基づいて `directory-query` を生成します。モジュールは、クエリテキストに表示されるカテゴリ名と、`categories` 変数内の対応するデータベースディレクトリの間のマッピングを維持します。

```
xquery version "1.0-ml";

module namespace my =
"http://marklogic.com/query/custom/ss-cat";
import module namespace search =
  "http://marklogic.com/appservices/search"
  at "/MarkLogic/appservices/search/search.xqy";

(: The category name to directory name mapping:)
declare variable $my:categories :=
  map:new((
    map:entry("c", "/contributors/"),
    map:entry("q", "/questions/")
  ));

(: parser implementation :)
declare function my:parse(
  $query-elem as element(),
  $options as element(search:options)
) as schema-element(cts:query)
{
let $query :=
  <root>{
    let $cat := $query-elem/search:text/text()
    let $dir :=
      if (map:contains($my:categories, $cat))
      then map:get($my:categories, $cat)[1]
      else "/"
    return cts:directory-query($dir, "infinity")
  }</root>/*
return
```

```
(: add qtextconst attribute so that search:unparse will
work -
   required for some search library functions :)
element { fn:node-name($query) }
  { attribute qtextconst {
      fn:concat(
        $query-elem/search:constraint-name, ":",
        $query-elem/search:text/text()) },
    $query/@*,
    $query/node()}
};
```

4.4.6.2 制約パーサーのインストール

次のスクリプトは、実装が ss-cat.xqy という名前のファイルに保存されていることを前提として、制約パーサーモジュールを modules データベースにインストールします。インストールは、DatabaseClient.config.query.custom.write を呼び出して実行します。最初のパラメータとして渡すモジュール名のベース名は、モジュールの名前空間の宣言のモジュール名と同じでなければなりません（ss-cat）。

```
var fs = require('fs');
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.config.query.custom.write(
  'ss-cat.xqy',
  [ {'role-name': 'app-user', capabilities: ['execute']} ],
  fs.createReadStream('./ss-cat.xqy')
).result(function(response) {
  console.log('Installed module ' + response.path);
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

install-parser.js という名前のファイルにスクリプトを保存した場合、スクリプトを実行すると次のような結果が生成されます。

```
$ node install-parser.sj
Installed module /marklogic/query/custom/ss-cat.xqy
```

4.4.6.3 文字列クエリでのカスタム制約の使用

この制約を使用するには、queryBuilder.parseFunction によって作成されたパースバインドをクエリに含めます。最初のパラメータは、実装をインストールしたときに使用したモジュール名とマッチしなければなりません。

例えば、次の呼び出しは、上記の手順でインストールした名前「cat」をカスタム制約パーサーにバインドし、クエリに「cat:c」または「cat:q」という形式のタームを含められるようにしています。

```
qb.parseFunction('ss-cat.xqy', qb.bind('cat'))
```

モジュール名（ss-cat.xqy）は、config.query.custom.write の最初のパラメータとして渡されたモジュール名と同じであることに注目してください。

次のスクリプトは、カスタム制約を使用して「marklogic AND cat:c」という形式のクエリテキストを指定し、投稿者カテゴリ（「cat:c」）内のドキュメントで「marklogic」の出現を検索します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');

var db = marklogic.createDatabaseClient(my.connInfo);
var qb = marklogic.queryBuilder;

db.documents.query( qb.where(
  qb.parsedFrom('marklogic AND cat:c',
    qb.parseBindings(
      qb.parseFunction('ss-cat.xqy', qb.bind('cat'))
  ))
)).result(function (documents) {
  for (var i in documents)
    console.log(JSON.stringify(documents[i].content, null,
2));
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

スクリプトを ss-cat.js という名前のファイルに保存して実行すると、この検索から 2 つの投稿者ドキュメントが返されます。

```
$ node ss-cat.js
{
  "Contributor": {
    "userName": "souser1248651@email.com",
    "reputation": 1,
```

```
    "displayName": "Nullable",
    "originalId": "1248651",
    "location": "Ogden, UT",
    "aboutMe": "...My current work includes work with
MarkLogic
       Application Server (Using XML, Xquery, and Xpath),
WPF/C#,
       and Android Development (Using Java)...",
    "id": "sou1248651"
  }
}
{
  "Contributor": {
    "userName": "souser1601813@email.com",
    "reputation": 91,
    "displayName": "grechaw",
    "originalId": "1601813",
    "location": "Occidental, CA",
    "aboutMe": "XML (XQuery, Java, XML database) software
engineer
       at MarkLogic.Hardcore accordion player.",
    "id": "sou1601813"
  }
}
```

「cat:c」タームを削除してクエリテキストを「marklogic」のみにすると、検索から質問ドキュメントも返されるようになります。

詳細およびその他の例については、『Search Developer's Guide』の「Creating a Custom Constraint」を参照してください。

4.4.7 追加情報

カスタム制約の作成および使用方法の詳細については、次のリソースを参照してください。

- Node.js API リファレンスの次の関数：
  - `queryBuilder.parsedFrom`
  - `queryBuilder.parseBindings`
  - `queryBuilder.parseFunction`
  - `queryBuilder.binding`
  - `queryBuilder.BindingParam` 引数を受け付ける任意のクエリビルダー：`queryBuilder.collection`、`queryBuilder.range`、`queryBuilder.scope`、`queryBuilder.value`、`queryBuilder.word` など

- 『Search Developer's Guide』の「Searching Using String Queries」
- 『Search Developer's Guide』の「Creating a Custom Constraint」
- 『Search Developer's Guide』の「Constraint Options」

4.5 Query By Example を使用した検索

このセクションでは、Query By Example（QBE）を使用した JSON ドキュメントの検索に関する次のトピックについて説明します。

- QBE の概要
- queryBuilder を使用した QBE の作成
- QBE を使用した XML コンテンツのクエリ
- 追加情報

4.5.1 QBE の概要

Query By Example を使用すると、データベース内のドキュメントの構造と類似する検索基準を使用して、「類似のドキュメント」を探すクエリのプロトタイプをすばやく作成できます。

例えば、ドキュメントに `author` プロパティが含まれている場合、次の RAW モードの QBE は `author` 値が「Mark Twain」のドキュメントにマッチします。

```
{ $query: { author: "Mark Twain" } }
```

Node.js クライアント API で QBE を作成するには、`queryBuilder.byExample` を使用します。JSON コンテンツを扱う場合、このインターフェイスは、コンテンツでモデル化された個々の検索基準（`{ author: "Mark Twain" }`）または `$query` オブジェクト全体を入力として受け付けます。次に例を示します。

```
db.documents.query( qb.where(
  qb.byExample( {author: 'Mark Twain'} ))
)
```

XML を検索するときに、シリアライズされた XML QBE を渡すことができます。詳細については、「QBE を使用した XML コンテンツのクエリ」（156 ページ）を参照してください。

QBE が公開する MarkLogic サーバー Search API のサブセットには、値クエリ、レンジクエリ、およびワードクエリが含まれています。QBE は、AND、OR、NOT、大なり、小なり、等号など、値に対する論理演算子と関係演算子もサポートしています。

QBE と Node.js API は、ドキュメントのコンテンツをクエリする目的でのみ使用できます。メタデータ検索はサポートされていません。また、フィールドも検索できません。メタデータをクエリしたり、フィールドを検索したりするには、`queryBuilder.collection`、`queryBuilder.property`、または `queryBuilder.field` などの他の `queryBuilder` 関数を使用します。metadataValues メタデータカテゴリを検索するには、フィールドクエリを使用します。

このガイドでは、QBE の概要についてのみ説明します。詳細については、『Search Developer's Guide』の「Searching Using Query By Example」を参照してください。

4.5.2 queryBuilder を使用した QBE の作成

QBE を作成するには、`queryBuilder.byExample` を呼び出して、1 つあるいは複数の検索基準パラメータを渡します。XML ドキュメントを扱う場合、完全な形式の QBE を渡すこともできます。詳細については、「QBE を使用した XML コンテンツのクエリ」（156 ページ）を参照してください。

例えば、「サンプルデータのロード」（205 ページ）で作成したドキュメントは `location` プロパティを含みます。このデータに対して次のスクリプトを実行することで、ノルウェーのオスロに住むすべての投稿者を検索できます。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');

var db = marklogic.createDatabaseClient(my.connInfo);
var qb = marklogic.queryBuilder;

db.documents.query(
    qb.where(qb.byExample( {location: 'Oslo, Norway'} ))
).result( function(results) {
  console.log(JSON.stringify(results, null, 2));
});
```

`qb.byExample` に渡した検索基準は、値が「Oslo, Norway」の `location` プロパティを含むドキュメントのみにマッチします。{*propertyName*: *value*} という形式の QBE 基準は値クエリであるため、値が正確に「Oslo, Norway」と一致する必要があります。

またワードクエリやレンジクエリなど、ドキュメントをモデル化する他のクエリタイプを作成できます。例えば、ワードクエリを使用すれば、`location` 値のバリエーションを許容できるように上記の制約を緩和できます。また、`reputation` の値が 400 を超える投稿者にのみマッチさせる基準を追加することもできます。次の表は、この検索を実現するために使用可能な QBE 基準とその説明を示したものです。

QBE 基準	説明
`location: {$word : 'oslo'}`	語句「oslo」が `location` の値に出現したときにマッチします。`$word` は、ワードクエリを表す予約済みのプロパティ名です。ワードクエリを使用する場合、マッチでは大文字 / 小文字が区別されません。また、値に他のワードが含まれているかどうかは問いません。詳細については、『Search Developer's Guide』の「Word Query」を参照してください。
`reputation: {$gt : 400}`	reputation の値が 400 を超えるドキュメントにマッチします。`$gt` は、「大なり」比較演算子を表す予約済みのプロパティ名です。詳細については、『Search Developer's Guide』の「Range Query」を参照してください。
`$filtered: true`	フィルタリングありの検索を実行します。QBE は、パフォーマンスを最大限に高められるようにデフォルトではフィルタリングなしの検索を使用します。ただし、`{$gt : 400}` などのレンジクエリでは、フィルタリングありの検索を行うか、レンジインデックスを利用する必要があるため、フィルタリングありの検索を有効にする必要があります。詳細については、『Search Developer's Guide』の「How Indexing Affects Your Query」を参照してください。

次のスクリプトは、このような基準を 1 つの QBE にまとめたものです。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);
var qb = marklogic.queryBuilder;

db.documents.query( qb.where(
  qb.byExample( {
    location: {$word : 'oslo'},
    reputation: {$gt : 400},
    $filtered: true
  }))
).result( function(results) {
  console.log(JSON.stringify(results, null, 2));
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

基準は、個別のオブジェクトまたはオブジェクトの配列として byExample に渡すことができます。例えば、次の呼び出しは、上記の byExample 呼び出しと同じです。

```
// criteria as individual objects
qb.byExample(
  {location: {$word : 'oslo'}},
  {reputation: {$gt : 400}},
  {$filtered: true}
)

// criteria as an array of objects
qb.byExample([
  {location: {$word : 'oslo'}},
  {reputation: {$gt : 400}},
  {$filtered: true}
))
```

これらの例での queryBuilder.byExample への入力は、RAW モードの QBE の $query 部分の検索基準に対応します。詳細については、『Search Developer's Guide』の「Constructing a QBE with the Node.js QueryBuilder」を参照してください。

QBE の RAW モードの $query 部分は、$query プロパティを持つオブジェクトを提供することで、queryBuilder.byExample に渡すこともできます。次に例を示します。

```
// raw QBE $query
qb.byExample(
  { $query: {
      location: {$word : 'oslo'},
      reputation: {$gt : 400},
      $filtered: true
  }}
)
```

4.5.3 QBE を使用した XML コンテンツのクエリ

「queryBuilder を使用した QBE の作成」（154 ページ）の説明に従って、JavaScript クエリ基準を queryBuilder.byExample に渡します。これは JSON コンテンツだけにマッチする JSON QBE を暗黙的に作成します。デフォルトでは、QBE は、QBE と同じコンテンツタイプのドキュメントだけにマッチします。つまり、JSON で表された QBE は JSON ドキュメントにマッチし、XML で表された QBE は XML ドキュメントにマッチします。シリアライズされた XML QBE を使用するか、$format QBE プロパティを「xml」に設定して、XML コンテンツを検索することはこれまでどおり可能です。

QBE を使用して XML コンテンツを検索するには、次のいずれかの方法を使用します。

- シリアライズされた XML QBE を `queryBuilder.byExample` への入力として渡します。クエリが XML 名前空間に依存している場合は、この手法を使用する必要があります。次に例を示します。

```
qb.byExample(
  '<q:qbe
xmlns:q="http://marklogic.com/appservices/querybyexample">
'+
    '<q:query>' +
      '<my:contributor
xmlns:my="http://marklogic.com/example">' +
        '<my:location><q:word>oslo</q:word></my:location>'
+
      '</my:contributor>' +
      '<my:contributor
xmlns:my="http://marklogic.com/example">' +
        '<my:reputation><q:gt>400</q:gt></my:reputation>'
+
      '</my:contributor>' +
    '<q:filtered>true</q:filtered>' +
    '</q:query>' +
  '</q:qbe>'
)
```

- `$format` プロパティと値「xml」とを含んだ完全な形式の QBE を表す `queryBuilder.byExample` に、JavaScript オブジェクトを渡します。名前空間にない XML コンテンツを扱う場合は、この手法だけを使用できます。次に例を示します。

```
qb.byExample({
  $query: {
    location: {$word : 'oslo'},
    reputation: {$gt : 400},
    $filtered: true
  },
  $format: 'xml'
})
```

どちらの場合でも、`queryBuilder.byExample` に渡されたデータは、JSON ドキュメントの検索時のクエリ基準だけではなく、完全な形式の QBE である必要があります（XML の場合、シリアライズされた QBE でもかまいません）。構文については、『Search Developer's Guide』の「Searching Using Query By Example」を参照してください。

XML にマッチするすべての検索と同様に、検索で返される XML コンテンツはシリアライズされ、文字列として返されます。

4.5.4 追加情報

QBE の作成および使用方法の詳細については、次のリソースを参照してください。

- Node.js API リファレンスの「`queryBuilder.byExample`」
- 『Search Developer's Guide』の「Searching Using Query By Example」

4.6 構造化クエリを使用した検索

`queryBuilder.Query` を返す `queryBuilder` 関数は、構造化クエリのサブクエリを作成します。構造化クエリとは、検索式の抽象構文ツリー表現です。構造化クエリは、文字列クエリまたは QBE の表現力が十分ではない場合、またはクエリを途中で遮って拡張または変更する必要がある場合に使用します。詳細については、『Search Developer's Guide』の「Structured Query Overview」を参照してください。

- 基本的な使用方法
- 例：構造化クエリの使用
- ビルダーメソッドタクソノミーのリファレンス
- クエリパラメータのヘルパー関数
- 検索結果の詳細設定

4.6.1 基本的な使用方法

`queryBuilder.where` などの `queryBuilder.BuiltQuery` を作成する関数に、1 つあるいは複数の `queryBuilder.query` オブジェクトを渡すと、構造化クエリの作成にこれらのクエリが使用されます。構造化クエリとは、検索式の抽象構文ツリー表現です。構造化クエリは、文字列クエリまたは QBE の表現力が十分ではない場合、またはクエリを途中で遮って拡張または変更する必要がある場合に使用します。詳細については、『Search Developer's Guide』の「Structured Query Overview」を参照してください。

構造化クエリは、`queryBuilder` のビルダーメソッドを使用して作成する 1 つあるいは複数の検索基準で構成されています。ビルダーのタクソノミーとそれぞれの例については、「ビルダーメソッドタクソノミーのリファレンス」（162 ページ）を参照してください。

例えば、次のコードスニペットは、クエリを構造化クエリとして MarkLogic サーバーに送ります。このクエリは、ターム「marklogic」も含まれているデータベースディレクトリ「/contributors/」内のドキュメントにマッチします。

```
db.documents.query(
  qb.where(
    qb.and(qb.directory("/contributors/",
           qb.term("marklogic"))
))
```

結果をカスタマイズするには、文字列クエリや QBE を使用して検索を行う場合と同様に、queryBuilder の結果の詳細設定メソッドを使用します。詳細については、「検索結果の詳細設定」（170 ページ）を参照してください。

4.6.2 例：構造化クエリの使用

次の例は、「サンプルデータのロード」（205 ページ）のサンプルデータに基づいています。

この例では、構造化クエリビルダーを使用して複合クエリを作成するいくつかの方法を示します。

次の例は、/contributors/ データベースディレクトリで、「marklogic」のタームを含むドキュメントを検索します。デフォルトで、クエリは、マッチしたドキュメントを返します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');

var db = marklogic.createDatabaseClient(my.connInfo);
var qb = marklogic.queryBuilder;

db.documents.query(
  qb.where(
    qb.and(
      qb.directory('/contributors/'),
      qb.term('marklogic')
    )
  )
).result( function(results) {
  console.log(JSON.stringify(results, null, 2));
});
```

このクエリは、ドキュメントディスクリプタの配列を返します。マッチするドキュメントごとに 1 つのディスクリプタとなります。サンプルデータには、/contributors/contrib3.json と /contributors/contrib4.json にマッチする 2 つのドキュメントが含まれているので、次のような出力になります。ドキュメントディスクリプタの content プロパティには、マッチしたドキュメントのコンテンツが含まれます。

```
[
  {
    "uri": "/contributors/contrib3.json",
    "category": "content",
    "format": "json",
    "contentType": "application/json",
    "contentLength": "323",
    "content": {
      "Contributor": {
        "userName": "souser1248651@email.com",
        "reputation": 1,
        "displayName": "Nullable",
        "originalId": "1248651",
        "location": "Ogden, UT",
        "aboutMe": "...My current work includes work with
MarkLogic
         Application Server (Using XML, Xquery, and Xpath),
WPF/C#,
          and Android Development (Using Java)...",
        "id": "sou1248651"
      }
    }
  },
  {
    "uri": "/contributors/contrib4.json",
    "category": "content",
    "format": "json",
    "contentType": "application/json",
    "contentLength": "273",
    "content": {
      "Contributor": {
        "userName": "souser1601813@email.com",
        "reputation": 91,
        "displayName": "grechaw",
        "originalId": "1601813",
        "location": "Occidental, CA",
        "aboutMe": "XML (XQuery, Java, XML database)
```

```
      software
                engineer at MarkLogic.Hardcore accordion
      player.",
              "id": "sou1601813"
            }
          }
        }
      ]
```

queryBuilder.where は渡されるクエリを暗黙的に AND で結合するので、必要に応じて queryBuilder.and の呼び出しを削除できます。例えば、元のクエリを次のように書き換えても、同じ結果を得ることができます。

```
db.documents.query(
  qb.where(
    qb.directory('/contributors/'),
    qb.term('marklogic')
  )
```

文字列クエリを、1 つあるいは複数の構造化クエリビルダーの結果と組み合わせることもできます。例えば、qb.where に渡されるクエリリストに qb.parsedFrom('java') を追加すれば、「java」も含んだドキュメントに結果をさらに制限できます。文字列クエリは、他のクエリタームと暗黙的に AND で結合されます。クエリを次のように変更した場合、結果セットには /contributors/contrib3.json だけが含まれます。

```
db.documents.query(
  qb.where(
    qb.directory('/contributors/'),
    qb.term('marklogic'),
    qb.parsedFrom('java')
  )
```

queryBuilder インターフェイスには、インデックスリファレンスなどのより複雑なクエリコンポーネントの構築を容易にするヘルパー関数が含まれます。詳細については、「クエリパラメータのヘルパー関数」（168 ページ）を参照してください。

他のクエリタイプと同様に、queryBuilder.slice と queryBuilder.withOptions を使用して、結果セットを詳細に設定できます。詳細については、「クエリ結果の詳細設定」（189 ページ）を参照してください。

4.6.3 ビルダーメソッドタクソノミーのリファレンス

構造化クエリでは、「クエリのタイプ」（135 ページ）で説明されているすべてのクエリタイプが、ビルダーメソッドを通じて明示的に公開されています。このセクションは、次のカテゴリに基づいて必要なビルダーを見つけるためのクイックリファレンスです。

- 基本的なコンテンツクエリ
- 論理構成子
- 場所修飾子
- ドキュメントセレクタ

ビルダー関数が受け付けるパラメータで示されているように、ほとんどのクエリタイプは相互に組み合わせて使用できます。詳細については、Node.js API リファレンスの `queryBuilder` インターフェイスを参照してください。

`queryBuilder` インターフェイスにより、クエリの基盤を形成する構造的詳細を理解することなく、複雑な構造化クエリを構築できます。このガイドには、『Search Developer's Guide』の構造化クエリに関する「Syntax Reference」への相互参照が示されています。特定のクエリタイプのコンポーネントに関する詳細が必要な場合は参照してください。

4.6.3.1 基本的なコンテンツクエリ

基本的なコンテンツクエリは、「JSON のプロパティ A に値 B が入っている」または「'dog' という語句が含まれる任意のドキュメント」のように、コンテンツに関する検索基準を表します。このようなクエリはサブクエリを含まないため、複雑な複合クエリの構造内で「リーフ」として機能します。

次の表は、基本的なコンテンツクエリを作成する Node.js ビルダーのメソッドを示したものです。クエリの特定の要素に関する詳細が必要な場合に参照できるように、対応する RAW モードの JSON 構造化クエリタイプへのリンクが用意されています。RAW モードのクエリを作成する必要はありません。Node.js API が自動的に作成します。

queryBuilder 関数	例	構造化クエリのサブクエリ
term	`qb.term('marklogic')`	term-query
word	`qb.word('aboutMe', 'marklogic')`	word-query
value	`qb.value('tags','java')`	value-query
range	`qb.range('reputation', '>=', 100)`	range-query
geospatial	`qb.geospatial( qb.geoElement ('gElemPoint'), qb.latlon(50, 44) )`	geo-elem-query geo-elem-pair-query geo-attr-pair-query geo-json-property-query geo-json-property-pair-query geo-path-query
geospatial Region	`q.geospatialRegion( q.geoPath('/envelope/region'), 'intersects', q.circle(5, q.point(10,20)) )`	geo-region-path-query
geoElement	`qb.geospatial( qb.geoElement ('gElemPoint'), qb.latlon(50, 44) )`	geo-elem-query

queryBuilder 関数	例	構造化クエリのサブクエリ

geoElementPair	qb.geospatial(
	  qb.geoElementPair(
	  'gElemPair',
	  'latitude',
	  'longitude'),
	  qb.latlon(50, 44)
	)
		geo-elem-pair-query

geoAttrPair	qb.geospatial(
	  qb.geoAttributePair(
	  'gAttrPair',
	  'latitude',
	  'longitude'),
	  qb.circle(100, 240,
	144)
	)
		geo-attr-pair-query

geoProperty	q.geospatial(
	  q.geoProperty
	('gElemPoint'),
	  q.point(34, 88)
	)
		geo-json-property-query

geoProperty Pair	qb.geospatial(
	  qb.geoPropertyPair(
	  'gElemPair',
	  'latitude',
	  'longitude'),
	  qb.latlon(12, 5)
	)
		geo-json-property-pair-query

geoPath	q.geospatial(
	  q.geoPath('parent/child
	'),
	  q.latlon(12, 5)
	)
		geo-path-query

4.6.3.2 論理構成子

論理構成子とは、1 つあるいは複数のサブクエリを結合して 1 つの論理式にするクエリです。例えば、「query1 と query2 の両方にマッチするドキュメント」または「query1、query2、query3 のいずれかとマッチするドキュメント」などです。

次の表は、論理構成子の Node.js ビルダーメソッドを示したものです。クエリの特定の要素に関する詳細が必要な場合に参照できるように、対応する RAW モードの JSON 構造化クエリタイプへのリンクが用意されています。RAW モードのクエリを作成する必要はありません。Node.js API が自動的に作成します。

queryBuilder 関数	例	構造化クエリの サブクエリ

and	`qb.and(`
	
	`qb.word('text','marklogic'),`
	`  qb.value('tags', 'java')`
	`)`
		and-query

andNot	`qb.andNot(`
	
	`qb.word('text','marklogic'),`
	`  qb.value('tags', 'java')`
	`)`
		and-not-query

boost	`qb.boost(`
	
	`qb.word('text','marklogic'),`
	`  qb.word('title', 'json')`
	`)`
		boost-query

not	`qb.not(qb.term('marklogic'))`	not-query

notIn	`qb.notIn(`
	`  qb.word('text','json'),`
	`  qb.word('text', 'json documents')`
	`)`
		not-in-query

or	`qb.or(`
	
	`qb.value('tags','marklogic'),`
	`  qb.value('tags', 'nosql')`
	`)`
		or-query

4.6.3.3 場所修飾子

場所修飾子は、コンテンツ内のみ、メタデータ内のみ、指定された JSON プロパティまたは XML 要素に含まれている場合のみなど、サブクエリがマッチした場所に基づいて結果を制限するクエリです。例えば、「メタデータ内でこのサブクエリにマッチするもの」または「JSON プロパティ P でこのサブクエリにマッチするもの」などです。

次の表は、場所修飾子を作成する Node.js ビルダーメソッドを示したものです。クエリの特定の要素に関する詳細が必要な場合に参照できるように、対応する RAW モードの JSON 構造化クエリタイプへのリンクが用意されています。RAW モードのクエリを作成する必要はありません。Node.js API が自動的に作成します。

queryBuilder 関数	例	構造化クエリのサブクエリ

documentFragment	`qb.documentFragment(`
	`  qb.term('marklogic')`
	`)`
		document-fragment-query

locksFragment	`qb.documentFragment(`
	`  qb.term('marklogic')`
	`)`
		locks-fragment-query

near	`qb.near(`
	`  qb.term('marklogic'),`
	`  qb.term('xquery'), 5`
	`)`
		near-query

properties	`qb.properties(`
	`  qb.term('marklogic')`
	`)`
		properties-fragment-query

scope	`qb.scope(`
	`  'aboutMe',`
	`  qb.term('marklogic')`
	`)`
		container-query

4.6.3.4 ドキュメントセレクタ

ドキュメントセレクタは、ドキュメントのグループをマッチする対象としてコンテンツ（内容）本体ではなく、データベースの属性（コレクションのメンバーシップ、ディレクトリ、URI など）を使用するクエリです。例えば、「コレクション A および B 内のすべてのドキュメント」または「ディレクトリ D 内のすべてのドキュメント」などがあります。

次の表は、ドキュメントセレクタを作成する Node.js ビルダーメソッドを示したものです。クエリの特定の要素に関する詳細が必要な場合に参照できるように、対応する RAW モードの JSON 構造化クエリタイプへのリンクが用意されています。RAW モードのクエリを作成する必要はありません。Node.js API が自動的に作成します。

queryBuilder 関数	例	構造化クエリのサブクエリ

collection	`qb.and(`
	`  qb.collection('marklogicians'),`
	`  qb.term('java')`
	`)`
		collection-query

directory	`qb.and(`
	`  qb.directory('/contributors/'),`
	`  qb.term('java')`
	`)`
		directory-query

document	`qb.and(`
	`  qb.document(`
	
	`'/contributors/contrib1.json',`
	
	`'/contributors/contrib3.json'),`
	`  qb.term('norway')`
	`)`
		document-query

4.6.4 クエリパラメータのヘルパー関数

queryBuilder インターフェイスには、構造的に重要なサブクエリパラメータを構築するヘルパー関数が用意されています。

例えば、コンテナクエリ（queryBuilder.scope）は、JSON プロパティまたは XML 要素など、コンテナ（またはスコープ）を特定するディスクリプタを必要とします。ヘルパー関数 queryBuilder.property および queryBuilder.element を使用することで、scope 関数が必要とするコンテナディスクリプタを定義できます。

次のコードスニペットは、ターム「marklogic」が「aboutMe」という名前の JSON プロパティに出現した場合にマッチするコンテナクエリを作成します。ヘルパー関数 queryBuilder.property は、JSON プロパティ名の仕様を作成します。

```
db.documents.query(
  qb.where(
    qb.scope(qb.property('aboutMe'), qb.term('marklogic'))
  )
)
```

queryBuilder が提供する主なヘルパー関数は、次の表に示すとおりです。詳細については、Node.js API リファレンスおよび『Search Developer's Guide』を参照してください。

ヘルパー関数	目的
anchor	bucket ヘルパー関数の数値または dateTime レンジを定義します。詳細については、『Search Developer's Guide』の「Constrained Searches and Faceted Navigation」を参照してください。
attribute	range、word、value、および位置情報クエリビルダーなどのクエリビルダーで使用する XML 要素属性を指定します。
bucket	facet ビルダーで使用する数値または dateTime レンジバケットを定義します。詳細については、『Search Developer's Guide』の「Constrained Searches and Faceted Navigation」を参照してください。
datatype	インデックスリファレンスの曖昧さを回避するために、range クエリビルダーにおいて使用可能なインデックスタイプ（int、string など）を指定します。この処理が必要になるのは、同じドキュメントコンポーネントに対して異なるタイプのインデックスが複数存在する場合のみです。

ヘルパー関数	目的
element	scope、range、word、value、および位置情報クエリビルダーなどのクエリビルダーで使用する XML 要素を指定します。
facet	calculate 結果ビルダーで使用する検索ファセットを定義します。詳細については、『Search Developer's Guide』の「Constrained Searches and Faceted Navigation」を参照してください。
facetOptions	facet ビルダーで使用する追加のオプションを指定します。詳細については、『Search Developer's Guide』の「Facet Options」を参照してください。
field	range、word、value の各クエリビルダーで使用するドキュメントまたは metadataValues フィールドを指定します。詳細については、『Administrator's Guide』の「Fields Database Settings」を参照してください。
fragmentScope	range、scope、value、または word クエリのスコープをドキュメントコンテンツまたはドキュメントプロパティに制限します。
pathIndex	range または geoPath などのクエリビルダーのパスレンジインデックスを指定します。データベース設定には、対応するパスレンジインデックスが含まれている必要があります。詳細については、『Administrator's Guide』の「Understanding Path Range Indexes」を参照してください。
property	range、scope、value、word、geoProperty、および geoPropertyPair などのクエリビルダーの JSON プロパティ名を指定します。
qname	range、scope、value、word、geoElement、geoElementPair、geoAttributePair などのクエリビルダーの XML 要素 QName（ローカル名および名前空間 URI）を指定します。また、属性識別子の作成にも使用します。
rangeOptions	range クエリビルダーで使用可能な追加の検索オプションです。詳細については、Node.js API リファレンスおよび『Search Developer's Guide』の「Range Options」を参照してください。

ヘルパー関数	目的
`score`	`orderBy` 結果ビルダーで使用するレンジクエリの関連度スコアリングアルゴリズムを指定します。詳細については、『Search Developer's Guide』の「Including a Range or Geospatial Query in Scoring」を参照してください。
`sort`	`orderBy` 結果ビルダーで使用する検索結果ソート基準と順序を定義する `sort-order` クエリオプションと同等の関数を指定します。詳細については、『Search Developer's Guide』の「sort-order」を参照してください。
`termOptions`	word および value クエリビルダーで使用する term-option クエリオプションと同等の関数を指定します。詳細については、『Search Developer's Guide』の「Term Options」を参照してください。
`weight`	クエリに割り当てる修正された重み付けを指定します。`word` および `value` などのクエリビルダーで使用できます。詳細については、『Search Developer's Guide』の「Using Weights to Influence Scores」を参照してください。

4.6.5 検索結果の詳細設定

`queryBuilder` インターフェイスには、検索結果の詳細設定のための複数の関数が用意されています。例えば、返す結果の数、結果のソート方法、および検索ファセットを含めるどうかを指定できます。

`queryBuilder.Query` オブジェクトを返すクエリビルダーとは異なり、このような詳細設定関数は、通常、`queryBuilder.BuiltQuery` オブジェクトを返します。

結果修飾子の呼び出しは連結できます。次に例を示します。

```
db.documents.query(qb.where(someQuery).slice(0,5).orderBy
(...))
```

詳細については、「クエリ結果の詳細設定」（189 ページ）を参照してください。

次の表は、queryBuilder でサポートされている結果修飾子関数とその説明をまとめたものです。詳細については、Node.js API リファレンスを参照してください。

ヘルパー関数	目的
anchor	bucket ヘルパー関数の数値または dateTime レンジを定義します。詳細については、「検索ファセットの生成」（184 ページ）および『Search Developer's Guide』の「Constrained Searches and Faceted Navigation」を参照してください。
bucket	facet ビルダーで使用する数値または dateTime レンジバケットを定義します。詳細については、「検索ファセットの生成」（184 ページ）および『Search Developer's Guide』の「Constrained Searches and Faceted Navigation」を参照してください。
facet	calculate 結果ビルダーで使用する検索ファセットを定義します。詳細については、「検索ファセットの生成」（184 ページ）および『Search Developer's Guide』の「Constrained Searches and Faceted Navigation」を参照してください。
facetOptions	facet ビルダーで使用する追加のオプションを指定します。詳細については、「検索ファセットの生成」（184 ページ）および『Search Developer's Guide』の「Facet Options」を参照してください。
calculate	検索ファセットの仕様を作成します。詳細については、「検索ファセットの生成」（184 ページ）を参照してください。
orderBy	ソート順序とシーケンスを指定します。例えば、ソートの基準にする JSON プロパティ、XML 要素、XML 要素属性を指定できます。詳細については、『Search Developer's Guide』の「sort-order」を参照してください。
slice	結果セットから返すドキュメントのスライスと、結果に適用するサーバーサイド変換を定義します。詳細については、「クエリ結果の詳細設定」（189 ページ）を参照してください。
withOptions	クエリの詳細設定や微調整に使用可能なその他のオプションです。例えば、withOptions を使用して、マッチするドキュメントから取得するデータのカテゴリ（コンテンツやメタデータ）を指定したり、クエリのメトリックスを要求したり、トランザクション ID を指定したりします。

4.7 複合クエリを使用した検索

複合クエリとは、さまざまなクエリタイプとクエリオプションの組み合わせが可能なクエリオブジェクトです。ほとんどの検索は、複合クエリを使用せずに実行できます。例えば、`queryBuilder.parsedFrom` と `queryBuilder.Query` の結果を `queryBuilder.where` に渡すだけで、文字列クエリと構造化クエリを組み合わせられます。

この機能は、上級ユーザーがすでに Search API に慣れている場合や、次のいずれかの要件を抱えている場合に最も適しています。

- MarkLogic サーバーに以前に存続していたクエリオプションを、アプリケーションで使用する必要がある。
- クエリ時にクエリオプションに対する非常に詳細な制御が必要である（ほとんどのクエリオプションは、`queryBuilder` メソッドなどの Node.js API の他の部分ですでに公開されています。可能な場合は、複合クエリに頼らず、これらのインターフェイスを使用してください）。

Node.js クライアント API では、CombinedQueryDefinition が複合クエリをカプセル化します。この API は `CombinedQueryDefinition` 用のビルダーは提供しません。`CombinedQueryDefinition` は次の形式を取ります。ここで `search` プロパティは複合クエリを含み、残りのプロパティは必要に応じて結果のカスタマイズに使用できます。

```
{ search: {
    query: { structuredQuery },
    qtext: stringQuery,
    options: { queryOptions }
  },
  categories: [ resultCategories ],
  optionsName: persistedOptionsName,
  pageStart: number,
  pageLength: number,
  view: results
}
```

複合クエリの部分には、構造化クエリ、文字列クエリ、および Search API クエリオプションの任意の組み合わせを含めることができます。`CombinedQueryDefinition` ラッパーの設定で指定されたオプションと競合するオプションを、複合クエリ内部で指定した場合、ラッパーオプション設定が複合オプション内部の設定を上書きします。例えば、`search.options` に `'page-length':5` が含まれ、`search.pageLength` が 10 に設定されている場合、ページ長は 10 になります。

次の表では、複合クエリのプロパティについて説明します。

プロパティ名	説明
query	オプション。『Search Developer's Guide』の「Searching Using Structured Queries」で説明されている構文に従った構造化クエリ。
qtext	オプション。Search API 文字列クエリ構文に従った文字列クエリ。詳細については、「文字列クエリを使用した検索」（140 ページ）および『Search Developer's Guide』の「Searching Using String Queries」を参照してください。
options	オプション。1 つあるいは複数の Search API クエリオプション。詳細については、『Search Developer's Guide』の「Appendix: Query Options Reference」を参照してください。注：特定のオプションを使用するには、rest-admin ロールまたは同等の権限が必要です。詳細については、『REST Application Developer's Guide』の「Using Dynamically Defined Query Options」を参照してください。

「クエリ結果の詳細設定」（189 ページ）で説明しているように、検索結果をカスタマイズするには、`categories`、`pageStart`、`pageLength`、および `view` プロパティを使用します。

検索に適用する、以前に存続していた一連のクエリオプションに名前を付けるには、`optionsName` プロパティを使用します。`CombinedQueryDefinition` に、複合クエリ内のオプションと永続クエリオプション名の両方が含まれる場合、2 つのオプションセットが結合されます。同等のオプションが両方で行われた場合、複合クエリの設定が優先されます。

注： クエリオプションを存続させるために Node.js クライアント API は使用できません。代わりに REST または Java クライアント API を使用してください。詳細については、『REST Application Developer's Guide』の「Configuring Query Options」または『Java Application Developer's Guide』の「Query Options」を参照してください。

次の例は、CombinedQueryDefinition を使用して、データベースディレクトリ「/contributors」内の「java」と「marklogic」を含んだドキュメントを検索します。複合クエリは、return-query オプションを true に設定して、最後のクエリ構造を結果に含めます。マッチしたドキュメントの代わりに検索結果サマリが返されるように、categories プロパティは「none」に設定されます。サマリには最後のクエリが含まれます。pageLength 設定により、結果は一度に 3 つ返されます。

```
db.documents.query({
  search: {
    qtext: 'java',
    query: {
      'directory-query' : { uri: '/contributors/' },
      'term-query': { text: ['marklogic'] }
    },
    options: {
      'return-query': true
    }
  },
  categories: [ 'none' ],
  pageLength: 3
})
```

4.8 値メタデータフィールドの検索

検索対象のキーでメタデータフィールドを定義すると、値メタデータ（キー / 値メタデータとも呼ばれます）だけを検索できます。メタデータキーに関してフィールドを定義したら、通常のフィールド検索機能を使用して、検索にメタデータフィールドを含めます。例えば、queryBuilder.field および queryBuilder.word を使用して、メタデータフィールドでワードクエリを作成できます。

詳細については、『Administrator's Guide』の「Metadata Fields」を参照してください。

4.9 レキシコンおよびレンジインデックスのクエリ

Node.js クライアント API を使用すると、次の方法でレキシコンとレンジインデックスを検索および分析できます。

- 単一のレキシコンまたはレンジインデックス内の値をクエリする。
- 複数のレンジインデックスで値の共起を検索する。
- ビルトインまたはユーザー定義の集計関数を使用して、レンジインデックスやレキシコンの値または値共起を分析する。詳細については、「集計関数を使用したレキシコンとレンジインデックスの分析」（181 ページ）を参照してください。

このセクションでは、次の関連トピックについて取り上げます。

- レキシコンまたはレンジインデックス内の値のクエリ
- レキシコンでの値共起の検索
- インデックスリファレンスの作成
- 値または共起クエリの結果の詳細設定
- 集計関数を使用したレキシコンとレンジインデックスの分析

関連する検索の概念については、『Search Developer's Guide』の「Browsing With Lexicons」、および『Administrator's Guide』の「Text Indexes」を参照してください。

4.9.1 レキシコンまたはレンジインデックス内の値のクエリ

`marklogic.valueBuilder` インターフェイスを使用してレキシコンおよびレンジインデックスに対するクエリを作成し、`DatabaseClient.values.read` を使用してクエリを適用します。

例えば、「reputation」JSON プロパティまたは XML 要素のレンジインデックスを含めるようにデータベースが設定されている場合、次のクエリはレンジインデックス内のすべての値を返すことになります。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');

var db = marklogic.createDatabaseClient(my.connInfo);
var vb = marklogic.valuesBuilder;

db.values.read(
  vb.fromIndexes('reputation')
).result(function (result) {
  console.log(JSON.stringify(result, null, 2));
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

スクリプトをファイルに保存して、「サンプルデータのロード」（205 ページ）のデータに対して実行すると、次のような結果が出力されます。クエリは各固有値の `values-response.tuple` 項目を返します。

```
{ "values-response": {
    "name": "structuredef",
    "types": {
      "type": [ "xs:int" ]
    },
    "tuple": [
      {
```

```
            "frequency": 1,
            "distinct-value": [ "1" ]
          },
          {
            "frequency": 1,
            "distinct-value": [ "91" ]
          },
          {
            "frequency": 1,
            "distinct-value": [ "272" ]
          },
          {
            "frequency": 1,
            "distinct-value": [ "446" ]
          }
        ],
        "metrics": {
          "values-resolution-time": "PT0.000146S",
          "total-time": "PT0.000822S"
        }
      }
    }
```

values.slice を使用すると、値のサブセットを取得できます。例えば、クエリが次のようになるように上記のスクリプトを変更すると、クエリは 3 番目の値から 2 つ分の値を返すようになります。

```
db.values.read(
  vb.fromIndexes('reputation')
    .slice(2,4)
)

==>
{ "values-response": {
    "name": "structuredef",
    "types": {
      "type": [ "xs:int" ]
    },
    "tuple": [
      {
        "frequency": 1,
        "distinct-value": [ "272" ]
      },
      {
        "frequency": 1,
        "distinct-value": [ "446" ]
```

```
        }
      ],
      "metrics": {
        "values-resolution-time": "PT0.000174S",
        "total-time": "PT0.000867S"
      }
    }
}
```

4.9.2 レキシコンでの値共起の検索

共起は、同じドキュメントフラグメントに出現する一連のインデックスまたはレキシコンの値です。Node.js クライアント API は、*n*-way の共起、つまり、同じフラグメントで出現する複数のレキシコンまたはインデックスの値のタプルのクエリをサポートしています。

複数のレンジインデックスまたはレキシコンで値の共起を見つけるには、`marklogic.valueBuilder` インターフェイスを使用してクエリを作成し、`DatabaseClient.values.read` を使用してそれを適用します。値クエリに複数のインデックスリファレンスが含まれている場合、結果は共起タプルになります。

例えば、データベース設定に「tags」の要素レンジインデックスと「id」の要素レンジインデックスが含まれている場合、次のスクリプトは「tags」および「id」JSON プロパティまたは XML 要素で値の共起を検索します（前述したように、JSON プロパティのレンジインデックスは要素レンジインデックスインターフェイスを使用します。詳細については、「インデックス付け」（136 ページ）を参照してください）。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');

var db = marklogic.createDatabaseClient(my.connInfo);
var vb = marklogic.valuesBuilder;

db.values.read(
  vb.fromIndexes('tags','id')
).result(function (result) {
  console.log(JSON.stringify(result, null, 2));
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

スクリプトをファイルに保存して実行すると、次のような結果が出力されます。このクエリは、各共起の `values-response.tuple` 項目を返します。プロパティ `values-response.types` は、各タプル内の値のデータ型を解釈するのに役立ちます。

```
{
  "values-response": {
    "name": "structuredef",
    "types": {
      "type": [
        "xs:string",
        "xs:string"
      ]
    },
    "tuple": [
      {
        "frequency": 1,
        "distinct-value": [
          "dbobject",
          "soq7684223"
        ]
      },
      {
        "frequency": 1,
        "distinct-value": [
          "dbobject",
          "sou69803"
        ]
      },...
    ],
    "metrics": {
      "values-resolution-time": "PT0.000472S",
      "total-time": "PT0.001251S"
    }
  }
}
```

values.slice を使用すると、値のサブセットを取得できます。例えば、クエリが次のようになるようにスクリプトを変更すると、クエリは 3 番目の値から順に 2 つのタプルを返すようになります。

```
db.values.read(
  vb.fromIndexes('tags','id').slice(2,4)
)

==>
{
  "values-response": {
    "name": "structuredef",
    "types": {
      "type": [
```

```
            "xs:string",
            "xs:string"
          ]
        },
        "tuple": [
          {
            "frequency": 1,
            "distinct-value": [
              "java",
              "soq22431350"
            ]
          },
          {
            "frequency": 1,
            "distinct-value": [
              "java",
              "soq7684223"
            ]
          }
        ],
        "metrics": {
          "values-resolution-time": "PT0.00024S",
          "total-time": "PT0.001018S"
        }
      }
    }
```

4.9.3 インデックスリファレンスの作成

値および共起クエリで使用するインデックスリファレンスを作成するには、`valuesBuilder.fromIndexes` を使用します。例えば、次のようなクエリには、「reputation」という名前の、JSON プロパティや XML 要素のインデックスに対する名前によるリファレンスが含まれています。

```
db.values.read(vb.fromIndexes('reputation'))
```

インデックスリファレンスのビルダーメソッドを使用すると、インデックスリファレンスの曖昧さを回避したり、別のタイプのインデックスを使用したり、コレーションを指定したりできます。次の説明は、`valuesBuilder.fromIndexes` への入力に適用されます。

- JSON プロパティのレンジインデックスは、シンプルな名前で識別されます。例えば、`vb.fromIndexes('reputation')` は、JSON プロパティ `reputation` のレンジインデックスを示します。

- レンジインデックスは、インデックスリファレンスで識別されます。例えば、`vb.fromIndexes(vb.field('questionId'))` は、フィールドレンジインデックスを示します。
- レンジインデックスのデータ型を明示的に指定していない場合、API はインデックス解決時にサーバーサイドでデータ型を調べます。データ型を明示的に指定するには、`valuesBuilder.datatype` を使用します。

例えば、次のすべてのインデックスリファレンスは、reputation という名前のプロパティに対応した JSON プロパティのレンジインデックスを示すものです。

```
vb.fromIndexes('reputation')

vb.fromIndexes(vb.range('reputation'))

vb.fromIndexes(vb.range(vb.property('reputation')))

vb.fromIndexes(vb.range(
  vb.property('reputation'), vb.datatype('int')))
```

次の表は、`valuesBuilder` が公開するインデックス定義ビルダーメソッドとその説明をまとめたものです。

レキシコンまたは インデックスタイプ	valuesBuilder ビルダーメソッド
uri	`vb.uri`
collection	`vb.collection`（引数なし）
range	*`name`* `vb.range`
field	`vb.field`

geospatial	`vb.geoAttributePair`
	`vb.geoElement`
	`vb.geoElementPair`
	`vb.geoPath`
	`vb.geoProperty`
	`vb.geoPropertyPair`

URI とコレクションレキシコンを使用するには、データベースで有効になっている必要があります。詳細については、『Administrator's Guide』の「Text Indexes」を参照してください。これらのレキシコンを指定するには、valuesBuilder.uri および valuesBuilder.collection（引数なし）を使用します。次に例を示します。

```
db.values.read(
  vb.fromIndexes(
    vb.uri(),            // the URI lexicon
    vb.collection())     // the collection lexicon
```

4.9.4 値または共起クエリの結果の詳細設定

クエリの結果は、次の方法で詳細に設定できます。

- 結果のサブセットを選択したり、結果の変換を指定したりするには、valuesBuilder.slice を使用します。
- 値クエリオプションを指定したり、特定のフォレストに結果を限定したりするには、valuesBuilder.BuiltQuery.withOptions を使用します。オプションのリストについては、cts.values（JavaScript）または cts:values（XQuery）の API ドキュメントを参照してください。
- 結果を、別のクエリにマッチするものに制限するには、valuesBuilder.BuiltQuery.where を使用します。

これらの詳細設定は、単独または任意の組み合わせで使用できます。

例えば、次のクエリは JSON プロパティ「reputation」のレンジインデックスから値を返します。この where 節では、コレクション「myInterestingCollection」内のドキュメントの値のみを選択しています。また、slice 節では 3 番目の値から 2 つ分の結果を選択し、withOptions 節では結果を降順で返すことを指定しています。

```
db.values.read(
  vb.fromIndexes('reputation').
  where(vb.collection('myInterestingCollection')).
  slice(2,4).
  withOptions({values: ['descending']})
```

4.9.5 集計関数を使用したレキシコンとレンジインデックスの分析

valuesBuilder.BuiltQuery.aggregates でビルトインまたはユーザー定義の集計関数を使用すると、レンジインデックスおよびレキシコンに対して集計値を計算できます。このセクションでは、次の内容を取り上げます。

- 集計関数の概要
- ビルトイン集計関数の使用
- ユーザー定義の集計関数の使用

4.9.5.1 集計関数の概要

集計関数は、レキシコンおよびレンジインデックスの値または値共起に対して演算を実行します。例えば、集計関数を使用して、レンジインデックス内の値の合計を計算できます。

1 つあるいは複数のビルトインまたはユーザー定義の集計関数を値または共起クエリに適用するには、`valuesBuilder.BuiltQuery.aggregates` を使用します。ビルトイン集計関数とユーザー定義の集計関数を 1 つのクエリに組み合わせることができます。

MarkLogic サーバーには、一般的な分析関数用の複数のビルトイン集計関数が用意されています。関数のリストについては、Node.js API リファレンスを参照してください。各ビルトイン集計関数の詳細については、『Search Developer's Guide』の「Using Builtin Aggregate Functions」を参照してください。

また、集計ユーザー定義関数（UDF）を C++ で実装して、それらをネイティブプラグインとして導入することもできます。集計 UDF を使用するには、使用する UDF を事前にインストールしておく必要があります。詳細については、『Application Developer's Guide』の「Implementing an Aggregate User-Defined Function」を参照してください。UDF を実装するネイティブプラグインは、『Application Developer's Guide』の「Using Native Plugins」の指示に従ってインストールする必要があります。

> **注：** 追加のパラメータを必要とする集計 UDF は、Node.js クライアント API を使用して適用することはできません。

4.9.5.2 ビルトイン集計関数の使用

ビルトイン集計関数を使用するには、その関数の名前を `valuesBuilder.BuiltQuery.aggregates` に渡します。サポートされているビルトイン集計関数名のリストについては、Node.js API リファレンスを参照してください。

例えば、次のスクリプトはビルトイン集計関数を使用し、`reputation` という名前の JSON プロパティについて、レンジインデックス内の値の最小、最大、標準偏差を計算します。`slice(0,0)` の形式の slice 節を使用して、集計値と値ではなく、計算した集計値のみを返します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');

var db = marklogic.createDatabaseClient(my.connInfo);
var vb = marklogic.valuesBuilder;

db.values.read(
  vb.fromIndexes('reputation')
```

```
    .aggregates('min', 'max', 'stddev')
    .slice(0,0)
).result(function (result) {
  console.log(JSON.stringify(result, null, 2));
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

スクリプトを実行すると、次のような出力が生成されます。

```
{ "values-response": {
  "name": "structuredef",
  "aggregate-result": [
    { "name": "min", "_value": "1" },
    { "name": "max", "_value": "446" },
    { "name": "stddev", "_value": "197.616632228498" }
  ],
  "metrics": {
    "aggregate-resolution-time": "PT0.000571S",
    "total-time": "PT0.001279S"
  }
} }
```

4.9.5.3 ユーザー定義の集計関数の使用

『Search Developer's Guide』の「Using Aggregate User-Defined Functions」で説明しているように、集計 UDF は、関数名とその集計関数を実装するプラグインの相対パスで指定します。UDF プラグインをクエリで使用するには、そのプラグインを MarkLogic サーバーにインストールしておく必要があります。集計 UDF の作成およびインストールの詳細については、『Application Developer's Guide』の「Aggregate User-Defined Functions」を参照してください。

プラグインをインストールしたら、`valuesBuilder.udf` を使用して UDF へのリファレンスを作成し、そのリファレンスを `valuesBuilder.builtQuery.aggregates` に渡します。例えば、次のスクリプトは、「native/sampleplugin」の拡張機能データベースにインストールしたプラグインによって提供される「count」という名前のネイティブ UDF を使用します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');

var db = marklogic.createDatabaseClient(my.connInfo);
var vb = marklogic.valuesBuilder;

//console.log(vb.fromIndexes(vb.range(vb.pathIndex('/id'))
```

```
));
db.values.read(
  vb.fromIndexes('reputation')
    .aggregates(vb.udf('native/sampleplugin', 'count')
    .slice(0,0)
).result(function (result) {
  console.log(JSON.stringify(result, null, 2));
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

4.10 検索ファセットの生成

『Search Developer's Guide』の「Constrained Searches and Faceted Navigation」で説明しているように、Node.js クライアント API を使用すると、クエリの結果にファセットを含めることができます。`queryBuilder.facet` を使用してファセットを定義し、`queryBuilder.calculate` を使用して検索に含めます。ファセットは、JSON プロパティ、XML 要素および属性、フィールドおよびパスに対して作成できます。ファセットには、レンジインデックスが利用されている必要があります。

このセクションでは、次の内容を取り上げます。

- 単純なファセットの定義
- ファセット名の指定
- ファセットオプションを含める
- バケットレンジの定義
- カスタム制約ファセットの作成と使用

詳細については、『Search Developer's Guide』の「Constrained Searches and Faceted Navigation」を参照してください。

4.10.1 単純なファセットの定義

次の例は、データベースディレクトリ「/contributors/」にあるドキュメントの reputation という JSON プロパティのファセットを示しています。`withOptions` 節によって、結果には、ファセットおよびマッチしたドキュメントではなく、ファセットのみが含まれます。詳細については、「検索結果からドキュメントディスクリプタまたは値を除外する」（191 ページ）を参照してください。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');

var db = marklogic.createDatabaseClient(my.connInfo);
var qb = marklogic.queryBuilder;
```

```
db.documents.query(
  qb.where(qb.directory('/contributors/'))
    .calculate(qb.facet('reputation'))
    .withOptions({categories: 'none'})
).result( function(results) {
  console.log(JSON.stringify(results, null, 2));
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

データベースに「reputation」に対するレンジインデックスが含まれている場合、スクリプトを「サンプルデータのロード」（205 ページ）のサンプルデータに対して実行すると、次のような結果が出力されます。

```
{ "snippet-format": "empty-snippet",
  "total": 4,
  "start": 1,
  "page-length": 0,
  "results": [],
  "facets": {
    "reputation": {
      "type": "xs:int",
      "facetValues": [
        { "name": "1",
          "count": 1,
          "value": 1 },
        { "name": "91",
          "count": 1,
          "value": 91 },
        { "name": "272",
          "count": 1,
          "value": 272 },
        { "name": "446",
          "count": 1,
          "value": 446 }
      ]
    }
  }
}
```

ファセットではデフォルトで、XML 要素や JSON プロパティなど、そのファセットの派生元のエンティティと同じ名前を使用しますが、カスタム名を付けることができます。詳細については、「ファセット名の指定」（186 ページ）を参照してください。

結果の `facets` プロパティには、「reputation」ファセットに対応する一連の値バケット（`reputation` の固有値ごとに 1 つのバケット）が含まれます。各バケットには、名前（デフォルトでは値から自動生成）、その値とのマッチ数、および実際の値が含まれます。

```
"facets": {
    "reputation": {        <-- name of the facet
      "type": "xs:int",
      "facetValues": [
        { "name": "1",     <-- bucket name
          "count": 1,      <-- number of matches with this
value
          "value": 1 }     <-- value associated with this
bucket
```

4.10.2 ファセット名の指定

デフォルトで、ファセットの名前は、そのファセットが基づいているインデックス付き要素またはプロパティ名から派生されます。例えば、「reputation」プロパティに対する次のファセットは、「reputation」のプロパティ名でファセットを生成します。

```
qb.facet('reputation')
==> "facets": { "reputation": {...}}
```

`queryBuilder.facet` に、最初の引数として自身の名前を渡すことで、この動作を無効にできます。例えば、「reputation」プロパティに対する次のファセットは、「rep」のプロパティ名でファセットを生成します。

```
qb.facet('rep', 'reputation')
==> "facets": { "rep": {...} }
```

4.10.3 ファセットオプションを含める

`queryBuilder.facetOptions` を使用すると、ソート順序や返す値の最大数などの属性に影響を与えるオプションをファセット定義に含めることができます。詳細については、『Search Developer's Guide』の「Facet Options」を参照してください。また、`cts.values`（JavaScript）または `cts:values`（XQuery）などのファセットインデックスタイプに対応するクエリの詳細な API ドキュメントを参照してください。

例えば次のファセット定義は、バケットを降順で並べ、バケット数を 2 つに制限するように要求します。したがって、[1, 91, 272, 446] という順序のバケットではなく、結果は [446, 272, 91, 1] の順序になります。また、最初の 2 つのバケット以降は削除されます。

```
qb.facet('rep','reputation',
qb.facetOptions('limit=2','descending')))

==>

"facets": {
  "reputation": {
    "type": "xs:int",
    "facetValues": [
      { "name": "446",
        "count": 1,
        "value": 446 },
      { "name": "272",
        "count": 1,
        "value": 272 }
    ]
  }
}
```

4.10.4 バケットレンジの定義

デフォルトでは、ファセットは個々の値ごとにバケット化されます。ただし、`queryBuilder.bucket` を使用して、数値および日付値に対する独自のバケットを定義できます。バケットには値の範囲を指定できます。バケットの値の範囲の上限と下限は、バケットアンカーと呼ばれます。両方のアンカー値を含めることも、上限アンカーまたは下限アンカーを省略することもできます。

`dateTime` 値のバケットでは、「now」および「start-of-day」などのシンボリックなアンカーを使用できます。実際の値は、クエリの評価時に計算されます。そのような定義は、計算されたバケットを表します。サポートされている値のリストについては、『Search Developer's Guide』の「computed-bucket」を参照してください。

例えば、次のようなファセット定義を使用して、reputation 値を「less than 50」、「50 to 100」、および「greater than 100」のバケットに分類できます。

```
qb.facet('reputation',
         qb.bucket('less than 50', '<', 50),
         qb.bucket('50 to 100', 50, '<', 101),
         qb.bucket('greater than 100', 101, '<'))
==>
```

```
"facets": {
  "reputation": {
    "type": "bucketed",
    "facetValues": [
      { "name": "less than 50",
        "count": 1,
        "value": "less than 50"
      },
      { "name": "50 to 100",
        "count": 1,
        "value": "50 to 100"
      },
      { "name": "greater than 100",
        "count": 2,
        "value": "greater than 100"
      }
    ]
  }
}
```

上記の例で、「<」は上限アンカー値と下限アンカー値の間の境界として機能する定数です。関係演算子ではありません。区切り文字を使用することで、API では下限のないバケット、上限のないバケット、上限と下限の両方があるバケットを処理できます。

バケット定義のその他の例については、『Search Developer's Guide』の「Buckets Example」、および『Search Developer's Guide』の「Computed Buckets Example」を参照してください。

4.10.5 カスタム制約ファセットの作成と使用

『Search Developer's Guide』の「Creating a Custom Constraint」で説明しているように、カスタム制約を定義するときに、制約用のファセットジェネレータも定義できます。カスタム制約ファセットジェネレータを使用するには、次の手順に従います。

1. start-facet 関数および finish-facet 関数を含む XQuery モジュールを実装します。詳細については、『Search Developer's Guide』の「Creating a Custom Constraint」を参照してください。

2. 「制約パーサーのインストール」（150 ページ）で説明しているように、`DatabaseClient.config.query.custom` インターフェイスを使用して、REST API インスタンスに関連付けられている modules データベースにカスタム制約モジュールをインストールします。

3. `queryBilder.CalculateFunction` を使用して、ファセット定義を作成するときにファセットジェネレータへのリファレンスを作成します。

例えば、「制約パーサーのインストール」（150 ページ）で示しているように、カスタム制約モジュールが `ss-cat.xqy` としてインストールされている場合は、次のように記述します。

```
db.config.query.write('ss-cat.xqy', ...)
```

これにより、ファセットジェネレータを次のようにファセット定義で使用できるようになります。

```
qb.facet('categories', qb.calculateFunction('ss-cat.xqy'))
```

4.11 クエリ結果の詳細設定

このセクションでは、Node.js クライアント API の次の機能について説明します。これらの機能により、`queryBuilder.slice` または `valuesBuilder.BuiltQuery.slice` を使用して検索結果をカスタマイズできるようになります。

- 使用可能な詳細設定
- クエリの結果のページネーション
- メタデータを返す
- 検索結果からドキュメントディスクリプタまたは値を除外する
- 検索スニペットの生成
- 検索結果の変換
- マッチする各ドキュメントの部分的な抽出

4.11.1 使用可能な詳細設定

デフォルトでは、`DatabaseClient.documents.query` を使用して検索を実行すると、マッチしたドキュメントディスクリプタが 1「ページ」分、関連度順に並べられて返されます。各ディスクリプタには、マッチするドキュメントのコンテンツが含まれています。

Node.js クライアント API では、この動作を変更可能な、複数の結果の詳細設定用 `queryBuilder` メソッドを提供しています。

- `queryBuilder.slice` を使用して「ページ」のサイズや開始ドキュメントを変更する。詳細については、「クエリの結果のページネーション」（190 ページ）を参照してください。
- `queryBuilder.orderBy` を使用して結果の順序を変更する。詳細については、『Node.js Client API Reference』を参照してください。

- queryBuilder.withOptions を使用して、コンテンツに加えて、またはコンテンツの代わりにメタデータを要求する。詳細については、「メタデータを返す」（191 ページ）を参照してください。
- queryBuilder.withOptions を使用して、応答からドキュメントディスクリプタを除外する。これは、スニペット、ファセット、メトリック、またはマッチに関するその他のデータだけを取得する場合に便利です。詳細については、「検索結果からドキュメントディスクリプタまたは値を除外する」（191 ページ）を参照してください。
- queryBuilder.snippet と queryBuilder.slice を使用して、マッチするドキュメントに加えて、またはマッチするドキュメントの代わりに検索マッチのスニペットを要求する。スニペットはカスタマイズできます。詳細については、「検索スニペットの生成」（192 ページ）を参照してください。
- queryBuilder.calculate を使用して、マッチするドキュメントに加えて、またはマッチするドキュメントの代わりに検索ファセットを要求する。ファセットバケットはカスタマイズできます。詳細については、「検索ファセットの生成」（184 ページ）を参照してください。
- マッチしたドキュメントまたは検索結果のサマリに読み取り変換を適用する。詳細については、「検索結果の変換」（193 ページ）を参照してください。

スライスでは、結果セットに含めるマッチするドキュメントの範囲を指定します。queryBuilder.slice を明示的に呼び出さないと、デフォルトのスライスが定義されたままになります。クエリに対してマッチするものがないため、または slice(0,0) などの空のページ範囲を定義したためにスライスが空の場合、他の詳細設定メソッド（calculate、orderBy、snippet、withOptions）は何も影響しません。

これらの機能は組み合わせて使用できます。例えば、ドキュメントディスクリプタがある場合でもない場合でも、スニペットとファセットを一緒に要求できます。

4.11.2 クエリの結果のページネーション

結果のスライス（バッチ）を取得するには、queryBuilder.slice および valuesBuilder.BuiltQuery.slice を使用します。結果のスライスは、Array.prototype.slice を使用した場合と同様に、ゼロベースの開始位置と終了位置で定義されます（終了位置の値はスライスには含まれません）。

例えば次のクエリは、最初の結果から 5 つ分の結果を返します。

```
qb.where(qb.parsedFrom('oslo')).slice(0,5)

vb.fromIndexes('reputation').slice(0,5)
```

To return the next 5 results, you would use queries such as the following

```
qb.where(qb.parsedFrom('oslo')).slice(5,10)

vb.fromIndexes('reputation').slice(5,10)
```

結果の最大数のデフォルト値は 10 です。

開始位置と終了位置をゼロに設定するとマッチ（または値）なしになりますが、検索の推定総マッチ数や値クエリの計算された集計などを含む結果の省略サマリが返されます。

4.11.3 メタデータを返す

デフォルトでは、クエリはマッチする各ドキュメントのドキュメントディスクリプタを返し、ディスクリプタにはドキュメントコンテンツが含まれています。コンテンツの代わりにメタデータを返すには、`queryBuilder.withOptions` の `categories` プロパティに `'metadata'` を設定します。次に例を示します。

```
db.documents.query(
  qb.where(qb.parsedFrom('oslo'))
    .withOptions({categories: 'metadata'})
)
```

メタデータとドキュメントの両方を返すには、categories に `'content'` と `'metadata'` の両方を設定します。次に例を示します。

```
db.documents.query(
  qb.where(qb.parsedFrom('oslo'))
    .withOptions({categories: ['content', 'metadata']})
)
```

4.11.4 検索結果からドキュメントディスクリプタまたは値を除外する

デフォルトでは、クエリはマッチする各ドキュメントのドキュメントディスクリプタを返し、ディスクリプタにはドキュメントコンテンツが含まれています。マッチするドキュメントなしでスニペット、ファセット、またはその他の検索結果データを取得する場合は、`queryBuilder.withOptions` の `categories` プロパティを `'none'` に設定します。

例えば次のクエリは、通常、2 つのドキュメントディスクリプタのコンテンツを返します。

```
db.documents.query(
  qb.where(qb.parsedFrom('oslo'))
)
```

次の withOptions 節を追加すると、ドキュメントディスクリプタを受け取る代わりに、検索スニペットを含む検索結果のサマリを受け取ります。

```
db.documents.query(
  qb.where(qb.parsedFrom('oslo'))
    .withOptions({categories: 'none'})
)
```

検索結果のサマリのコンテンツは、クエリに適用するその他の詳細設定によって異なりますが、ドキュメントディスクリプタが含まれることはありません。

4.11.5 検索スニペットの生成

検索結果ページでは一般的に、マッチするドキュメントの一部が表示され、さらにマッチ部分が強調表示されます。場合によっては、検索マッチのコンテキストを説明するテキストも表示されます。このような検索結果の断片は、スニペットと呼ばれます。

デフォルトではスニペットはクエリ結果に含まれません。スニペットを要求するには、queryBuilder.snippet を使用してスライス定義に snippet 節を含めます。例えば、次のクエリは、デフォルトの形式でスニペットを返します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');

var db = marklogic.createDatabaseClient(my.connInfo);
var qb = marklogic.queryBuilder;

db.documents.query(
    qb.where(
      qb.byExample({aboutMe: {$word: 'marklogic'}})
    ).slice(qb.snippet())
).result( function(results) {
  console.log(JSON.stringify(results, null, 2));
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

また、開始位置と終了位置を指定する slice に snippet 節を含めることもできます。次に例を示します。

```
slice(0, 5, qb.snippet())
```

マッチするドキュメントなしでスニペットを取得するには、withOptions({categories: 'none'}) 節を追加します。次に例を示します。

```
...slice(qb.snippet()).withOptions({categories: 'none'})
```

queryBuilder.snippet に名前を指定することで、いずれかのビルトインスニペットジェネレータまたは独自のカスタムスニペットジェネレータを使用できます。例えば、次のスライス定義は、ビルトインの metadata-snippet ジェネレータによって生成されたスニペットを要求します。

```
slice(0, 5, qb.snippet('metadata')
```

一部のビルトインスニペッタは追加のオプションを受け付けます。これは、queryBuilder.snippet の 2 番目のパラメータで指定できます。例えば、次のスニペット定義は、スニペットテキストのサイズを 25 文字に制限します。

```
qb.snippet('my-snippeter.xqy', {'max-snippet-chars': 25})
```

サポートされているオプションの詳細については、Node.js API リファレンスおよび『Search Developer's Guide』の「Specifying transform-results Options」を参照してください。

カスタムスニペッタを使用するには、次の手順に従います。

1. XQuery にスニペットジェネレータを実装します。snippet 関数は、『Search Developer's Guide』の「Specifying Your Own Code in transform-results」で指定されているインターフェイスに準拠している必要があります。

2. DatabaseClient.config.query.snippet.write を使用して、スニペットモジュールを REST API インスタンスの modules データベースにインストールします。

3. カスタムスニペットモジュールの名前を、queryBuilder.snippet に提供するスニペッタ名として使用します。次に例を示します。

```
slice(0, 5, qb.snippet('my-snippeter.xqy'))
```

カスタムスニペッタにオプションやパラメータを渡すことはできません。

スニペットの生成の詳細については、『Search Developer's Guide』の「Modifying Your Snippet Results」を参照してください。

4.11.6 検索結果の変換

変換関数を適用すると、検索または値クエリからの応答に対して任意の変更を加えることができます。変換は、クエリによって返される各ドキュメントと、検索または値の応答サマリ（存在する場合）に適用されます。

変換関数を使用するには、事前に MarkLogic サーバーにインストールしておく必要があります。変換関数をインストールおよび管理するには、`DatabaseClient.config.transforms` を使用します。

クエリで変換を使用するには、`queryBuilder.transform` または `valuesBuilder.transform` で変換ディスクリプタを作成します。インストール済みの変換関数の名前を指定する必要があります。実装固有のパラメータを含めることもできます。詳細および例については、「コンテンツ変換機能を扱う」（245 ページ）を参照してください。

例えば、次のクエリは、「js-query-transform」という名前の変換を検索結果に適用します。ドキュメントは返されないため（`withOptions`）、クエリは検索結果のサマリのみを返し、変換はサマリにのみ適用されます。クエリがドキュメントを返した場合、変換はマッチした各ドキュメントにも適用されます。

```
db.documents.query(
  qb.where(
    qb.byExample({writeTimestamp: {'$exists': {}}})
  ).slice(qb.transform('js-query-transform'))
   .withOptions({categories: 'none'})
)
```

同じ方法で、値クエリに変換を適用することもできます。次に例を示します。

```
db.values.read(
  vb.fromIndexes('reputation')
    .slice(0, 5, vb.transform('js-query-transform'))
```

詳細については、「コンテンツ変換機能を扱う」（245 ページ）を参照してください。

4.11.7 マッチする各ドキュメントの部分的な抽出

ドキュメント全体ではなく、マッチする各ドキュメント内のコンテンツのサブセットを返すには、`queryBuilder.extract` を使用します。選択したプロパティ、選択したプロパティとその祖先、または選択したプロパティ以外のすべての要素を返すことができます。デフォルトでは、選択したプロパティのみが含まれています。

プロパティは、XPath 式を使用して指定します。指定できるのは、パスのレンジインデックスを作成する目的で使用可能な XPath のサブセットに制限されます。詳細については、『Administrator's Guide』の「Limitations on Index Path Expressions」を参照してください。

次の例は、「queryBuilder を使用した QBE の作成」（154 ページ）の最初のクエリと同じ検索を実行しますが、queryBuilder.slice と queryBuilder.extract を使用して結果を絞り込み、マッチするドキュメントから displayName プロパティと location プロパティのみを返します。「サンプルデータのロード」（205 ページ）で作成したドキュメントに対して検索を実行すると、2 つのドキュメントがマッチします。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');

var db = marklogic.createDatabaseClient(my.connInfo);
var qb = marklogic.queryBuilder;

db.documents.query(
    qb.where(qb.byExample( {location: 'Oslo, Norway'} ))
      .slice(qb.extract(
        ['/Contributor/displayName',
'/Contributor/location'])
      )
).result( function(matches) {
  matches.forEach(function(match) {
    console.log(match.content);
  });
});
```

上記の方法で queryBuilder.extract を使用すると、マッチする各ドキュメントは次の形式のコンテンツを含むドキュメントディスクリプタを生成します。

```
{ context: original-document-context,
  extracted: [ obj-from-path1, obj-from-path2, ...]}
```

例えば、上記のクエリは次の出力を生成します。

```
{ context: 'fn:doc("/contributors/contrib1.json")',
  extracted: [
    { displayName: 'Lars Fosdal' },
    { location: 'Oslo, Norway' } ]
}
{ context: 'fn:doc("/contributors/contrib2.json")',
  extracted: [
    { displayName: 'petrumo' },
    { location: 'Oslo, Norway' } ]
}
```

また、selected 値を queryBuilder.extract に渡すことで、元のマッチするドキュメントの疎な表現を生成できます。選択したプロパティと祖先を含む疎なドキュメント、または選択したプロパティが除外されたドキュメント全体を作成できます。

例えば、次のクエリは同じプロパティを返しますが、それらの祖先が含まれています。

```
db.documents.query(
    qb.where(qb.byExample( {location: 'Oslo, Norway'} ))
      .slice(qb.extract({
        paths: ['/Contributor/displayName',
'/Contributor/location'],
        selected: 'include-with-ancestors'
      }))
)
```

このクエリの出力は、次のような元のドキュメントの疎なバージョンになります。

```
{ Contributor: {
  displayName: 'Lars Fosdal',
  location: 'Oslo, Norway' } }
{ Contributor: {
  displayName: 'petrumo',
  location: 'Oslo, Norway' } }
```

次の表は、queryBuilder.extract の selected パラメータでサポートされている各値が、返されるコンテンツにどのような影響を与えるのかを示したものです。

selected 値	出力

include
(デフォルト)
	{ context:
	'fn:doc("/contributors/contrib1.json")',
	  extracted: [
	  { displayName: 'Lars Fosdal' },
	  { location: 'Oslo, Norway' }] }
	{ context:
	'fn:doc("/contributors/contrib2.json")',
	  extracted: [
	  { displayName: 'petrumo' },
	  { location: 'Oslo, Norway' }] }

selected 値	出力
include-with-ancestors	{ Contributor: { displayName: 'Lars Fosdal', location: 'Oslo, Norway' } } { Contributor: { displayName: 'petrumo', location: 'Oslo, Norway' } }
exclude	{ Contributor: { userName: 'souser10002@email.com', reputation: 446, originalId: '10002', aboutMe: 'Software Developer since 1987, ...', id: 'sou10002' } } { Contributor: { userName: 'souser1000634@email.com', reputation: 272, originalId: '1000634', aboutMe: 'Developer at AspiroTV', id: 'sou1000634' } }
all	{ Contributor: { userName: 'souser10002@email.com', reputation: 446, displayName: 'Lars Fosdal', originalId: '10002', location: 'Oslo, Norway', aboutMe: 'Software Developer since 1987...', id: 'sou10002' } } { Contributor: { userName: 'souser1000634@email.com', reputation: 272, displayName: 'petrumo', originalId: '1000634', location: 'Oslo, Norway', aboutMe: 'Developer at AspiroTV', id: 'sou1000634' } }

抽出パスが、マッチしたドキュメント内のコンテンツとマッチしない場合は、対応するプロパティが省略されます。抽出パスが一切マッチしない場合、ドキュメントのディスクリプタは返されますが、`extracted` プロパティまたは疎なドキュメントではなく、`extracted-none` プロパティが格納されます。次に例を示します。

```
{ context: 'fn:doc("/contributors/contrib1.json")',
  extracted-none: null
}
```

4.12 検索語入力候補の生成

このセクションでは、Node.js クライアント API を使用して検索語入力候補を生成する方法について説明します。ここでは、次の内容を取り上げます。

- 候補インターフェイスについて
- 例：検索語候補の生成

4.12.1 候補インターフェイスについて

通常、検索アプリケーションでは、検索ボックスへの入力時に検索語の候補が表示されます。このような候補は、データベース内のタームに基づいており、一般的に、ユーザーインターフェイスの対話性を高め、アプリケーションに適した検索語をすばやく提案するために使用されます。

候補は、リクエストで指定するレンジインデックスおよびレキシコンから取り出されます。パフォーマンス上の理由から、ワードレキシコンよりもレンジまたはコレクションインデックスのほうが推奨されます。詳細については、`search:suggest` の使用上の注意を参照してください。候補は、追加の検索基準でさらにフィルタリングできます。

Node.js クライアント API を使用して検索語入力候補を生成するには、`DatabaseClient.documents.suggest` を使用します。最もシンプルな候補リクエストは、次の形式になります。

```
db.documents.suggest(partialText, qualifyingQuery)
```

partialText は候補の生成対象となるクエリテキストで、*qualifyingQuery* は追加の検索基準（インデックスとレキシコンのバインドなど）です。修飾クエリ（qualifying query）は恣意的に複雑にすることもできますが、通常は入力されたその一部が、その後完成した語句によって置き換えられます。

例えば、次の呼び出しは、部分的な語句「doc」の候補をリクエストします。`qb.parsedFrom` の最初のパラメータは空の文字列であるため、追加の検索基準はありません。

```
db.documents.suggest('doc',
  qb.where(qb.parsedFrom('',
    qb.parseBindings(
      qb.value('prefix', qb.bind('prefix')),
      qb.range('name', qb.bindDefault()))
  ))
)
```

修飾クエリのパースバインドには、「name」という名前の JSON プロパティに対するレンジクエリ（`qb.range('name', ...)`）への非修飾ターム（`qb.bindDefault()`）のバインドが含まれます。データベースには、マッチするレンジインデックスが含まれている必要があります。

このため、データベースに次の形式のドキュメントが含まれている場合、「doc」の候補は「name」の値からのみ取り出され、「prefix」または「alias」の値からは取り出されません。

```
{ "prefix": "xdmp",
  "name": "documentLoad",
  "alias": "document-load" }
```

ユーザーが候補から検索語を完全に入力したら、ドキュメントクエリ内の空の文字列は完全な語句に置き換えられます。このため、ユーザーが「documentLoad」というタームを完全に入力すると、同じクエリを次のように使用してマッチするドキュメントを取得できます。

```
db.documents.query(
  qb.where(qb.parsedFrom('documentLoad',
    qb.parseBindings(
      qb.value('prefix', qb.bind('prefix')),
      qb.range('name', qb.bindDefault()))
  ))
)
```

修飾クエリには、他の検索基準を含めることができます。次の例は、クエリ「prefix:xdmp」を追加します。バインドにより、「prefix」タームが「prefix」という名前の JSON プロパティに対する値クエリに関連付けられます。「prefix:xdmp」タームは、ユーザーが以前に検索ボックスに入力したテキストの一部である可能性があります。

```
db.documents.suggest('doc',
  qb.where(qb.parsedFrom('prefix:xdmp',
    qb.parseBindings(
     qb.value('prefix', qb.bind('prefix')),
      qb.range('name', qb.bindDefault()))
  ))
)
```

この場合、前回と同様に、候補は「name」プロパティから取り出されますが、「prefix:xdmp」クエリを満たすドキュメント内に出現する値に制限されます。つまり、候補は、次の両方の基準を満たすドキュメント内の値から取り出されます。

- 値が「doc」で始まる JSON プロパティ「name」が含まれる
- 「xdmp」と完全に同じ値を持つ「prefix」という名前の JSON プロパティが含まれる

また、完成させるタームには明示的なバインドも使用できます。例えば、次の呼び出しは、「aka:doc」の候補をリクエストします。「aka」は JSON プロパティ「alias」に対するレンジインデックスにバインドされています。候補は、このプロパティの値からのみ取り出されます。

```
db.documents.suggest('aka:doc',
  qb.where(qb.parsedFrom('',
    qb.parseBindings(
     qb.range('alias', qb.bind('aka')),
      qb.value('prefix', qb.bind('prefix')),
      qb.range('name', qb.bindDefault()))
  ))
)
```

この場合に返される候補には、プレフィックスが含まれます。例えば、「aka:document-load」などの候補になります。

修飾クエリには、文字列クエリおよび構造化クエリのコンポーネントを両方とも含めることができますが、通常は、候補を制約するためのインデックスまたはレキシコンバインドを少なくともう 1 つ含めることになります。例えば、次のコードは、候補をデータベースディレクトリ `/suggest/` 内のドキュメントに制限するディレクトリクエリを追加します。

```
db.documents.suggest('doc',
  qb.where(qb.parsedFrom('',
    qb.parseBindings(
      qb.value('prefix', qb.bind('prefix')),
      qb.range('name', qb.bindDefault()))
  ), qb.directory('/suggest/', true))
)
```

追加の `suggestBinding` パラメータを含めることで、修飾クエリを変更することなく候補ごとにバインドをオーバーライドできます。

以前に作成した修飾クエリを使用していて、何らかの理由（パフォーマンスなど）で候補の範囲を制限するバインドを追加したい場合は、`queryBuilder.suggestBindings` を使用してオーバーライドバインドを追加できます。

例えば、次のコードは、JSON プロパティ「alias」のレンジインデックスに対するバインドで、修飾クエリ内の無修飾タームのバインドをオーバーライドします。そのため、ドキュメントに値「documentLoad」の name プロパティと値「document-load」の alias プロパティが含まれている場合、候補には `suggestBindings` の指定がなくても「documentLoad」が含まれますが、「document-load」はオーバーライドありの場合に含まれます。

```
db.documents.suggest('doc',
  qb.where(qb.parsedFrom('',
    qb.parseBindings(
      qb.value('prefix', qb.bind('prefix')),
      qb.range('name', qb.bindDefault()))
  )),
  qb.suggestBindings(qb.range('alias', qb.bindDefault()))
)
```

オーバーライドはバインドごとに行われます。上記の例では、無修飾タームのデフォルトのバインドのみがオーバーライドされます。「prefix」のバインドは、suggestBindings に「prefix」のバインドが含まれない限り有効です。

4.12.2 例：検索語候補の生成

このセクションの例は、「候補インターフェイスについて」（198 ページ）で説明しているユースケースを示したものです。

このスクリプトは、まずドキュメント例をデータベースに読み込んで、それを基に候補を生成します。この例を実行するには、次のレンジインデックスを追加する必要があります。これは、管理画面または Admin API を使用して作成できます。詳細については、『Administrator's Guide』の「Range Indexes and Lexicons」を参照してください。

- タイプ「string」、ローカル名「name」の要素レンジインデックス。
- タイプ「string」、ローカル名「alias」の要素レンジインデックス。

この例では、「候補インターフェイスについて」（198 ページ）で詳細に説明している次のユースケースについて説明します。

- ケース 1：`name` プロパティから取り出された「doc」の候補
- ケース 2：`prefix` が xdmp の `name` から取り出された「doc」の候補
- ケース 3：`prefix` が xdmp の `name` から取り出されたドキュメント「doc」の候補。候補は `/suggest/` ディレクトリ内のドキュメントから取り出される。

- ケース 4 :「aka:doc」の候補。「aka」プレフィックスにより、候補は `alias` プロパティから取り出される。
- ケース 5 : 候補のバインドのオーバーライドによって `alias` プロパティから取り出された「doc」の候補。

次の表は、ドキュメント例のプロパティ値を参照しやすいようにまとめたものです。

URI	name	prefix	Alias
/suggest/load.json	documentLoad	xdmp	document-load
/suggest/insert.json	documentInsert	xdmp	document-insert
/suggest/query.json	documentQuery	cts	document-query
/suggest/search.json	search	cts	search
/elsewhere/delete.json	documentDelete	xdmp	document-delete

この例を実行すると、次のような結果が生成されます。

```
1: Suggestions for naked term "doc":
["documentDelete","documentInsert","documentLoad","documen
tQuery"]

2: Suggestions filtered by prefix:xdmp:
["documentDelete","documentInsert","documentLoad"]

3: Suggestions filtered by prefix:xdmp and dir /suggest/:
["documentInsert","documentLoad"]

4: Suggestions for "aka:doc":
[
  "aka:document-delete",
  "aka:document-insert",
  "aka:document-load",
  "aka:document-query"
]

5: Suggestions with overriding bindings:
["document-delete","document-insert","document-load"]
```

例を実行するには、次のスクリプトをファイルにコピーし、データベース接続情報を必要に応じて変更し、node コマンドでスクリプトを実行します。「このガイドの例の使用方法」（33 ページ）で説明しているように、このスクリプトでは、接続情報が my-connection.js という名前のファイルに記述されているものと想定しています。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);
var qb = marklogic.queryBuilder;

// NOTE: This example requires a database configuration
// that includes two element range index:
// - type string, localname name
// - type string, localname alias

// Initialize the database with the sample documents
db.documents.write([
  { uri: '/suggest/load.json',
    contentType: 'application/json',
    content: {
      prefix: 'xdmp',
      name: 'documentLoad',
      alias: 'document-load'
  } },
  { uri: '/suggest/insert.json',
    contentType: 'application/json',
    content: {
      prefix: 'xdmp',
      name: 'documentInsert',
      alias: 'document-insert'
  } },
  { uri: '/suggest/query.json',
    contentType: 'application/json',
    content: {
      prefix: 'cts',
      name: 'documentQuery',
      alias: 'document-query'
  } },
  { uri: '/suggest/search.json',
    contentType: 'application/json',
    content: {
      prefix: 'cts',
      name: 'search',
      alias: 'search'
  } },
  { uri: '/elsewhere/delete.json',
```

```
    contentType: 'application/json',
    content: {
      prefix: 'xdmp',
      name: 'documentDelete',
      alias: 'document-delete'
  } },
]).result().then(function(response) {
  // (1) Get suggestions for a naked term
  return db.documents.suggest('doc',
    qb.where(qb.parsedFrom('',
      qb.parseBindings(
        qb.range('name', qb.bindDefault()))
    ))
  ).result(null, function(error) {
    console.log(JSON.stringify(error, null, 2));
  });
}).then(function(response) {
  console.log('1: Suggestions for naked term "doc":');
  console.log(JSON.stringify(response));

  // (2) Get suggestions for a qualified term
  return db.documents.suggest('doc',
    qb.where( qb.parsedFrom('prefix:xdmp',
      qb.parseBindings(
        qb.value('prefix', qb.bind('prefix')),
        qb.range('name', qb.bindDefault()))
    ))
  ).result(null, function(error) {
    console.log(JSON.stringify(error, null, 2));
  });
}).then(function(response) {
  console.log('\n2: Suggestions filtered by prefix:xdmp:');
  console.log(JSON.stringify(response));

  // (3) Suggestions limited by directory
  return db.documents.suggest('doc',
    qb.where( qb.parsedFrom('prefix:xdmp',
      qb.parseBindings(
        qb.value('prefix', qb.bind('prefix')),
        qb.range('name', qb.bindDefault()))
      ),
      qb.directory('/suggest/', true))
  ).result(null, function(error) {
    console.log(JSON.stringify(error, null, 2));
  });
```

```
}).then(function(response) {
  console.log('\n3: Suggestions filtered by prefix:xdmp and
dir /suggest/:');
  console.log(JSON.stringify(response));

  // (4) Get suggestions for a term with a binding
  return db.documents.suggest('aka:doc',
    qb.where( qb.parsedFrom('',
      qb.parseBindings(
        qb.range('alias', qb.bind('aka')),
        qb.range('name', qb.bindDefault()))
    ))
  ).result(null, function(error) {
    console.log(JSON.stringify(error, null, 2));
  });
}).then(function(response) {
  console.log('\n4: Suggestions for "aka:doc":');
  console.log(JSON.stringify(response, null, 2));

  // (5) Get suggestions using a binding override
  return db.documents.suggest('doc',
    qb.where( qb.parsedFrom('prefix:xdmp',
      qb.parseBindings(
        qb.value('prefix', qb.bind('prefix')),
        qb.range('name', qb.bindDefault()))
    )),
    qb.suggestBindings(
        qb.range('alias', qb.bindDefault()))
  ).result(null, function(error) {
    console.log(JSON.stringify(error, null, 2));
  });
}).then(function(response) {
  console.log('\n5: Suggestions with overriding
bindings:');
  console.log(JSON.stringify(response));
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

4.13 サンプルデータのロード

この章のいくつかの例では、MarkLogic サンプルスタックのシードデータから生成されたデータを使用しています。サンプルスタックは、MarkLogic リファレンス用アプリケーションアーキテクチャのオープンソースの実装です。詳細については、『Reference Application Architecture Guide』を参照してください。

データをロードするには、次のスクリプトをファイルにコピーして実行します。このスクリプトでは、「このガイドの例の使用方法」（33 ページ）で説明している接続データを使用します。

一部の例では、レンジインデックスが必要です。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');

var documents = [
{ uri: '/contributors/contrib1.json', content:
  {"Contributor":{
    "userName":"souser10002@email.com", "reputation":446,
    "displayName":"Lars Fosdal", "originalId":"10002",
    "location":"Oslo, Norway",
    "aboutMe":"Software Developer since 1987, mainly using
Delphi.",
    "id":"sou10002"}}},
{ uri: '/contributors/contrib2.json', content:
  {"Contributor":{
   "userName":"souser1000634@email.com", "reputation":272,
    "displayName":"petrumo", "originalId":"1000634",
    "location":"Oslo, Norway",
    "aboutMe":"Developer at AspiroTV",
    "id":"sou1000634"}}},
{ uri: '/contributors/contrib3.json', content:
  {"Contributor":{
    "userName":"souser1248651@email.com", "reputation":1,
    "displayName":"Nullable", "originalId":"1248651",
    "location":"Ogden, UT",
    "aboutMe":"...My current work includes work with
MarkLogic Application Server (Using XML, Xquery, and
Xpath), WPF/C#, and Android Development (Using Java)...",
    "id":"sou1248651"}}},
{ uri: '/contributors/contrib4.json', content:
  {"Contributor":{
    "userName":"souser1601813@email.com", "reputation":91,
    "displayName":"grechaw", "originalId":"1601813",
    "location":"Occidental, CA",
    "aboutMe":"XML (XQuery, Java, XML database) software
engineer at MarkLogic.Hardcore accordion player.",
    "id":"sou1601813"}}},
{ uri: '/questions/q1.json', content:
  { "tags": [ "java", "sql", "json", "nosql", "marklogic"
],
    "owner": {
```

```
      "userName": "souser1238625@email.com",
      "displayName": "Raevik",
      "id": "sou1238625"
    },
    "id": "soq22431350",
    "accepted": false,
    "text": "I have a MarkLogic DB instance populated with
JSON documents that interest me.I have executed a basic
search and have a SearchHandle that will give me the URIs
that matched.Am I required to now parse through the
flattened JSON string looking for my key?",
    "creationDate": "2014-03-16T00:06:06.497",
    "title": "MarkLogic basic questions on equivalent of
SELECT with Java API"
 }},
{ uri: '/questions/q2.json', content:
  { "tags": [ "java", "dbobject", "mongodb" ],
    "owner": {
      "userName": "souser69803@email.com",
      "displayName": "Ankur",
      "id": "sou69803"
    },
    "id": "soq7684223",
    "accepted": true,
    "text": "MongoDB seems to return BSON/JSON objects.  I
thought that surely you'd be able to retrieve values as
Strings, ints etc. which can then be saved as POJO.  I have
a DBObject (instantiated as a BasicDBObject) as a result of
iterating over a list ...(cur.next()).  Is the only way
(other than using some sort of persistence framework) to
get the data into a POJO to use a JSON
serlialiser/deserialiser?",
    "creationDate": "2011-10-07T07:27:18.097",
   "title": "Convert DBObject to a POJO using MongoDB Java
Driver"
  }},
{ uri: '/questions/q3.json', content:
  { "tags": [ "json", "marklogic" ],
    "owner": {
      "userName": "souser1238625@email.com",
      "displayName": "Raevik",
      "id": "sou1238625"
    },
    "id": "soq22412345",
    "accepted": false,
    "text": "Does marklogic manage JSON documents?",
```

```
    "creationDate": "2014-02-10T00:13:03.282",
    "title": "JSON document management in MarkLogic"
 }},
];

var db = marklogic.createDatabaseClient(my.connInfo);

db.documents.write(documents)
  .result(null, function(error) {
      console.log(JSON.stringify(error));
    });
```

5.0 セマンティックデータを扱う

この章では、Node.js クライアント API を使用してセマンティックトリプルを読み込み、セマンティックグラフを管理し、セマンティックデータをクエリする方法について、次のトピックで説明します。

- 一般的なセマンティックタスクの概要
- トリプルのロード
- SPARQL を使用したセマンティックトリプルのクエリ
- 例：SPARQL クエリ
- グラフの管理
- SPARQL Update を使用したグラフおよびグラフデータの管理
- SPARQL クエリまたは Update への推論ルールの適用

この章で取り上げるのは、セマンティック操作に対する Node.js クライアント API の使用について特有の詳細のみです。また、『Semantics Developer's Guide』を参照してください。

5.1 一般的なセマンティックタスクの概要

次の表は、セマンティックに関連する一般的なタスクと、そのタスクを実行するのに最適なメソッドを示したものです。インターフェイスの完全なリストについては、Node.js API リファレンスを参照してください。

目的	使用するメソッド
SPARQL Update を使用せずにセマンティックトリプルを名前付きグラフまたはデフォルトグラフにロードする。	`DatabaseClient.graphs.write` 詳細については、「トリプルのロード」（210 ページ）を参照してください。
SPARQL Update を使用してグラフまたはグラフデータを管理する。	`DatabaseClient.graphs.sparlUpdate` 詳細については、「SPARQL Update を使用したグラフおよびグラフデータの管理」（222 ページ）を参照してください。

目的	使用するメソッド
データベースからセマンティックグラフを読み取る。	`DatabaseClient.graphs.read` 詳細については、「グラフのコンテンツ、メタデータ、またはパーミッションの取得」（219 ページ）を参照してください。
SPARQL を使用してセマンティックデータをクエリする。	`DatabaseClient.graphs.sparql` 詳細については、「SPARQL を使用したセマンティックトリプルのクエリ」（213 ページ）を参照してください。

5.2 トリプルのロード

このトピックでは、`DatabaseClient.graphs.write` または `DatabaseClient.graphs.writeStream` を使用して、いくつかの形式でセマンティックトリプルをデータベースに読み込む方法を取り上げます。サポートされている形式の一覧については、『Semantics Developer's Guide』の「Supported RDF Triple Formats」を参照してください。SPARQL Update リクエストを使用してトリプルを挿入することもできます。詳細については、「SPARQL Update を使用したグラフおよびグラフデータの管理」（222 ページ）を参照してください。

> **注：** セマンティック REST サービスを使用したり、SPARQL クエリで GRAPH '?g' コンストラクタを使用したりする場合、データベースでコレクションレキシコンを有効にする必要があります。

単一のリクエストでトリプルのブロックを MarkLogic サーバーにアップロードするには、`DatabaseClient.graphs.write` を使用します。多数のトリプルを MarkLogic サーバーに段階的にストリーミングするには、`DatabaseClient.graphs.writeStream` を使用します。詳細については、「データベースへのストリーミング」（25 ページ）を参照してください。入力セマンティックデータは、文字列、オブジェクト、または `Readable` ストリームとして表現されます。

`DatabaseClient.graphs.write` または `DatabaseClient.graphs.writeStream` の呼び出しには、少なくとも入力トリプルの形式を示す MIME タイプパラメータが必ず含まれます。また、グラフ URI も含めている場合は、トリプルはそのグラフに読み込まれます。グラフ URI を含めていない場合、トリプルはデフォルトのグラフに読み込まれます。つまり、宛先のグラフに応じて、次のいずれかの形式を使用できます。

```
// load into the default graph
db.graphs.write(mimeType, triples)

// load into a named graph
db.graphs.write(graphURI, mimeType, triples)
```

必要に応じて、ブール型のリペアフラグを渡すこともできます。フラグが存在し、true に設定されている場合、MarkLogic サーバーは読み込みの際に無効な入力トリプルのリペアを試行します。次に例を示します。

```
db.graphs.write(graphURI, true, mimeType, triples)
```

また、位置パラメータではなく呼び出しオブジェクトで書き込み関数を呼び出すこともできます。呼び出しオブジェクトには、次のプロパティが用意されています。

```
db.graphs.write({
  uri: graphURI,              // optional, omit for default
graph
  contentType: mimeType,     // required
  data: triples,             // required
  repair: boolean            // optional
})
```

`DatabaseClient.graphs.write` または `DatabaseClient.graphs.writeStream` の呼び出しの出力は、トリプルがデフォルトグラフまたは名前付きグラフのどちらに読み込まれたのかを示すオブジェクトです。宛先が名前付きグラフの場合は、そのグラフ名が `graph` プロパティに返されます。次に例を示します。

```
// result of loading into default graph
{ defaultGraph: true, graph: null }

// result of loading into a named graph
{ defaultGraph: false, graph: 'example-graph-uri' }
```

次の例は、`DatabaseClient.graphs.write` を使用して、RDF/JSON 形式の一連のトリプルをデフォルトグラフにロードします。

```
var fs = require('fs');
var marklogic = require('marklogic');
var my = require('./my-connection.js');

var db = marklogic.createDatabaseClient(my.connInfo);

var triples = {
  'http://dbpedia.org/resource/Joyce_Carol_Oates' : {
    'http://dbpedia.org/property/influences' : [ {
```

```
      'type' : 'uri',
      'value' :
'http://dbpedia.org/resource/Ernest_Hemingway'
    } ] ,
    'http://dbpedia.org/ontology/influencedBy' : [ {
      'type' : 'uri',
      'value' :
'http://dbpedia.org/resource/Ernest_Hemingway'
    } ]
  },
  'http://dbpedia.org/resource/Death_in_the_Afternoon' : {
    'http://dbpedia.org/ontology/author' : [ {
      'type' : 'uri',
      'value' :
'http://dbpedia.org/resource/Ernest_Hemingway'
    } ] ,
    'http://dbpedia.org/property/author' : [ {
      'type' : 'uri',
      'value' :
'http://dbpedia.org/resource/Ernest_Hemingway'
    } ]
  }
};

db.graphs.write('application/rdf+json', triples).result(
    function(response) {
      if (response.defaultGraph) {
        console.log('Loaded into default graph');
      } else {
       console.log('Loaded into graph ' + response.graph);
      };
    },
    function(error) { console.log(JSON.stringify(error)); }
);
```

次の例は、DatabaseClient.graphs.writeStream を使用して、N-Quads 形式の一連のトリプルを名前付きグラフ（example-graph）にロードします。このトリプルは、Readable ストリームを使用してファイル「input.nq」からストリーミングされ、DatabaseClient.graphs.createWriteStream によって返された Writable ストリームにパイプされます。入力ストリームからのすべてのデータが転送されるまでトランザクションはコミットされません。

```
var fs = require('fs');
var marklogic = require('marklogic');
var my = require('./my-connection.js');
```

```
var db = marklogic.createDatabaseClient(my.connInfo);
// Load into a named graph using a write stream
var writer =
  db.graphs.createWriteStream('example-graph',
'application/n-quads');
writer.result(
  function(response) {
    if (response.defaultGraph) {
      console.log('Loaded triples into default graph');
    } else {
      console.log('Loaded triples into graph ' +
response.graph);
    };
  },
  function(error) { console.log(JSON.stringify(error)); }
);
fs.createReadStream('input.nq').pipe(writer);
```

セマンティックグラフを読み取るには `DatabaseClient.graphs.read` を使用します。詳細については、「グラフのコンテンツ、メタデータ、またはパーミッションの取得」（219 ページ）を参照してください。セマンティックデータをクエリするには、`DatabaseClient.graphs.sparql` を使用します。詳細については、「SPARQL を使用したセマンティックトリプルのクエリ」（213 ページ）を参照してください。その他の操作については Node.js API リファレンスを参照してください。

5.3 SPARQL を使用したセマンティックトリプルのクエリ

データベース内のトリプルに対して SPARQL クエリを評価するには、`DatabaseClient.graphs.sparql` メソッドを使用します。MarkLogic での SPARQL クエリの使用方法の詳細については、『Semantics Developer's Guide』を参照してください。

注： セマンティック REST サービスを使用したり、SPARQL クエリで GRAPH '?g' コンストラクタを使用したりする場合、データベースでコレクションレキシコンを有効にする必要があります。

`DatabaseClient.graphs.sparql` を呼び出すことで、SPARQL クエリを評価できます。クエリは、`result()` を呼び出すまで、評価のために MarkLogic に送信されません。次の方法で `sparql` メソッドを呼び出すことができます。

```
// (1) db.graphs.sparql(responseContentType, query)
db.graphs.sparql(
  'application/sparql-results+json',
  'SELECT ?s ?p WHERE {?s ?p Paris }')
```

```
// (2) db.graphs.sparql(responseContentType,
defaultGraphUris, query)
db.graphs.sparql(
  'application/sparql-results+json',
  'http://def/graph1', 'http://def/graph2',
  'SELECT ?s ?p WHERE {?s ?p Paris }')

// (3) db.graphs.sparql(callObject)
db.graphs.sparql({
  contentType: 'application/sparql-results+json',
  query: 'SELECT ?s ?p WHERE {?s ?p Paris }',
  start: 0,
  length: 15
})
```

少なくとも SPARQL クエリ文字列とレスポンスコンテンツタイプ（MIME タイプ）を呼び出しに含める必要があります。受け付けられるレスポンス MIME タイプの完全なリストについては、『Semantics Developer's Guide』の「SPARQL Query Types and Output Formats」を参照してください。

呼び出しオブジェクトを渡すと、名前付きグラフ URI、追加ドキュメントクエリ、結果サブセットコントロール、推論ルールセットなどの、クエリの属性を設定できます。設定プロパティの完全なリストについては、『Node.js Client API Reference』の「`graphs.sparql`」を参照してください。

5.4 例：SPARQL クエリ

次の例は、文字列リテラルとして表した SPARQL クエリをデフォルトグラフに対して評価します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');

var db = marklogic.createDatabaseClient(my.connInfo);
var vb = marklogic.valuesBuilder;

var query = [
  'PREFIX foaf: <http://xmlns.com/foaf/0.1/>' ,
  'PREFIX ppl:  <http://people.org/>' ,
  'SELECT ?personName1' ,
  'WHERE {' ,
  '    ?personUri1 foaf:name  ?personName1 ;' ,
  '                foaf:knows ppl:person3 .',
  '    ?personUri1 foaf:name  ?personName1 .',
  '    }'
];
```

```
db.graphs.sparql('application/sparql-results+json',
query.join('\n')
).result(function (result) {
  console.log(JSON.stringify(result, null, 2));
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

スクリプトを実行すると次のような出力が生成され、Person 1 と Person 2 が Person 3 を知っていることを示すトリプルになります。

```
{ "head": {
    "vars": [ "personName1" ]
  },
  "results": {
    "bindings": [
      {
        "personName1": {
          "type": "literal",
          "value": "Person 1",
          "datatype":
"http://www.w3.org/2001/XMLSchema#string"
        }
      },
      {
        "personName1": {
          "type": "literal",
          "value": "Person 2",
          "datatype":
"http://www.w3.org/2001/XMLSchema#string"
        }
      }
    ]
}}
```

その他の例については、`node-client-api` プロジェクトソースディレクトリの `test-basic/graphs.js` を参照してください。

5.5 グラフの管理

このセクションでは、次のグラフ管理操作について取り上げます。

- グラフの作成または置換
- 既存のグラフへのトリプルの追加
- グラフの削除

- グラフのコンテンツ、メタデータ、またはパーミッションの取得
- グラフの有無のテスト
- グラフのリストの取得

SPARQL Update を使用して同様の操作を実行することもできます。詳細については、「SPARQL Update を使用したグラフおよびグラフデータの管理」（222 ページ）を参照してください。

注： 多くのグラフ管理操作は、managed triples にのみ適用されます。

5.5.1 グラフの作成または置換

`DatabaseClient.graphs.write` を使用して、SPARQL Update を使用せずに、グラフを作成または置換できます。詳細および例については、「トリプルのロード」（210 ページ）を参照してください。`DatabaseClient.graphs.sparqlUpdate` を使用して、グラフを作成または修正することもできます。詳細については、「SPARQL Update を使用したグラフおよびグラフデータの管理」（222 ページ）を参照してください。

書き込むときにグラフが存在しない場合、グラフが作成されます。書き込むときにグラフがすでに存在している場合、グラフ内の unmanaged triples は、新しいトリプルに置き換えられます。これはグラフの削除および再作成と同等です。

グラフ内の管理対象外のトリプルは、この操作の影響を受けません。

また、『Node.js Client API Reference』を参照してください。

5.5.2 既存のグラフへのトリプルの追加

`DatabaseClient.graphs.merge` を使用して、現在のコンテンツを置き換えずに、既存のグラフにトリプルを追加します。他の更新操作またはより詳細な制御には、SPARQL Update を使用し、`DatabaseClient.graphs.sparqlUpdate` を使用してグラフとやり取りします。詳細については、「SPARQL Update を使用したグラフおよびグラフデータの管理」（222 ページ）を参照してください。グラフのコンテンツを置換するには、`DatabaseClient.graphs.write` を使用します。詳細については、「グラフの作成または置換」（216 ページ）を参照してください。

次の形式を使用して `graphs.merge` を呼び出せます。

```
// Add triples to a named graph
db.graphs.merge(graphUri, contentType, tripleData)

// Add triples to the default graph
db.graphs.merge(null, contentType, tripleData)
```

```
// Add triples to a named graph or the default graph using
a call object
db.graphs.merge({uri: ..., contentType: ..., data: ...,
...})
```

呼び出しオブジェクトパターンを使用して、パーミッションやトランザクション ID などの操作パラメータを指定します。詳細については、『Node.js Client API Reference』を参照してください。呼び出しオブジェクトパターンを使用すると、uri プロパティを省略するか、これを null に設定してトリプルをデフォルトのグラフに結合できます。

次の例は、呼び出しオブジェクトパターンを使用して、必要に応じてデータを修正しながら、2 つの新しいトリプルを「MyGraph」というグラフに挿入します。この例では、トリプルは Turtle 形式で文字列として渡されます。他のトリプル形式を使用できます。文字列の代わりに、ストリーム、バッファ、またはオブジェクトとしてトリプルデータを提供することもできます。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');

var db = marklogic.createDatabaseClient(my.connInfo);

var triples = [
  '@prefix p1: <http://example.org/marklogic/predicate/>
.',
  '@prefix p0: <http://example.org/marklogic/people/> .',
  'p0:Julie_Smith  p1:livesIn \"Sterling\" .',
  'p0:Jim_Smith    p1:livesIn \"Bath\" .'
];
db.graphs.merge({
    uri: 'MyGraph',
    contentType: 'text/turtle',
    data: triples.join('\n'),
    repair: true
}).result(
    function(response) {
      console.log(JSON.stringify(response));
    },
    function(error) { console.log(JSON.stringify(error)); }
);
```

操作が成功した場合、スクリプトは次のような出力を生成します。

```
{"defaultGraph":false,"graph":"MyGraph","graphType":"named
"}
```

また、『Node.js Client API Reference』を参照してください。

5.5.3 グラフの削除

`DatabaseClient.graphs.remove` を使用して、名前付きグラフまたはデフォルトのグラフ内のトリプルを削除します。この操作は、managed triples にのみ影響します。グラフに管理対象外のトリプルが含まれている場合、組み込まれたトリプルは影響を受けず、この操作後もグラフは存続します。

SPARQL Update リクエストを使用してグラフを削除することもできます。詳細については、「SPARQL Update を使用したグラフおよびグラフデータの管理」（222 ページ）を参照してください。

パラメータを指定せずに `remove` を呼び出すと、デフォルトグラフが削除されます。グラフの URI を指定して `remove` を呼び出すと、その名前付きグラフが削除されます。

次の例は、URI が `example-graph` のグラフ内のすべてのトリプルを削除します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.graphs.remove('example-graph').result(
  function(response) {
console.log(JSON.stringify(response)); }
);

// expected output:
// {"defaultGraph":false,"graph":"example-graph"}
```

デフォルトのグラフを削除するには、グラフ URI を呼び出しから省略します。次の例は、デフォルトグラフ内のすべてのトリプルを削除します。これはデフォルトグラフであるため、レスポンスの `graph` プロパティの値は `null` です。名前付きグラフの場合、`graph` プロパティにはグラフの URI が格納されます。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.graphs.remove().result(
  function(response) {
console.log(JSON.stringify(response)); }
);

// expected output:
// {"defaultGraph":true,"graph":null}
```

また、『Node.js Client API Reference』を参照してください。

5.5.4 グラフのコンテンツ、メタデータ、またはパーミッションの取得

グラフ内のすべてのトリプル、グラフメタデータ、またはグラフパーミッションを取得するには、DatabaseClient.graphs.read を使用します。デフォルトのグラフでも名前付きグラフでもこのメソッドを使用できます。

次の形式で graphs.read を呼び出せます。

```
// (1) Retrieve all triples in the default graph
db.graphs.read(responseContentType)

// (2) Retrieve all triples in a named graph
db.graphs.read(uri, responseContentType)

// (3) Retrieve triples, metadata, or permissions using a
call object
db.graphs.read({contentType: ..., ...})
```

呼び出しオブジェクトパターンを使用するときには、少なくとも contentType プロパティをその呼び出しオブジェクトに含める必要があります。名前付きグラフで操作するには、uri プロパティも含める必要があります。uri プロパティを省略するか、これを null に設定した場合は、操作はデフォルトのグラフに適用されます。

複数の PDF 形式でトリプルを取得できます。サポートしている形式のリストについては、graphs.read の API リファレンスを参照してください。

次の例は、「MyGraph」というグラフからすべてのトリプルを読み取ります。トリプルは Turtle 形式で返されます。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');

var db = marklogic.createDatabaseClient(my.connInfo);

db.graphs.read('MyGraph','text/turtle')
  .result(
    function(response) {
      for (var line of response.split('\n')) {
        console.log(line);
      }
    },
    function(error) { console.log(JSON.stringify(error)); }
  );
```

グラフメタデータまたはパーミッションの取得時に呼び出しオブジェクトパターンが要求されます。呼び出しオブジェクトもトリプルの取得に使用できます。取得する対象（コンテンツ、メタデータ、またはパーミッション）を指定するには、呼び出しオブジェクトの `category` プロパティを使用します。

次の例は、「MyGraph」というグラフに関するメタデータを取得します。

```
db.graphs.read({
  uri: 'MyGraph',
  contentType: 'application/json',
  category: 'metadata'
});
```

また、『Node.js Client API Reference』を参照してください。

5.5.5 グラフの有無のテスト

グラフの有無をテストするには、`DatabaseClient.graphs.probe` を使用します。次の例は、URI が「http://marklogic.com/example/graph」であるグラフの有無をテストします。

```
db.graphs.probe('http://marklogic.com/example/graph')
  .result(
    function(response) {
      console.log(JSON.stringify(response));
    },
    function(error) { console.log(JSON.stringify(error)); }
  );
```

グラフが存在する場合、呼び出しは次のような出力を生成します。

```
{ "contentType": null,
  "contentLength": null,
  "versionId": null,
  "location": null,
  "systemTime": null,
  "exists": true,
  "defaultGraph": false,
  "graph": "http://marklogic.com/example/graph",
  "graphType": "named"
}
```

グラフが存在しない場合、グラフは次のような出力を生成します。

```
{ "exists": false,
  "defaultGraph": false,
  "graph": "NoMyGraph",
  "graphType": "named"
}
```

デフォルトのグラフの有無を調べるには、グラフの URI を省略します。

5.5.6 グラフのリストの取得

MarkLogic に格納されているグラフのリストを取得するには、`DatabaseClient.graphs.list` を使用します。レスポンスに対して想定されるコンテンツ MIME を指定する必要があります。`text/plain` または `text/html` としてリストを取得できます。

次の例は、1 行ごとに 1 つの URI を示した使用可能なグラフ URI のリストを取得します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');

var db = marklogic.createDatabaseClient(my.connInfo);

db.graphs.list('text/uri-list')
  .result(
    function(response) {
      for (var uri of response.split('\n')) {
        console.log(uri);
      }
    },
    function(error) { console.log(JSON.stringify(error)); }
);
```

データベースにデフォルトのグラフと URI が「MyGraph」のグラフが含まれている場合、上記の操作の未処理レスポンスは、次の形式の文字列になります。各 URI は改行（「\n」）で区切られます。

```
「MyGraph\nhttp://marklogic.com/semantics#graphs\n」
```

また、『Node.js Client API Reference』を参照してください。

5.6 SPARQL Update を使用したグラフおよびグラフデータの管理

SPARQL Update リクエストを使用して、`DatabaseClient.graphs.sparqlUpdate` を呼び出すことにより、Node.js からのグラフおよびグラフデータを管理できます。managed triples を扱うときにのみこのインターフェイスを使用できます。また、SPARQL Update を使用せずに、グラフおよびグラフデータを管理できます。詳細については、「グラフの管理」（215 ページ）と「トリプルのロード」（210 ページ）を参照してください。

次の方法で、`graphs.sparqlUpdate` を呼び出せます。

```
// (1) pass only the SPARQL Update as a string or
ReadableStream
db.graphs.sparqlUpdate(updateOperation)

// (2) pass a call config object that includes the SPARQL
Update
db.graphs.sparqlUpdate({data: updateOperation, ...})
```

呼び出しオブジェクトパターンを使用すると、パーミッション、推論ルールセット、トランザクション制御、変数バインドなど、操作の詳細を制御できます。呼び出しオブジェクトを使用すると、`data` プロパティは、SPARQL Update リクエスト文字列またはストリームを保持します。

次の例は、SPARQL Update リクエストだけを渡し、グラフを作成します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');

var db = marklogic.createDatabaseClient(my.connInfo);

db.graphs.sparqlUpdate(
    'CREATE GRAPH
<http://marklogic.com/semantics/tutorial/update> ;'
).result(
    function(response) {
      console.log('Graph created.');
      console.log(JSON.stringify(response));
    },
    function(error) { console.log(JSON.stringify(error)); }
);
```

このスクリプトを実行すると、次の出力が生成されます。

```
Graph created.
{"defaultGraph":false,"graph":null,"graphType":"inline"}
```

次の例は呼び出しオブジェクトを使用して、グラフを作成し、トリプルを挿入します。呼び出しオブジェクトは、グラフにパーミッションを定義し、SPARQL Update リクエストで使用される変数バインドを定義します。

```
var graphURI = 'http://marklogic.com/sparqlupd/example';
db.graphs.sparqlUpdate({
    data:
      'CREATE GRAPH <' + graphURI + '> ;' +
      'INSERT {GRAPH <' + graphURI + '> {?s ?p ?b1 }} '+
      'WHERE  {GRAPH <' + graphURI + '> ' +
          '{?s ?p ?o. filter (?p = ?b2) }}',
    bindings: {
      b1: 'bindval1',
      b2: {value: 'bindval2', type: 'string'}
    },
    permissions: [
      { 'role-name': 'app-user', capabilities: ['read']},
      { 'role-name': 'admin', capabilities:
['read','update']},
    ]
})
```

呼び出しオブジェクトで使用できる設定プロパティの完全なリストについては、『Node.js Client API Reference』の「`graphs.sparqlUpdate`」を参照してください。

MarkLogic での SPARQL Update の使用方法の詳細については、『Semantics Developer's Guide』の「SPARQL Update」を参照してください。

5.7 SPARQL クエリまたは Update への推論ルールの適用

SPARQL クエリまたは SPARQL Update リクエストに適用する、1 つあるいは複数の推論ルールセットを指定できます。推論ルールを使用すれば、クエリまたは更新時にデータに関する新しいファセットを「検出」できます。

SPARQL の推論は、『Semantics Developer's Guide』の「Inference」で詳しく説明されています。このセクションでは、Node.js クライアント API に特有な使用方法の詳細だけを扱います。

ここでは、次の内容を取り上げます。

- 基本的な推論ルールセットの使用方法
- 例：推論ルールセットを使用した SPARQL クエリ
- 例：推論ルールセットを使用した SPARQL Update
- デフォルトのデータベースルールセットの制御

5.7.1 基本的な推論ルールセットの使用方法

`graphs.sparql` または `graphs.sparqlUpdate` を使用するときに、1 つあるいは複数の推論ルールセットを含めるには、`rulesets` プロパティを含んだ呼び出し設定オブジェクトを渡します。

例えば、`graphs.sparql` で 1 つあるいは複数のルールセットを指定するには、少なくとも次のプロパティを含む入力呼び出しオブジェクトを作成します。

```
db.graphs.sparql({'content-type': ..., query: ...,
rulesets: ...})
```

`graphs.sparqlUpdate` で 1 つあるいは複数のルールセットを指定するには、少なくとも次のプロパティを含む入力呼び出しオブジェクトを作成します。

```
db.graphs.sparqlUpdate({data: ..., rulesets: ...})
```

`rulesets` プロパティの値は、単一の文字列にすることも文字列の配列にすることもできます。`rulesets` 内の各文字列は、ビルトインルールセットファイルの名前か、スキーマデータベースにインストールされているカスタムルールセットの URI のどちらかにする必要があります。

ビルトインルールセットの詳細については、『Semantics Developer's Guide』の「Rulesets」を参照してください。

カスタムルールセットの作成の詳細については、『Semantics Developer's Guide』の「Creating a New Ruleset」を参照してください。
`DatabaseClient.documents.write` など、標準的なドキュメント操作を使用してスキーマデータベースにカスタムルールセットをインストールできます。

MarkLogic でのセマンティック推論の詳細については、『Semantics Developer's Guide』の「Inference」を参照してください。

5.7.2 例：推論ルールセットを使用した SPARQL クエリ

次の例は、ビルトイン `subPropertyOf.rules` ルールセットをクエリに適用します。このルールセットは、MarkLogic をインストールするときに、*MARKLOGIC_INSTALL_DIR*`/Config` ディレクトリに自動的にインストールされます。

```
db.graphs.sparql({
  contentType: 'application/sparql-results+json',
  query: 'SELECT ?s ?p WHERE {?s ?p Paris }',
  rulesets: 'subPropertyOf.rules'
})
```

5.7.3 例：推論ルールセットを使用した SPARQL Update

次の例は、ビルトイン `sameAs.rules` ルールセットと、URI が `/my/rules/custom.rules` のスキーマデータベースにインストールされているカスタムルールセットを、SPARQL Update リクエストに適用します。

```
var update = [
  PREFIX exp: <http://example.org/marklogic/people>
  PREFIX pre: <http://example.org/marklogic/predicate>
  INSERT DATA {
    GRAPH <MyGraph>{
      exp:John_Smith pre:livesIn "London" .
      exp:Jane_Smith pre:livesIn "London" .
      exp:Jack_Smith pre:livesIn "Glasgow" .
  }}
];
db.graphs.sparqlUpdate({
  data: update.join('\n'),
  rulesets: ['sameAs.rules', '/my/rules/custom.rules']
})
```

5.7.4 デフォルトのデータベースルールセットの制御

すべてのデータベースには、暗黙のデフォルトの推論ルールセットがあります。『Semantics Developer's Guide』の「Using the Admin UI to Specify a Default Ruleset for a Database」で説明しているように、データベースのデフォルトのルールセットはカスタマイズできます。

データベースのデフォルトのルールセットは通常、すべての SPARQL クエリおよび更新操作に適用されます。ただし、入力呼び出しオブジェクトの `defaultRulesets` プロパティを使用することにより、クエリおよび更新操作に、データベースのデフォルトのルールセットを含めるかどうかを制御できます。

データベースのデフォルトのルールセットを推論から除外するには、`defaultRulesets` を「exclude」に設定します。データベースのデフォルトのルールセットを含めるには、`defaultRulesets` を「include」に設定します。明示的に `defaultRulesets` を設定しない場合、データベースのデフォルトのルールセットが操作に含まれます。

次の例は、クエリ操作からデフォルトのルールセットを除外します。

```
db.graphs.sparql({
  contentType: 'application/sparql-results+json',
  query: 'SELECT ?s ?p WHERE {?s ?p Paris }',
  rulesets: 'subPropertyOf.rules',
  defaultRuleSets: 'exclude'
})
```

6.0 トランザクションの管理

この章では、Node.js クライアント API を使用したトランザクション管理に関する次のトピックについて説明します。

6.1 トランザクションの概要

このセクションでは、Node.js クライアント API で使用する MarkLogic サーバートランザクションモデルの概要について説明します。MarkLogic トランザクションモデルの詳細については、『Application Developer's Guide』の「Understanding Transactions in MarkLogic Server」を参照してください。

デフォルトでは、データベースに対する各操作は、シングルステートメントトランザクションに相当します。つまり、操作は 1 つのトランザクションとして評価されます。例えば、`DatabaseClient.documents.write` を使用してデータベース内の 1 つあるいは複数のドキュメントを更新すると、実質的にはサーバーサイドハンドラが新しいトランザクションを作成し、ドキュメントを更新し、トランザクションをコミットして応答を返すことになります。更新したドキュメントは、書き込み操作が正常に完了すると、データベースに表示され、他の操作に利用できます。いずれかのドキュメントの更新中にエラーが発生した場合、操作全体が失敗します。

また、Node.js クライアント API により、アプリケーションはトランザクションの境界を直接制御できるようになり、同一トランザクションコンテキスト内で複数の操作を評価できます。これは、『Application Developer's Guide』の「Multi-Statement Transaction Concept Summary」で説明しているマルチステートメントトランザクションと同じです。

マルチステートメントトランザクションを使用すると、複数の操作を実行して、それらを 1 つのトランザクションとしてコミットできます。これにより、データベース内で関連する更新がすべて行われるか、全く行われないようにできます。Node.js クライアント API のドキュメント操作および検索機能は、`DatabaseClient.transactions` インターフェイスを通じたマルチステートメントトランザクションと、ほとんどの操作にトランザクションオブジェクトを渡す処理をサポートしています。

マルチステートメントトランザクションを使用するには、次の手順に従います。

1. `DatabaseClient.transactions.open` を使用してマルチステートメントトランザクションを作成します。この操作はトランザクションオブジェクトを返します。「トランザクションの作成」（229 ページ）を参照してください。

2. `txid` パラメータのトランザクションオブジェクトを含めて、トランザクションのコンテキストで 1 つあるいは複数の操作を実行します。「トランザクションと操作の関連付け」（230 ページ）を参照してください。

3. `DatabaseClient.transactions.commit` を使用してトランザクションをコミットするか、`DatabaseClient.transactions.rollback` を使用してトランザクションをロールバックします。詳細は「トランザクションのコミット」（231 ページ）および「トランザクションのロールバック」（231 ページ）を参照してください。

アプリケーションがロードバランサーを通じて MarkLogic サーバーとやり取りする場合は、セッションアフィニティを維持するためにリクエストに HostId クッキーを含める必要があります。詳細については、「ロードバランサー使用時のトランザクションの管理」（233 ページ）を参照してください。

トランザクションを明示的に作成した場合は、それを明示的にコミットするかロールバックする必要があります。そのようにしないと、リクエストまたはトランザクションがタイムアウトになるまでトランザクションが実行されたまま（オープンなまま）になります。実行しているトランザクションは、ロックを保持し、システムリソースを消費するため、完了後はトランザクションを終了させることが重要です。

トランザクションがコミットされる前に、リクエストまたはトランザクションがタイムアウトになると、そのトランザクションは自動的にロールバックされ、すべての更新が破棄されます。アプリケーションサーバーのリクエストのタイムアウトは、Admin UI で設定します。個別のトランザクションのタイムアウトを設定するには、トランザクション作成時に `timeLimit` リクエストパラメータを設定します。

6.2 トランザクションの作成

マルチステートメントトランザクションを作成するには、
`DatabaseClient.transactions.open` を使用します。次に例を示します。

```
var txObj = null;
db.transactions.open().result()
  .then(function(response) {
    txObj = response
  });
```

この呼び出しは、ロードバランサーがある場合でも、トランザクション全体でホストアフィニティを維持するために必要な状態をカプセル化するトランザクションオブジェクトを返します。

マルチステートメントトランザクションは明示的にコミットまたはロールバックする必要があります。リクエストがタイムアウトになる前にトランザクションをコミットまたはロールバックしないと、自動的にロールバックされます。`open` に制限時間（秒単位）を指定して、トランザクションに短い制限時間を割り当てることができます。例えば、次の呼び出しは時間制限を `tlimit` に設定し、ステートフルトランザクションを返します。

```
db.transactions.open({timeLimit: tlimit})
```

しかし、制限時間を使ってトランザクションをロールバックさせないでください。制限時間は、安全策（フェールセーフ）として用意されているものです。制限時間まで待つのではなく、適切な時期にトランザクションを明示的にロールバックしてください。

また、トランザクションの作成時にシンボリック名を指定することもできます。
トランザクションオブジェクト（または ID）は、トランザクションパラメータを受け付けるすべての操作で使用する必要がありますが、
`DatabaseClient.transactions.read` ではシンボリック名を使用でき、
管理画面などのトランザクションステータスに表示されます。

例えば次の呼び出しでは、適切なプロパティ名の入力呼び出しオブジェクトを使用して、制限時間と名前の両方を提供しています。

```
db.transactions.open({
  timeLimit: 45,
  transactionName: 'mySpecialTxn'
});
```

6.3 トランザクションと操作の関連付け

DatabaseClient.transactions.open を使用してトランザクションを作成したら、結果のトランザクションオブジェクト（または ID）をさまざまな操作に渡して、特定のトランザクションのコンテキストで操作を実行できます。

例えば、特定のマルチステートメントトランザクションのコンテキストでドキュメントを更新するには、DatabaseClient.documents.write 呼び出しにトランザクション ID を含めます。

```
var txnObj = null;
db.transactions.open(true).result()
  .then(function(response) {
     txnObj = response;
     return db.documents.write({
       uri: '/my/documents.json',
       content: {some: 'content'},
       contentType: 'application/json”,
      txid: txnObj
     }).result;
  ...
```

マルチステートメントトランザクションの一部である更新は、この同一トランザクション内の後続の操作では認識されますが、トランザクションがコミットされるまでそれ以外の場所では認識されません。

複数のトランザクションを同時に行うことができます。その際に、他のユーザーがこの同じデータベースを同時に使用できます。競合を防止するため、トランザクションで更新が発生する場合は、ドキュメントがコミットまたはロールバックされるまでトランザクションは常にロックされます。このため、リソースの競合を回避するために、できる限り短期間でトランザクションをコミットまたはロールバックする必要があります。

> **注：** 操作を実行するデータベースコンテキストは、トランザクションが作成されたデータベースコンテキストと同じでなければなりません。一貫性が保証されるのは、単一の DatabaseClient 設定を使用している場合のみです。

トランザクションの一部ではない操作とトランザクションの一部である操作は、混在させることができます。txid パラメータまたは呼び出しオブジェクトプロパティを持たない操作は、マルチステートメントトランザクションの一部ではありません。ただし、通常はこれらの操作を同一トランザクション内にグループ化するため、適切なタイミングでトランザクションをコミットまたはロールバックできます。

6.4 トランザクションのコミット

マルチステートメントトランザクションをコミットするには、DatabaseClient.transactions.commit を使用します。commit 呼び出しでは、DatabaseClient.transactions.open からトランザクションオブジェクト（または ID）を取得します。次に例を示します。

```
db.transactions.commit(transactionObj);
```

トランザクションは、いったんコミットするとロールバックできなくなり、トランザクションオブジェクト（または ID）は使用できなくなります。別のトランザクションを実行するには、open を呼び出して新しいトランザクションを取得します。

注： トランザクションをコミットまたはロールバックするデータベースコンテキストは、トランザクションが作成されたデータベースコンテキストと同じでなければなりません。一貫性が保証されるのは、単一の DatabaseClient 設定を使用している場合のみです。

6.5 トランザクションのロールバック

エラーまたは例外が発生した場合は、DatabaseClient.transactions.rollback を使用して実行中の（オープンな）トランザクションをロールバックできます。

```
db.transactions.rollback(transactionObj);
```

rollback を呼び出すと、トランザクションの残りの部分がキャンセルされ、データベースはトランザクション開始前の状態に戻ります。タイムアウトになるのを待つのではなく、明示的にトランザクションをロールバックすることをお勧めします。

トランザクションをロールバックするには、rest-writer または rest-admin ロールまたは同等の権限を持っている必要があります。

注： トランザクションをコミットまたはロールバックするデータベースコンテキストは、トランザクションが作成されたデータベースコンテキストと同じでなければなりません。一貫性が保証されるのは、単一の DatabaseClient 設定を使用している場合のみです。

マルチステートメントトランザクションを扱う場合、rollback を呼び出す catch 節を含めることにより、エラーが発生した場合に明示的にトランザクションがロールバックされていることを確認する必要があります。次に例を示します。

```
var txnObj = null;
db.transactions.open(true).result().
  then(function(response) {
    txnObj = response;
    return db.documents.read({uris: oldUri, txid:
```

```
txnObj}).result();
    }).
  then(...).
  catch(function() {
    db.transactions.rollback(txnObj);
    });
```

6.6 例：マルチステートメントトランザクションでの Promise の使用

次の関数は、マルチステートメントトランザクション内で操作を同期させるための、Promise パターンの使用法を示したものです。Promise の詳細については、「Promise 結果処理パターン」（23 ページ）を参照してください。

この関数はドキュメントを「移動」します。つまり初期 URI からコンテンツを読み取り、そのコンテンツを新しい URI でデータベースに挿入し、元のドキュメントを削除するという処理を行います。この関数は、まずトランザクションを作成し、そのトランザクションのコンテキストで読み取り、書き込み、および削除操作を実行します。これらの操作が完了すると、トランザクションがコミットされます。エラーが発生した場合、トランザクションはロールバックされます。

```
function transactionalMove(oldUri, newUri) {
  var txnObj = null;
  db.transactions.open(true).result().
    then(function(response) {
      txnObj = response;
      return db.documents.read({uris: oldUri, txid:
txnObj}).result();
      }).
    then(function(documents) {
      documents[0].uri = newUri;
      return db.documents.write(
        {documents: documents, txid: txnObj}).result();
      }).
    then(function(response) {
      return db.documents.remove({uri: oldUri, txid:
txnObj}).result();
      }).
    then(function(response) {
      return db.transactions.commit(txnObj).result();
      }).
    catch(function(error) {
      console.log('ERROR: ' + JSON.stringify(error));
      db.transactions.rollback(txnObj);
      });
}
```

6.7 トランザクションのステータスの確認

トランザクションのステータスをクエリするには、
`DatabaseClient.transactions.read` を使用します。次に例を示します。

```
db.transactions.read(transactionObj)
```

6.8 ロードバランサー使用時のトランザクションの管理

このセクションは、マルチステートメントトランザクションを使用しロードバランサーを通じて MarkLogic サーバークラスタとやり取りする、クライアントアプリケーションのみに関するものです。

ロードバランサーを使用している場合は、同じセッション内であっても、アプリケーションから MarkLogic サーバーへのリクエストを別のホストにルーティングできます。これは MarkLogic サーバーとのほとんどのインタラクションに影響を及ぼしませんが、同一マルチステートメントトランザクションの操作は MarkLogic クラスタ内の同一ホストにルーティングする必要があります。このようにロードバランサーを使用した一貫したルーティングのことをセッションアフィニティといいます。

適切にセッションアフィニティを維持するには、単純な文字列トランザクション ID ではなく、トランザクションオブジェクトを返す方法で、
`DatabaseClient.transactions.open` を呼び出す必要があります。つまり、`withState` パラメータ（または呼び出しオブジェクトプロパティ）が明示的に false に設定されてはいないことを確認する必要があります。`transactions.open` はデフォルトでオブジェクトを返します。

例えば、次の呼び出しは要求されたトランザクションオブジェクトを返します。

```
...db.transactions.open();

...db.transactions.open({withState: true, ...});
```

このようにトランザクションを開始すると、返されたトランザクションオブジェクトで `HostId` クッキーがキャッシュされます。アプリケーションが、MarkLogic サーバーにリクエストを送る操作にトランザクションオブジェクトを渡すときには、`HostId` クッキーがリクエストに自動的に含められます。`HostId` クッキーを使用するようにロードバランサーを設定して、セッションアフィニティを維持できます。

`HostId` クッキーによる、セッションアフィニティ維持のためのロードバランサー設定手順は、ロードバランサーによって異なります。詳細については、ロードバランサーのドキュメントを参照してください。

ロードバランサーを通じてリクエストをルーティングしない場合、`HostId` クッキーは無視されます。Node.js クライアント API は、`HostId` クッキーを保持しません。クッキーにはセッション状態は一切含まれていません。API が、このクッキーの値を他の目的で使用することはありません。

7.0 拡張機能、変換機能、サーバーサイドコードの実行

この章では、拡張機能と変換機能の作成および使用方法、および Node.js クライアント API を使用して MarkLogic サーバーでコードの任意のブロックとライブラリモジュールを実行する方法に関する次のトピックについて説明します。

7.1 API を拡張およびカスタマイズする方法

特定の拡張ポイントを通じて、またはアプリケーションから任意のサーバーサイドコードを実行して、Node.js クライアント API の動作を拡張およびカスタマイズできます。

- コンテンツの変換：ドキュメントをデータベースに書き込むとき、またはデータベースから読み取るときに、ユーザー定義の変換関数を適用できます。詳細については、「コンテンツ変換機能を扱う」（245 ページ）を参照してください。また、パッチ機能についてカスタムの置換コンテンツジェネレータを定義することもできます。詳細については、「MarkLogic サーバーでの置換データの作成」（116 ページ）を参照してください。
- 検索結果の変換：ドキュメントや値をクエリする際に、検索結果のサマリまたはマッチするドキュメントにユーザー定義の変換関数を適用できます。詳細については、「コンテンツ変換機能を扱う」（245 ページ）を参照してください。
- リソースサービスの拡張：`DatabaseClient.resources` インターフェイスを使用して、Node.js からアクセス可能な独自の REST エンドポイントを定義します。リソースサービス拡張の詳細については、この章で説明します。手順については、「コンテンツ変換機能を扱う」（245 ページ）を参照してください。
- アドホッククエリの実行：XQuery または JavaScript コードの任意のブロックを MarkLogic サーバーに送信して評価します。詳細については、「アドホックコードとサーバーサイドモジュールの評価」（266 ページ）を参照してください。
- サーバーサイドモジュールの評価：ユーザー定義の XQuery または JavaScript モジュールを MarkLogic サーバーにインストールした後で、それらを評価します。詳細については、「アドホックコードとサーバーサイドモジュールの評価」（266 ページ）を参照してください。

これらの機能に加え、API には、カスタム制約パーサー、ファセット、スニペットジェネレータ、ドキュメントパッチコンテンツジェネレータなど、ユーザー定義のサーバーサイドコード用のフックが用意されています。そのようなコードやリソースサービスの拡張機能および変換機能を使用するには、それらを REST API インスタンスに関連付けられた modules データベースにインストールしておく必要があります。Node.js API には、このような専用アセットをインストールするためのインターフェイスが用意されています。また依存するライブラリなどのアセットを、`DatabaseClient.config` インターフェイスからインストールできます。詳細については、「アセット管理の概要」（275 ページ）を参照してください。

7.2 リソースサービス拡張機能を扱う

このセクションでは、リソースサービス拡張機能の概念と、その作成、インストール、使用、および管理方法について説明します。ここでは、次の内容を取り上げます。

7.2.1 リソースサービス拡張機能とは

リソースサービス拡張機能は、XQuery モジュールおよびサーバーサイド JavaScript モジュールに対する RESTful インターフェイスを作成することで Node.js クライアント API を拡張します。サーバーサイド拡張機能は、REST クライアント API から拡張として受け取った GET、PUT、POST、および DELETE といった HTTP リクエストを処理するための関数を実装します。Node.js クライアント API を使用すると、`DatabaseClient.resources` インターフェイスを介してこれらのメソッドを呼び出すことができます。独自の Node.js インターフェイスを `DatabaseClient.resources` 操作にラップして、ドメイン固有の方法でサービスを公開できます。

例えば、MarkLogic サーバー上で単語を検索したり、スペルチェックしたり、不明な単語の候補を示す辞書プログラムリソース拡張を作成できます。`lookUpWords()` や `spellCheck()` など、アプリケーションプログラマが呼び出すことができる個別の操作は、リソース拡張を公開するドメイン固有のサービスです。

Node.js クライアント API を使用してリソース拡張を作成および使用する基本的な手順は、次のとおりです。

1. リソース用サービスを実装する XQuery または JavaScript モジュールを作成します。

2. `DatabaseClient.config.resources.write` を使用して、REST API インスタンスに関連付けられた modules データベースにリソースサービスの拡張実装をインストールします。

3. `DatabaseClient.resources.get` などの `DatabaseClient.resources` 操作を使用して、リソース拡張メソッドにアクセスします。

`DatabaseClient.config.resources` インターフェイスは、インストール済み拡張機能を動的に検出することもできます。拡張機能をインストールする際に、メソッドのパラメータ名やタイプ情報などのメタデータを指定することで、動的に検出された拡張機能を使いやすくできます。ここでメタデータは、単に情報として利用されています。

拡張機能が依存する他のモジュールまたはアセットは、`DatabaseClient.extlibs` インターフェイスを使用して modules データベースにインストールできます。詳細については、「modules データベースでのアセットの管理」（275 ページ）を参照してください。

完全な例については、「例：リソースサービス拡張機能のインストールと使用」（240 ページ）を参照してください。

7.2.2 リソースサービス拡張機能の作成

リソースサービス拡張機能は、サーバーサイド JavaScript または XQuery を使用して実装できます。インターフェイスは複数の MarkLogic クライアント API 間で共有されるため、Java クライアント API、Node.js クライアント API、および REST クライアント API で同じ拡張機能を使用できます。

1 つのクライアント API で拡張機能をインストールすれば、それをすべての API で使用できます。例えば、REST クライアント API を使用してインストールしたリソースサービス拡張機能は、Node.js クライアント API および Java クライアント API を使用して実装したアプリケーションで使用できます。

インターフェイスの定義、オーサリングのガイドライン、および例の実装については、『REST Application Developer's Guide』の「Extending the REST API」を参照してください。

7.2.3 リソースサービス拡張機能のインストール

リソース拡張機能を使用するには、MarkLogic サーバーにその実装をインストールする必要があります。リソースサービス拡張機能をインストールするには、`rest-admin` ロールまたは同等の権限を持っている必要があります。

拡張機能の実装をインストールするには、次の手順に従います。

1. リソース拡張機能が追加のライブラリモジュールに依存する場合は、それらの依存ライブラリを MarkLogic サーバーにインストールします。詳細については、「modules データベースでのアセットの管理」（275 ページ）を参照してください。

2. 必要に応じて、プロバイダ、説明、およびバージョンなどの属性を表す、拡張機能のメタデータを定義します。

3. `DatabaseClient.config.resources.write` を呼び出して、`DatabaseClient` オブジェクトに関連付けられている REST API インスタンスの modules データベースに拡張機能をインストールします。呼び出しでは、拡張機能の名前、実装言語（XQuery または JavaScript）、および実装ソースコードを提供する必要があります。また、オプションのメタデータを含めることもできます。

   注： XQuery 拡張機能の場合、その拡張機能は、拡張機能モジュールの名前空間の宣言内の名前と同じ名前でインストールする必要があります。例えば、次のモジュールの名前空間を持つ XQuery 拡張機能は「example」としてインストールする必要があります。

```
xquery version "1.0-ml";
module namespace yourNSPrefix =
  "http://marklogic.com/rest-api/resource/example";
...
```

例えば、次のコードは、JavaScript 拡張機能を「js-example」という名前で、メタデータなしでインストールします。拡張機能の実装は、ファイル `js-example.sjs` からストリーミングされます。

```
var fs = require('fs');
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.config.resources.write(
  'js-example', 'javascript',
  fs.createReadStream('./js-example.sjs')
```

```
).result(function(response) {
  console.log('Installed extension: ' + response.name);
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

次のコードは、同じ拡張機能を完全なメタデータとともにインストールします。すべてのメタデータプロパティを指定する必要はありません。しかし name、format、および source を含める必要があります。

```
var fs = require('fs');
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.config.resources.write({
  name: 'js-example',
  format: 'javascript',
  source: fs.createReadStream('./js-example.sjs'),
  // everything below this is optional metadata
  title: 'Example JavaScript Extension',
  description: 'An example of implementing resource
extensions in SJS',
  provider: 'MarkLogic',
  version: 1.0
}).result(function(response) {
  console.log('Installed extension: ' + response.name);
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

完全な例については、「例：リソースサービス拡張機能のインストールと使用」（240 ページ）を参照してください。

7.2.4 リソースサービス拡張機能の使用

リソースサービス拡張の HTTP メソッドを呼び出すには、DatabaseClient.resources インターフェイスを使用します。このインターフェイスには、各 HTTP 動詞（get、put、post、および remove（DELETE））に対応する関数が用意されています。例えば、次のようにパラメータなしで拡張機能の get メソッドを呼び出すことができます。

```
db.resources.get('js-example')
```

また、実装が想定するパラメータをカプセル化したオブジェクトや、トランザクション ID を渡すこともできます。

呼び出しの結果はメソッドによって異なります。例えば、GET 拡張機能インターフェイスにより 1 つあるいは複数のドキュメントを返すことができます。このため、`resources.get` を呼び出した結果は、応答内のドキュメントを段階的に処理できるストリーム関数を持つオブジェクトになります。

次の例は、『REST Application Developer's Guide』の「Example: JavaScript Resource Service Extension」からリソースサービス拡張機能の `get` 関数を呼び出します。「a」、「b」、「c」の 3 つのパラメータがこの実装に渡されます。この拡張機能の GET 実装は、指定したパラメータを JSON ドキュメントとしてエコーバックし、パラメータごとに 1 つのドキュメントを返します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.resources.get({
  name: 'js-example',
  params: { a: 1, b: 2, c: 'three'}
}).result(function(response) {
  console.log(response);
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

呼び出しが成功した場合、出力は次のようになります。各配列項目内の `content` プロパティの値は、拡張機能の GET メソッド実装によって返されたドキュメントです。

```
[ { contentType: 'application/json',
    format: 'json',
    contentLength: '29',
    content: { name: 'c', value: 'three' } },
  { contentType: 'application/json',
    format: 'json',
    contentLength: '25',
    content: { name: 'b', value: '2' } },
  { contentType: 'application/json',
    format: 'json',
    contentLength: '25',
    content: { name: 'a', value: '1' } } ]
```

詳細については、「例：リソースサービス拡張機能のインストールと使用」（240 ページ）を参照してください。

7.2.5 例：リソースサービス拡張機能のインストールと使用

この例では、『REST Application Developer's Guide』の「Example: JavaScript Resource Service Extension」で説明されている JavaScript 拡張機能の GET および PUT メソッドをインストールおよび実行する方法を示します。この拡張機能は、あらゆる MarkLogic クライアント API（Node.js、Java、REST）で使用できます。

次の手順に従って拡張機能をインストールし、GET メソッドを実行します。この拡張機能の GET メソッドは、1 つあるいは複数の呼び出し元が定義したパラメータを受け付けて、渡された各パラメータに対して { "name": *param-name*, "value": *param-value* } という形式の JSON ドキュメントを返します。

1. 拡張機能の実装を、`js-example.sjs` という名前のファイルにコピーします。この実装については、『REST Application Developer's Guide』の「JavaScript Extension Implementation」を参照してください。

2. 次のスクリプトを実行して、拡張機能を「js-example」という名前でインストールします。拡張機能をインストールするには、`rest-admin` ロールまたは同等の権限を持っている必要があります。詳細については、「リソースサービス拡張機能のインストール」（237 ページ）を参照してください。

```
var fs = require('fs');
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.config.resources.write({
  name: 'js-example',
  format: 'javascript',
  source: fs.createReadStream('./js-example.sjs'),
  // everything below this is optional metadata
  title: 'Example JavaScript Extension',
  description: 'An example of implementing resource
extensions in SJS',
  provider: 'MarkLogic',
  version: 1.0
}).result(function(response) {
  console.log('Installed extension: ' + response.name);
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

3. 必要に応じて、次のスクリプトを使用して拡張機能に関するメタデータを取得します。詳細については、「リソースサービス拡張機能の検出」（244 ページ）を参照してください。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.config.resources.read('js-example').result(
  function(response) {
    console.log(response);
  },
  function(error) {
    console.log(JSON.stringify(error, null, 2));
  });
```

4. 次のスクリプトを実行して、拡張機能の GET メソッドを実行します。詳細については、「リソースサービス拡張機能の使用」（238 ページ）を参照してください。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.resources.get({
  name: 'js-example',
  params: { a: 1, b: 2, c: 'three'}
}).result(function(response) {
  console.log(response);
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

GET メソッドは、渡された各パラメータに対して { name: *pName*, value: *pValue*} という形式の JSON ドキュメントを生成します。したがって、上記の呼び出しにより 3 つのドキュメントが生成されます。GET メソッドを呼び出して得られる出力は、次のようになります。

```
[ { contentType: 'application/json',
    format: 'json',
    contentLength: '29',
    content: { name: 'c', value: 'three' } },
  { contentType: 'application/json',
    format: 'json',
    contentLength: '25',
    content: { name: 'b', value: '2' } },
  { contentType: 'application/json',
    format: 'json',
    contentLength: '25',
    content: { name: 'a', value: '1' } } ]
```

5. 次のスクリプトを実行して、拡張機能の PUT メソッドを実行します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.resources.put({
  name: 'js-example',
  params: {
    basename: ['one', 'two']},
  documents: [
    { contentType: 'application/json',
    content: {key1:'value1'} },
    { contentType: 'application/json',
    content: {key2:'value2'} },
  ]
}).result(function(response) {
  console.log(JSON.stringify(response, null, 2));
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

この拡張機能の PUT メソッドは、入力として JSON および XML ドキュメントを受け付けます。入力する各 JSON ドキュメントについて、`written` プロパティがドキュメントに追加されてから、そのドキュメントがデータベースに挿入されます。XML ドキュメントは、変更なしでデータベースに挿入されます。ドキュメント URI は、呼び出し元によって提供された「basename」パラメータから生成されます。PUT メソッドを呼び出して得られる出力は、次のとおりです。

```
{ "written": [
    "/extensions/one.json",
    "/extensions/two.json"
] }
```

PUT の実行によって作成された 2 つのドキュメントを調べてみると、written プロパティが追加されていることがわかります。例えば、/extensions/one.json のコンテンツは次のようになっています（タイムスタンプの値は実際とは異なります）。

```
{
  "key1": "value1",
  "written": "08:35:54-08:00"
}
```

実装からクライアントにエラーを報告するには、「拡張機能および変換機能でのエラーの報告」（262 ページ）で説明されている規則に従う必要があります。例えば、各入力ドキュメントの basename パラメータ値を渡さないと、拡張機能は次のようにエラーを報告します。

```
if (docs.count > basenames.length) {
  returnErrToClient(400, 'Bad Request',
    'Insufficient number of uri basenames.Expected ' +
    docs.count + ' got ' + basenames.length + '.');
    // unreachable - control does not return from fn.error
}
```

エラーは、次の形式でクライアントアプリケーションに渡されます。

```
{
  "message": "js-example: response with invalid 400
status",
  "statusCode": 400,
  "body": {
    "errorResponse": {
      "statusCode": 400,
      "status": "Bad Request",
      "messageCode": "RESTAPI-SRVEXERR",
      "message": "Insufficient number of uri
basenames.Expected 2 got 1."
    }
  }
}
```

7.2.6 リソースサービス拡張機能の実装の取得

リソースサービス拡張機能の実装を取得するには、`DatabaseClient.config.resources.read` を使用します。このインターフェイスを使用するには、`rest-admin` ロールまたは同等の権限を持っている必要があります。

例えば次の呼び出しは、「js-example」としてインストールされたリソースサービス拡張機能の実装を取得します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.config.resources.read('js-example').result(
  function(response) {
    console.log(response);
  },
  function(error) {
    console.log(JSON.stringify(error, null, 2));
  });
```

7.2.7 リソースサービス拡張機能の検出

`DatabaseClient.config.resources.list` を使用すると、インストールされているリソース拡張機能について、名前やインターフェイス、その他のメタデータを取得できます。このインターフェイスを使用するには、`rest-reader` ロールまたは同等の権限を持っている必要があります。

特定の拡張機能に関する取得可能な情報の量は、拡張機能のインストール時に提供されたメタデータの量によって決まります。名前とメソッドは常に取得できます。プロバイダ、バージョン、およびメソッドパラメータ情報などの詳細はオプションです。

デフォルトでは、このリクエストを呼び出すたびに拡張機能のメタデータが再構築されます。これにより、メタデータは必ず最新の状態になります。

次の例は、インストールされている拡張機能に関するデータを取得します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.config.resources.list().result(
function(response) {
  console.log('Installed extensions: ');
  console.log(JSON.stringify(response, null, 2));
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

「リソースサービス拡張機能のインストール」（237 ページ）で示しているように、「js-example」という名前の 1 つの拡張機能をメタデータとともにインストールした場合、上記のスクリプトの出力は次のようになります。

```
{ "resources": {
    "resource": [ {
      "name": "js-example",
      "source-format": "javascript",
      "provider-name": "MarkLogic",
      "title": "Example JavaScript Extension",
      "version": "1",
      "description": "An example of implementing resource
extensions in SJS",
      "methods": {
        "method": [
          { "method-name": "get" },
          { "method-name": "post" },
          { "method-name": "put" },
```

```
            { "method-name": "delete" }
          ]
        },
        "resource-source": "/v1/resources/js-example"
      }
    ] }
  }
```

7.2.8 リソースサービス拡張機能の削除

リソースサービス拡張機能を削除するには、`DatabaseClient.config.resources.remove` を使用します。拡張機能のインストールに使用したのと同じ名前を指定する必要があります。拡張機能を削除するには、`rest-admin` ロールまたは同等の権限を持っている必要があります。

拡張機能の削除は、冪等（べきとう）操作（何回繰り返しても、1 回だけ実行したときと同じ結果になる操作）です。つまり、指定した拡張機能が存在するかどうかにかかわらず同じ応答を受け取ります。

次のコードスニペットは、「js-example」という名前のリソースサービス拡張機能を削除します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.config.resources.remove('js-example').result(
  function(response) {
    console.log('Removed extension: ', response.name);
  },
  function(error) {
    console.log(JSON.stringify(error, null, 2));
  });
```

7.3 コンテンツ変換機能を扱う

このセクションでは、コンテンツ変換機能の概念と、変換機能の作成、インストール、適用、および管理方法について説明します。ここでは、次の内容を取り上げます。

- コンテンツ変換機能とは
- 変換機能の作成
- 変換機能のインストール
- 変換機能の使用
- 例：読み取り、書き込み、およびクエリの変換機能

7.3.1 コンテンツ変換機能とは

Node.js クライアント API では、カスタムのコンテンツ変換機能を作成して、ドキュメントの読み込みや取得などの操作時に適用できます。例えば、各ドキュメントをデータベースに挿入する際に、そのドキュメントの JSON プロパティまたは XML 要素を追加または変更する書き込み変換機能を作成できます。API には、次の種類の変換を適用するためのフックが用意されています。

- 書き込み変換：ドキュメントをデータベースに挿入する前に適用されます。
- 読み取り変換：データベースからドキュメントを読み取るときに適用されます。デフォルトの読み取り変換とリクエストごとの読み取り変換の両方を設定できます。
- 検索結果の変換：マッチするドキュメントとメタデータを返すだけではなく、サマリを含むクエリを作成したときに検索結果のサマリに適用できます。

変換は、サーバーサイド JavaScript 関数、XQuery 関数、または XSLT スタイルシート（入力としてドキュメントを受け付け、出力としてドキュメントを生成するもの）として実装されます。また、『REST Application Developer's Guide』の「Writing Transformations」で説明しているインターフェイスおよびガイドラインに準拠している必要があります。変換では、変換固有のパラメータを利用できます。

変換を使用するには、その変換を REST API インスタンスに関連付けられた modules データベースにインストールする必要があります。Node.js を使用して変換をインストールおよび管理するには、`DatabaseClient.config.transforms` インターフェイスを使用します。詳細については、「変換機能のインストール」（247 ページ）を参照してください。

変換を適用するには、`DatabaseClient.documents.write`、`DatabaseClient.documents.read`、および `DatabaseClient.documents.query` などのサポートされている操作に目的の変換名を渡します。詳細については、「変換機能の使用」（248 ページ）を参照してください。

7.3.2 変換機能の作成

変換関数は、サーバーサイド JavaScript または XQuery を使用して実装できます。インターフェイスは複数の MarkLogic クライアント API 間で共有されるため、Java クライアント API、Node.js クライアント API、および REST クライアント API で同じ変換機能を使用できます。

変換モジュールには、`transform` という名前のエクスポートが含まれている必要があります。次に例を示します。

```
function insertTimestamp(context, params, content)
{...}
exports.transform = insertTimestamp;
```

あるクライアント API で変換機能をインストールすれば、すべての API で使用できるようになります。例えば、Node.js クライアント API を使用してインストールした変換機能は、Node.js クライアント API または Java クライアント API を使用して実装したアプリケーションで使用できます。

インターフェイスの定義、オーサリングのガイドライン、および例の実装については、『REST Application Developer's Guide』の「Writing Transformations」を参照してください。変換機能からクライアントにエラーを返すには、「拡張機能および変換機能でのエラーの報告」（262 ページ）で説明されている規則を使用します。

Node.js および JavaScript の完全な例については、「例：読み取り、書き込み、およびクエリの変換機能」（250 ページ）を参照してください。

7.3.3 変換機能のインストール

REST API インスタンスに関連付けられている modules データベースに変換機能をインストールするには、`DatabaseClient.config.transforms.write` を使用します。このインターフェイスを使用することで、API が想定する規則に従って変換機能がインストールされるため、Node.js クライアント API を使って後から変換機能を適用および管理できます。

変換機能をインストールするには、`rest-admin` ロールまたは同等の権限を持っている必要があります。

インストール時に変換機能に関するオプションのメタデータを含めることができます。ここでメタデータは、単に情報として利用されています。これは、`DatabaseClient.config.transforms.list` を使用して取得できます。

次のスクリプトは、「js-transform」という変換機能をインストールします。変換機能の実装は、「transform.sjs」というファイルから読み取られます。必須なのは `name`、`format`、および `source` だけで、その他のメタデータはすべてオプションです。

```
var fs = require('fs');
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.config.transforms.write({
  name: 'js-transform',
  format: 'javascript',
  source: fs.createReadStream('./transform.sjs'),
```

```
    // everything below this is optional metadata
    title: 'Example JavaScript Transform',
    description: 'An example of an SJS read/write transform',
    provider: 'MarkLogic',
    version: 1.0
  }).result(function(response) {
    console.log('Installed transform: ' + response.name);
  }, function(error) {
    console.log(JSON.stringify(error, null, 2));
  });
```

完全な例については、「例：読み取り、書き込み、およびクエリの変換機能」（250 ページ）を参照してください。

インストールされている変換機能のリストとそのメタデータは、`DatabaseClient.transforms.list` を使用して取得できます。詳細については、「インストールされている変換機能の検出」（261 ページ）を参照してください。

完全な例については、「例：読み取り、書き込み、およびクエリの変換機能」（250 ページ）を参照してください。

7.3.4 変換機能の使用

ドキュメントの読み取り、書き込み、クエリの各操作に対する変換は次のように指定できます。

- `DatabaseClient.documents.read`
- `DatabaseClient.documents.write` および `DatabaseClient.documents.createWriteStream`
- `DatabaseClient.documents.query`、`queryBuilder.slice` を使用
- `DatabaseClient.values.read`、`valuesBuilder.slice` を使用

どの場合も、`DatabaseClient.config.transforms.write` またはその他のクライアント API の同等の操作を使用して、以前にインストールした変換機能の名前を指定します。

完全な例については、「例：読み取り、書き込み、およびクエリの変換機能」（250 ページ）を参照してください。

変換は、操作ごとに 1 つだけ指定できます。変換は、すべての入力（`write`）または出力（`read` または `query`）に適用されます。

documents.read または documents.write に対して変換機能を指定するには、次のいずれかの形式を使用して呼び出しオブジェクトに transform プロパティを追加します。最初の 2 つの形式は同等です。3 番目の形式を使用して、変換機能が想定するパラメータを渡します。

```
transform: transformName

transform: [ transformName ]

transform: [ transformName, {paramName: paramValue, ...}]
```

読み取りと書き込みの場合は、入力呼び出しオブジェクトの直接の子として transform プロパティを含めます。例えば、1 つのドキュメントディスクリプタを documents.write に渡す場合は、変換機能をディスクリプタに含めることができます。

```
db.documents.write({
  uri: '/doc/example.json',
  contentType: 'application/json',
  content: { some: 'data' },
  transform: ['js-write-transform']
})
```

それに対して、documents.write に複数ドキュメントの入力形式を使用する場合は、変換ディスクリプタを各ドキュメントディスクリプタの内部ではなく、最上位レベルのオブジェクトに含めます。次に例を示します。

```
db.documents.write({
  documents: [
    { uri: '/transforms/example1.json',
      contentType: 'application/json',
      content: { some: 'data' },
    },
    { uri: '/transforms/example2.json',
      contentType: 'application/json',
      content: { some: 'more data' },
    }
  ],
  transform: ['js-write-transform']
})
```

DatabaseClient.documents.query または DatabaseClient.values.read からの結果に変換機能を適用するには、transform ビルダーを使用してディスクリプタを作成し、slice 節を通じて、そのディスクリプタをクエリにアタッチします。例えば、次の呼び出しは、変換機能をコンテンツクエリに適用します。

```
db.documents.query(
  qb.where(
    qb.byExample({writeTimestamp: {'$exists': {}}})
  ).slice(qb.transform('js-query-transform', {a: 1, b:
'two'}))
)
```

次の呼び出しは、同じ変換機能（追加のパラメータなし）を値クエリに適用します。

```
db.values.read(
  vb.fromIndexes('reputation')
    .slice(3,5, vb.transform('js-query-transform'))
)
```

7.3.5 例：読み取り、書き込み、およびクエリの変換機能

この例では、変換機能をインストールして、読み取り操作、書き込み操作、およびクエリ操作で使用する方法を示します。この例は、次の 3 種類の変換を順番に実行できるように設計されています。

1. 変換機能のインストール
2. 書き込み変換機能の使用
3. 読み取り変換機能の使用
4. クエリ変換機能の使用

各変換機能のソースコードは、次のセクションに示しています。

- 読み取り変換機能のソースコード
- 書き込み変換機能のソースコード
- クエリ変換機能のソースコード

7.3.5.1 変換機能のインストール

この手順に従って、読み取り、書き込み、およびクエリの変換機能の例をインストールします。詳細については、「変換機能のインストール」（247 ページ）を参照してください。

1. 「読み取り変換機能のソースコード」（258 ページ）で説明しているコードから、`read-transform.sjs` という名前のファイルを作成します。
2. 「書き込み変換機能のソースコード」（259 ページ）で説明しているコードから、`write-transform.sjs` という名前のファイルを作成します。

3. 「クエリ変換機能のソースコード」（260 ページ）で説明しているコードから、query-transform.sjs という名前のファイルを作成します。

4. 次のコードを install-transform.js という名前のファイルにコピーします。このスクリプトは、3 つの変換機能すべてをインストールします。

```
var fs = require('fs');
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

// Descriptors for 3 transforms: Read, write, and query.
var transforms = [
  {
    name: 'js-read-transform',
    format: 'javascript',
    source: fs.createReadStream('./read-transform.sjs'),
      // everything below this is optional metadata
    title: 'Example JavaScript Read Transform',
    description: 'An example of an SJS read transform',
    provider: 'MarkLogic',
    version: 1.0
  },
  { name: 'js-write-transform',
    format: 'javascript',
    source: fs.createReadStream('./write-transform.sjs')
  },
  {
    name: 'js-query-transform',
    format: 'javascript',
    source: fs.createReadStream('./query-transform.sjs')
  }
];

// Install the transforms
transforms.forEach( function installTransform(transform) {
  db.config.transforms.write(transform).result(
    function(response) {
      console.log('Installed transform: ' + response.name);
    },
    function(error) {
      console.log(JSON.stringify(error, null, 2));
    }
  );
});
```

インストールが正常に完了すると、次のような結果が表示されます。

```
$ node install-transform.js
Installed transform: js-write-transform
Installed transform: js-query-transform
Installed transform: js-read-transform
```

7.3.5.2 書き込み変換機能の使用

次のスクリプトは、書き込み変換機能を使用してデータベースにドキュメントを書き込みます。このスクリプトは、「変換機能の使用」（248 ページ）で説明している使用ガイドラインに従っています。

「変換機能のインストール」（250 ページ）の指示に従って、変換機能のインストールが完了している必要があります。

この例の変換機能は `writeTimestamp` を入力ドキュメントに追加します。詳細については、「書き込み変換機能のソースコード」（259 ページ）を参照してください。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');

var db = marklogic.createDatabaseClient(my.connInfo);

db.documents.write({
  documents: [
    { uri: '/transforms/example1.json',
      contentType: 'application/json',
      content: { some: 'data' },
    },
    { uri: '/transforms/example2.json',
      contentType: 'application/json',
      content: { some: 'more data' },
    }
  ],
  transform: ['js-write-transform']
}).result(function(response) {
  response.documents.forEach(function(document) {
    console.log(document.uri);
  });
}, function(error) {
  console.log(JSON.stringify(error));
});
```

上記のスクリプトを実行すると、次のような出力が表示されます。

```
$ node write.js
/transforms/example1.json
/transforms/example2.json
```

Query Console を使用してデータベース内のドキュメントを調べると、writeTimestamp プロパティがそれぞれのコンテンツに追加されていることがわかります。例えば、/transform/example1.json には次のようなコンテンツが含まれています。

```
{
  "some": "data",
  "writeTimestamp": "2015-01-02T10:33:39.330483-08:00"
}
```

変換機能は、書き込み操作のすべてのドキュメントに適用されます。ドキュメントごとに異なる変換機能を指定することはできません。ドキュメントを 1 つだけ挿入する場合は、documents 配列を作成する代わりに、単一のドキュメントディスクリプタに変換機能を含めることができます。次に例を示します。

```
db.documents.write({
  uri: '/transforms/example3.json',
  contentType: 'application/json',
  content: { some: 'even more data' },
  transform: ['js-write-transform']
})...
```

変換機能には、パラメータを渡すことができます。例については、「読み取り変換機能の使用」（253 ページ）を参照してください。

7.3.5.3 読み取り変換機能の使用

このセクションでは、「変換機能の使用」（248 ページ）の使用ガイドラインに従って読み取り変換機能を適用する方法を示します。

このスクリプトを実行する前に、例の変換機能をインストールし、書き込み変換機能の例を実行しておく必要があります。詳細については、「変換機能のインストール」（250 ページ）および「書き込み変換機能の使用」（252 ページ）を参照してください。

このスクリプトは、「書き込み変換機能の使用」（252 ページ）の手順で挿入したドキュメントを読み取ります。読み取り変換機能が各ドキュメントに適用されます。この変換機能は、返されたドキュメントに readTimestamp プロパティを追加します。変換機能は、プロパティ名 / 値のペアを入力パラメータとして受け付けることで、出力へのプロパティの追加もサポートします。この例では、変換ディスクリプタで次のパラメータを指定して、2 つのプロパティ extra1 および extra2 を出力ドキュメントに追加します。

```
transform: ['js-read-transform', {extra1: 1, extra2: 'two'}]
```

次のスクリプトをファイルにコピーし、node コマンドで実行して読み取り変換機能の例を実行します。変換機能の実装をレビューする方法については、「読み取り変換機能のソースコード」（258 ページ）を参照してください。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');

var db = marklogic.createDatabaseClient(my.connInfo);

db.documents.read({
  uris: ['/transforms/example1.json',
'/transforms/example2.json'],
  transform: ['js-read-transform', {extra1: 1, extra2:
'two'}]
}).stream().on('data', function(document) {
  console.log("URI: " + document.uri);
  console.log(JSON.stringify(document.content, null, 2) +
'\n');
}).on('end', function() {
  console.log('Finished');
})
```

スクリプトを実行すると、次のような出力が表示されます。

```
URI: /transforms/example1.json
{
  "some": "data",
  "writeTimestamp": "2015-01-02T10:33:39.330483-08:00",
  "readTimestamp": "2015-01-02T10:44:18.538343-08:00",
  "extra2": "two",
  "extra1": "1"
}

URI: /transforms/example2.json
{
```

```
    "some": "more data",
    "writeTimestamp": "2015-01-02T10:33:39.355913-08:00",
    "readTimestamp": "2015-01-02T10:44:18.5632-08:00",
    "extra2": "two",
    "extra1": "1"
}
```

readTimestamp、extra1、および extra2 の各プロパティが読み取り変換機能によって追加されます。これらのプロパティは、読み取り出力の一部です。データベース内のドキュメントは変更されません。読み込み時の書き込み変換機能により、writeTimestamp プロパティがドキュメントに追加されています。詳細については、「書き込み変換機能の使用」（252 ページ）を参照してください。

7.3.5.4 クエリ変換機能の使用

次のスクリプトは、「変換機能の使用」（248 ページ）の使用ガイドラインに従って変換機能をクエリ操作に適用します。

このスクリプトを実行する前に、変換機能の例をインストールし、書き込み変換機能の例を実行しておく必要があります。詳細については、「変換機能のインストール」（250 ページ）および「書き込み変換機能の使用」（252 ページ）を参照してください。

以下のスクリプトは、QBE を使用して JSON プロパティ writeTimestamp を持つすべてのドキュメントを読み取ります。このプロパティは、「書き込み変換機能の使用」（252 ページ）の手順で、書き込み変換機能によっていくつかのドキュメントに追加されています。

このスクリプトは、マッチするドキュメントを返すクエリと、検索結果のサマリのみを返すクエリを作成します。この変換機能は、ドキュメントを取得するときに、「読み取り変換機能のソースコード」（258 ページ）で説明している読み取り変換機能と全く同じように動作します。つまり、各ドキュメントに readTimestamp プロパティを追加し、必要に応じて各入力パラメータに対応するプロパティを追加します。検索結果のサマリを JSON として取得すると、サマリに queryTimestamp プロパティが追加されます。

クエリ時に変換機能を適用する方法を示すには、次のスクリプトをファイルにコピーし、node コマンドを使用して実行します。変換機能の実装をレビューする方法については、「クエリ変換機能のソースコード」（260 ページ）を参照してください。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');

var db = marklogic.createDatabaseClient(my.connInfo);
var qb = marklogic.queryBuilder;
```

```
// Retrieve just search result summary by setting slice to
0
db.documents.query(
  qb.where(
    qb.byExample({writeTimestamp: {'$exists': {}}})
  ).slice(qb.transform('js-query-transform'))
   .withOptions({categories: 'none'})
).result(function(response) {
  console.log(JSON.stringify(response, null, 2));
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});

// Retrieve matching documents instead of summary
db.documents.query(
  qb.where(
    qb.byExample({writeTimestamp: {'$exists': {}}})
  ).slice(qb.transform('js-query-transform', {a: 1, b:
'two'}))
).stream().on('data', function(document) {
  console.log("URI: " + document.uri);
  console.log(JSON.stringify(document.content, null, 2) +
'\n');
}).on('end', function() {
  console.log('Finished');
});
```

スクリプトを実行すると、次のような出力が表示されます。太字のプロパティは、この変換機能によって追加されたものです。

```
$ node query.js
URI: /transforms/example1.json
{
  "some": "data",
  "writeTimestamp": "2015-01-02T10:33:39.330483-08:00",
  "readTimestamp": "2015-01-02T11:09:58.410351-08:00",
  "b": "two",
  "a": "1"
}

URI: /transforms/example2.json
{
  "some": "more data",
  "writeTimestamp": "2015-01-02T10:33:39.355913-08:00",
  "readTimestamp": "2015-01-02T11:09:58.431676-08:00",
  "b": "two",
```

```
  "a": "1"
}

Finished
[
  {
    "snippet-format": "snippet",
    "total": 2,
    "start": 1,
    "page-length": 0,
    "results": [],
    "metrics": {
      "query-resolution-time": "PT0.001552S",
      "facet-resolution-time": "PT0.000141S",
      "snippet-resolution-time": "PT0S",
      "total-time": "PT0.165583S"
    },
    "queryTimestamp": "2015-01-02T11:10:00.208708-08:00"
  }
]
```

変換の指定内容は、結果を詳細に設定する `slice` 節の一部であることと、またビルダー（`qb.transform`）が変換の指定内容の作成に使用されることに注意してください。次に例を示します。

```
slice(qb.transform('js-query-transform', {a: 1, b: 'two'}))
```

変換の指定内容のシンタックスは、`documents.read` と `documents.write` ではクエリコンテキストが異なります。このため、ビルダーを使用するのが最適な方法です。

クエリ操作では、マッチしたドキュメントや結果サマリといったすべての出力に対して変換機能が呼び出されます。Node.js クライアント API でクエリを実行した場合、検索結果のサマリは常に JSON になります。そのため、プロパティを調べるだけでマッチするドキュメントと区別できます。例えば、クエリ変換は、次の処理を実行して検索サマリを識別します。

```
if (result.hasOwnProperty('snippet-format')) {
  // search result summary
  result.queryTimestamp = fn.currentDateTime();
}
```

Java クライアント API のような別のクライアント API の代わりに変換機能が呼び出されている場合、結果のサマリは XML になります。またこのクエリではドキュメントと結果のサマリの両方を取得できます。

7.3.5.5 読み取り変換機能のソースコード

次のサーバーサイド JavaScript モジュールは、DatabaseClient.documents.read などの読み取り操作で変換機能として使用することを目的としたものです。この変換機能は、JSON ドキュメントの読み取り時に出力ドキュメントにプロパティを追加します。詳細については、コードのコメントを参照してください。

次のコードを read-transform.sjs という名前のファイルにコピーします。別のファイル名を使用することもできますが、このセクションで示すインストールスクリプトは、この名前を前提として記述されています。

```
// Example Read Transform
//
// If the input is a JSON document:
// - Add a readTimestamp to the result document.
// - For each parameter passed in by the client, add a
//   property of the form: propName: propValue.
// Other document types are unchanged.
function readTimestamp(context, params, content)
{
  //if (context.inputType.search('json') >= 0) {
  if (context.inputType.search('json') >= 0) {
    var result = content.toObject();

    result.readTimestamp = fn.currentDateTime();

    // Add a property for each caller-supplied request
param
    for (var pname in params) {
      if (params.hasOwnProperty(pname)) {
        result[pname] = params[pname];
      }
    }
    return result;
  } else {
    // Pass thru for non-JSON documents
    return content;
  }
};

exports.transform = readTimestamp;
```

7.3.5.6 書き込み変換機能のソースコード

次のサーバーサイド JavaScript モジュールは、`DatabaseClient.documents.write` などの書き込み操作で変換機能として使用することを目的としたものです。この変換機能は、読み込むあらゆる JSON ドキュメントにプロパティを追加します。詳細については、コードのコメントを参照してください。

次のコードを `write-transform.sjs` という名前のファイルにコピーします。別のファイル名を使用することもできますが、このセクションで示すインストールスクリプトは、この名前を前提として記述されています。

```
// Example Write Transform
//
// If the input is a JSON document:
// - Add a writeTimestamp to the document.
// - For each parameter passed in by the client, add a
property
//   of the form "propName: propValue" to the document.
// Non-JSON documents are returned unmodified.
function writeTimestamp(context, params, content)
{
  if (context.inputType.search('json') >= 0) {
    var result = content.toObject();
    result.writeTimestamp = fn.currentDateTime();

    // Add a property for each caller-supplied request
param
    for (var pname in params) {
      if (params.hasOwnProperty(pname)) {
        result[pname] = params[pname];
      }
    }
    return result;
  } else {
    // Pass thru for non-JSON documents
    return content;
  }
};

exports.transform = writeTimestamp;
```

7.3.5.7 クエリ変換機能のソースコード

次のサーバーサイド JavaScript モジュールは、DatabaseClient.documents.query などのクエリ操作で変換機能として使用することを目的としたものです。この目的で、「読み取り変換機能のソースコード」（258 ページ）で説明している読み取り変換機能を使用することもできます。ただしクエリコンテキストでは、変換機能はマッチするドキュメントと生成された検索サマリの両方に適用されます。ここでは説明上、この変換機能は 2 つのケースを区別します。

この変換機能は JSON 出力にプロパティを追加しますが、検索結果のサマリとマッチするドキュメントを区別し、snippet-format プロパティを含む JSON 入力コードが検索サマリで、その他の JSON 入力がクエリにマッチするドキュメントであると想定して処理を行います。

次のコードを query-transform.sjs という名前のファイルにコピーします。別のファイル名を使用することもできますが、このセクションで示すインストールスクリプトは、この名前を前提として記述されています。

```
// When applied to a query operation, a transform is
invoked on
// both the search result summary and the matching
documents (when
// used as a multi-document read).
//
// The transform does the following:
// - For a JSON search result summary (determined by the
presence
//   of a search-snippet property), add a queryTimestamp
property.
// - For a JSON document, add a readTimestamp property.
// - For all other input, pass it through unchanged.
function queryTimestamp(context, params, content)
{
  if (context.inputType.search('json') >= 0) {
    var result = content.toObject();
    if (result.hasOwnProperty('snippet-format')) {
      // search result summary
      result.queryTimestamp = fn.currentDateTime();
    } else {
      // JSON document.Add readTimestamp property plus a
property
      // for each param passed in by the client.
      result.readTimestamp = fn.currentDateTime();
      for (var pname in params) {
        if (params.hasOwnProperty(pname)) {
          result[pname] = params[pname];
```

```
        }
      }
    }
    return result;
  } else {
    // Pass thru for non-JSON documents or XML search
summary
    return content;
  }
};

exports.transform = queryTimestamp;
```

7.3.6 インストールされている変換機能の検出

インストールされている変換機能の名前とメタデータは、`DatabaseClient.transforms.list` を使用して取得できます。インストールされている変換機能のリストを取得するには、`read-reader` ロールまたは同等の権限を持っている必要があります。

次の例は、インストールされている変換機能のリストを取得し、コンソールに応答を表示します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.config.transforms.list().result(
function(response) {
  console.log('Installed transforms: ');
  console.log(JSON.stringify(response, null, 2));
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

「変換機能のインストール」（247 ページ）の説明に従って変換機能をインストールした場合は、上記のスクリプトを実行すると次のような出力が生成されます。

```
{ "transforms": {
    "transform": [ {
        "name": "js-transform",
        "source-format": "javascript",
        "title": "Example JavaScript Transform",
        "version": "1",
        "provider-name": "MarkLogic",
        "description": "An example of an SJS read/write
```

```
transform",
        "transform-parameters": "",
        "transform-source":
"/v1/config/transforms/js-transform"
    } ]
} }
```

その他の例については、Node.js クライアント API GitHub プロジェクトの `test-basic/documents-transform.js` を参照してください。

7.3.7 変換機能の削除

MarkLogic サーバーから変換機能をアンインストールするには、`DatabaseClient.transforms.remove` を使用します。アンインストール操作は、冪等（べきとう）操作（何回繰り返しても、1 回だけ実行したときと同じ結果になる操作）です。つまり、指定した変換機能がインストールされているかどうかにかかわらず同じ結果になります。

変換機能をアンインストールするには、`rest-admin` ロールまたは同等の権限を持っている必要があります。

次のスクリプトは、「js-transform」という名前の変換機能をアンインストールします。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.config.transforms.remove('js-transform').result(
  function(response) {
    console.log('Removed transform: ', response.name);
  },
  function(error) {
    console.log(JSON.stringify(error, null, 2));
  });
```

7.4 拡張機能および変換機能でのエラーの報告

拡張機能および変換機能は、同じメカニズムを使用して呼び出し元のアプリケーションにエラーを報告します。

`RESTAPI-SRVEXERR` を報告し、data パラメータで追加情報を提供するには、`fn.error`（JavaScript）または `fn:error`（XQuery）を使用します。応答ステータスコードとステータスメッセージを制御したり、追加のエラー報告の応答ペイロードを提供したりできます。

別の方法でエラーを報告すると、クライアントアプリケーションには 500 Internal Server Error として返されます。

例については、次のトピックを参照してください。

- 例：JavaScript でのエラーの報告
- 例：XQuery でのエラーの報告

7.4.1 例：JavaScript でのエラーの報告

JavaScript 拡張機能または変換機能からクライアントアプリケーションにエラーを返すには、`fn.error` を使用して `RESTAPI-SRVEXERR` エラーを報告し、`fn.error` の `data` パラメータで追加情報を提供します。応答ステータスコードとステータスメッセージを制御したり、追加のエラー報告の応答ペイロードを提供したりできます。例えば、次の方法でクライアントにエラーを返すことができます。

```
fn.error(null, 'RESTAPI-SRVEXERR',
  Sequence.from([400, 'Bad Request',
                 'Insufficient number of uri
basenames.']));
  // unreachable - control does not return from fn.error
```

`fn:error` の 3 番目のパラメータは、`(`*`status-code`*`, '`*`status-message`*`', '`*`payload-format`*`', '`*`response-payload`*`')` という形式のシーケンスでなければなりません。つまり、`fn.error` を使用して `RESTAPI-SRVEXERR` を報告する場合、`fn.error` の `data` パラメータは次の項目を含むシーケンスであり、すべてオプションです。

- HTTP ステータスコード。デフォルト：400。
- HTTP ステータスメッセージ。デフォルト：Bad Request。
- 応答ペイロード。ペイロードは、REST API インスタンスの設定に応じて JSON または XML になるため、テキストに制限しておくことをお勧めします。

  注： ベストプラクティスは、`RESTAPI-SRVEXERR` を使用する方法です。他のエラーを報告したり、他の例外を発生させたりした場合、呼び出し元アプリケーションには 500 Server Internal Error として報告されます。

`xdmp.arrayValues` または `Sequence.from` を使用して、JavaScript 配列からシーケンスを作成できます。

`fn.error` から制御は戻りません。呼び出す前に、必要なクリーンアップまたは他のタスクを実行しておく必要があります。

次のようなユーティリティ関数を使用して、拡張機能の実装において詳細の大半を抽象化できます。

```
function returnErrToClient(statusCode, statusMsg, body)
{
  fn.error(null, 'RESTAPI-SRVEXERR',
           Sequence.from([statusCode, statusMsg, body]));
  // unreachable
};
```

この関数の使用例を次に示します。

```
returnErrToClient(400, 'Bad Request',
  'Insufficient number of uri basenames.');
```

Node.js API を使用して次のように拡張機能の呼び出しからエラーをトラップすると、

```
db.resources.put({
  ...
}).result(function(response) {
  console.log(JSON.stringify(response, null, 2));
}, function(error) {
  console.log(JSON.stringify(error, null, 2));
});
```

出力は次のようになります。

```
{
  "message": "js-example: response with invalid 400
status",
  "statusCode": 400,
  "body": {
    "errorResponse": {
      "statusCode": 400,
      "status": "Bad Request",
      "messageCode": "RESTAPI-SRVEXERR",
      "message": "Insufficient number of uri basenames."
    }
  }
}
```

実際の例については、「例：リソースサービス拡張機能のインストールと使用」（240 ページ）を参照してください。

7.4.2 例：XQuery でのエラーの報告

RESTAPI-SRVEXERR エラーを報告し、fn:error の $data パラメータで追加情報を提供するには、fn:error を使用します。応答ステータスコードとステータスメッセージを制御したり、追加のエラー報告の応答ペイロードを提供したりできます。例えば、次の方法でクライアントにエラーを返すことができます。

```
fn:error((),"RESTAPI-SRVEXERR",
  (415, "Unsupported Input Type",
   "Only application/xml is supported"))
```

fn:error の 3 番目のパラメータは、("*status-code*", "*status-message*", "*response-payload*") という形式のシーケンスでなければなりません。つまり、fn:error を使用して RESTAPI-SRVEXERR を報告する場合、fn:error の $data パラメータは次のメンバーを含むシーケンスであり、すべてオプションです。

- HTTP ステータスコード。デフォルト：400。
- HTTP ステータスメッセージ。デフォルト：Bad Request。
- 応答ペイロード。ペイロードは、REST API インスタンスの設定に応じて JSON または XML になるため、テキストに制限しておくことをお勧めします。

注： ベストプラクティスは、RESTAPI-SRVEXERR を使用する方法です。他のエラーを報告したり、他の例外を発生させたりした場合、呼び出し元アプリケーションには 500 Server Internal Error として報告されます。

例えば、このリソース拡張関数は、入力コンテンツタイプが想定されたタイプでない場合に RESTAPI-SRVEXERR を報告します。

```
declare function example:put(
    $context as map:map,
    $params  as map:map,
    $input   as document-node()
) as document-node()
{
    (: get 'input-types' to use in content negotiation :)
    let $input-types := map:get($context,"input-types")
    let $negotiate :=
        if ($input-types = "application/xml")
        then () (: process, insert/update :)
        else fn:error((),"RESTAPI-SRVEXERR",
          ("415", "Raven", "nevermore"))
   return document { "Done"}  (: may return a document node
:)
};
```

想定されていないコンテンツタイプで拡張機能に対して PUT リクエストを行った場合、`fn:error` 呼び出しによって、そのリクエストはステータス 415 の失敗となり、応答ボディ内に追加のエラー説明が含まれるようになります。

```
HTTP/1.1 415 Raven
Content-type: application/xml
Server: MarkLogic
Set-Cookie: SessionID=714070bdf4076536; path=/
Content-Length: 62
Connection: close

<?xml version="1.0" encoding="UTF-8"?>
<word>nevermore</word>
```

7.5 アドホックコードとサーバーサイドモジュールの評価

`DatabaseClient.eval` または `DatabaseClient.xqueryEval` を使用すると、XQuery のアドホックブロックまたは MarkLogic サーバー上のサーバーサイド JavaScript コードを評価できます。コードブロックは、クライアントアプリケーションで生成されます。また、`DatabaseClient.invoke` を使用すると、以前にインストールした XQuery または MarkLogic サーバー上のサーバーサイド JavaScript モジュールを評価できます。

このセクションでは、評価および呼び出しの使用方法に関する次のトピックについて説明します。

- 必要な権限
- アドホッククエリの評価
- MarkLogic サーバー上にインストールされているモジュールの呼び出し
- Eval または Invoke の結果の解釈
- 外部変数値の指定

7.5.1 必要な権限

`DatabaseClient.eval`、`DatabaseClient.xqueryEval`、および `DatabaseClient.invoke` を使用するには、Node.js クライアント API を使用した通常の読み取り / 書き込み / クエリ操作で必要となる権限に加え、追加の権限が必要になります。

`DatabaseClient.eval` または `DatabaseClient.xqueryEval` を使用するには、少なくとも次の権限か同等の権限を持っている必要があります。

- `http://marklogic.com/xdmp/privileges/xdmp-eval`
- `http://marklogic.com/xdmp/privileges/xdmp-eval-in`

- http://marklogic.com/xdmp/privileges/xdbc-eval
- http://marklogic.com/xdmp/privileges/xdbc-eval-in

DatabaseClient.invoke を使用するには、少なくとも次の権限か同等の権限を持っている必要があります。

- http://marklogic.com/xdmp/privileges/xdmp-invoke
- http://marklogic.com/xdmp/privileges/xdmp-invoke-in
- http://marklogic.com/xdmp/privileges/xdbc-invoke
- http://marklogic.com/xdmp/privileges/xdbc-invoke-in

上記の権限は、サーバーサイドコードの評価 / 呼び出しを行うためだけの権限です。そのコードによって実行される操作では、追加の権限が必要になることがあります。

7.5.2 アドホッククエリの評価

MarkLogic サーバーで JavaScript のアドホックブロックを評価するには、DatabaseClient.eval を使用します。『JavaScript Reference Guide』で説明している MarkLogic のサーバーサイド JavaScript ダイアレクトを使用する必要があります。XQuery のアドホックブロックを評価するには、DatabaseClient.xqueryEval を使用します。呼び出し規則および応答規則は、eval と xqueryEval のどちらでも同じです。これらの操作は、xdmp.eval（JavaScript）または xdmp:eval（XQuery）ビルトイン関数を使用した場合と同じです。このコードは、DatabaseClient オブジェクトに関連付けられているデータベースのコンテキストで評価されます。

eval または xqueryEval を使用するには、追加のセキュリティ権限が必要です。詳細については、「必要な権限」（266 ページ）を参照してください。

次のいずれかの形式を使用して、eval および xqueryEval を呼び出すことができます。このコードは、必須パラメータ / プロパティに過ぎません。

```
db.eval(codeAsString, externalVarsObj)
db.eval({source: codeAsString, variables: externalVarsObj,
txid:...})

db.xqueryEval(codeAsString, externalVarsObj)
db.xqueryEval({
  source: codeAsString,
  variables: externalVarsObj,
  txid:...})
```

外部変数を使用すると、MarkLogic サーバーに変数値を渡すことができます。変数値はアドホックコードに代入されます。詳細については、「外部変数値の指定」（273 ページ）を参照してください。

例えば、次の JavaScript コードを評価するとします。`word1` と `word2` は、アプリケーションによって提供された外部変数値です。

```
word1 + " " + word2
```

次の呼び出しで、MarkLogic サーバー上のコードを評価します。`word1` と `word2` の値は、2 番目のパラメータによって MarkLogic サーバーに渡されます。

```
db.eval('word1 + " " + word2', {word1: 'hello', word2:
'world'})
```

eval を呼び出して得られる応答は、コードブロックによって返された値ごとに 1 つの項目が含まれている配列です。各項目には、戻り値と、値を解釈するのに役立つタイプ情報が含まれています。詳細については、「Eval または Invoke の結果の解釈」（272 ページ）を参照してください。

例えば、上記の呼び出しは、次の応答を返します。

```
[{
  "format": "text",
  "datatype": "string",
  "value": "hello world"
}]
```

ドキュメント、オブジェクト、配列に加え、アトミック値を返すことができます。複数の項目を返すには、`Sequence`（JavaScript のみ）またはシーケンスを返す必要があります。`Sequence` は配列などから作成するか、またはジェネレータから作成でき、多くのビルトイン関数が複数の値と、`Sequence` を返します。サーバーサイド JavaScript でシーケンスを作成するには、JavaScript 配列に `xdmp.arrayValues` または `Sequence.from` を適用します。

例えば、2 つの入力値の合計長と連結された文字列を返すように前述の例を拡張するには、配列内に結果を蓄積して、その配列に `Sequence.from` を適用します。

```
Sequence.from([word1.length + word2.length, word1 + " " +
word2])
```

次のスクリプトは、上記のコードを評価します。レスポンスには次の 2 つの配列項目が含まれます。1 つは長さに関するものであり、もう 1 つは連結された文字列に関するものです。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.eval(
   'Sequence.from([word1.length + word2.length, word1 + " "
+ word2])',
   {word1: 'hello', word2: 'world'}
).result(function(response) {
    console.log(JSON.stringify(response, null, 2));
}, function(error) {
    console.log(JSON.stringify(error, null, 2));
});
```

スクリプトを実行すると、次の出力が生成されます。

```
[
  {
    "format": "text",
    "datatype": "integer",
    "value": 10
  },
  {
    "format": "text",
    "datatype": "string",
    "value": "hello world"
  }
]
```

次のスクリプトは、DatabaseClient.xqueryEval を使用してブロックの XQuery を評価します。これは、前述した JavaScript の eval と同じ操作を実行します。出力は前の場合と全く同じになります。XQuery では、アドホックコードで外部変数を明示的に宣言する必要があります。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.xqueryEval(
  'xquery version "1.0-ml";' +
  'declare variable $word1 as xs:string external;' +
```

```
    'declare variable $word2 as xs:string external;' +
    '(fn:string-length($word1) + fn:string-length($word2),' +
    ' concat($word1, " ", $word2))',
    {word1: 'hello', word2: 'world'}
).result(function(response) {
    console.log(JSON.stringify(response, null, 2));
}, function(error) {
    console.log(JSON.stringify(error, null, 2));
});
```

その他の例については、GitHub 上の Node.js クライアント API ソースプロジェクトの `test-basic/server-exec.js` を参照してください。

7.5.3 MarkLogic サーバー上にインストールされているモジュールの呼び出し

`DatabaseClient.invoke` は、MarkLogic サーバー上にインストールされている XQuery またはサーバーサイド JavaScript モジュールに対して使用できます。これは、ビルトインサーバー関数 `xdmp.invoke`（JavaScript）または `xdmp:invoke`（XQuery）を呼び出すのと同じです。invoke を使用するには、追加のセキュリティ権限が必要です。詳細については、「必要な権限」（266 ページ）を参照してください。

呼び出すモジュールは、事前に MarkLogic サーバーにインストールしておく必要があります。モジュールは、`DatabaseClient.config.extlibs.write` または同等の操作を使用して、REST API インスタンスに関連付けられている modules データベースにインストールできます。詳細については、「modules データベースでのアセットの管理」（275 ページ）を参照してください。

> **注：** `DatabaseClient.config.extlibs.write` を使用してモジュールをインストールすると、`/ext/` プレフィックスがパスに追加されます。このプレフィックスは、`config.extlibs` インターフェイスを使用する場合は省略し、`invoke` を呼び出す場合はモジュールパスに含めてください。

モジュールをインストールする場合は、モジュールパス、コンテンツタイプ、およびソースコードを含める必要があります。JavaScript モジュールの場合は、コンテンツタイプを `application/vnd.marklogic-javascript` に設定し、モジュールパスのファイル拡張子を `.sjs` に設定します。XQuery モジュールの場合は、コンテンツタイプを `application/xquery` に設定し、モジュールパスのファイル拡張子を `.xqy` に設定します。以下の例を参照してください。

外部変数を使用すると、ランタイム中であっても任意の値をモジュールに渡すことができます。詳細については、「外部変数値の指定」（273 ページ）を参照してください。

呼び出しの応答は、呼び出されたモジュールによって返された値ごとに 1 つの項目が含まれている配列です。詳細については、「Eval または Invoke の結果の解釈」（272 ページ）を参照してください。

次の例は、MarkLogic サーバー上に JavaScript モジュールをインストールし、`DatabaseClient.invoke` を使用してそれを評価します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

// (1) Install the module in the modules database
//     Note: You do not need to install on every invocation.
//      It is included here to make the example
self-contained.
db.config.extlibs.write({
  path: '/invoke/example.sjs',
  contentType: 'application/vnd.marklogic-javascript',
  source: 'Sequence.from([word1, word2, word1 + " " +
word2])'
}).result().then(function(response) {
  console.log('Installed module: ' + response.path);

  // (2) Invoke the module
  return db.invoke({
    path: '/ext/' + response.path,
    variables: {word1: 'hello', word2: 'world'}
  }).result(function(response) {
    console.log(JSON.stringify(response, null, 2));
  }, function(error) {
    console.log(JSON.stringify(error, null, 2));
  });
}, function(error) {
    console.log(JSON.stringify(error, null, 2));
});
```

このスクリプトをファイルに保存して実行すると、次のような結果が表示されます。

```
[
  {
    "format": "text",
    "datatype": "string",
    "value": "hello"
  },
  {
    "format": "text",
```

```
    "datatype": "string",
    "value": "world"
  },
  {
    "format": "text",
    "datatype": "string",
    "value": "hello world"
  }
]
```

同等の XQuery モジュールをインストールするには、次のような呼び出しを使用します。

```
db.config.extlibs.write({
  path: '/invoke/example.xqy',
  contentType: 'application/xquery',
  source:
    'xquery version "1.0-ml";' +
    'declare variable $word1 as xs:string external;' +
    'declare variable $word2 as xs:string external;' +
    '($word1, $word2, fn:concat($word1, " ", $word2))'
})
```

7.5.4 Eval または Invoke の結果の解釈

`DatabaseClient.eval`、`DatabaseClient.xqueryEval`、または `DatabaseClient.invoke` を使用してサーバーサイドコードを評価または呼び出した場合、応答は常にサーバーが返した値ごとに 1 つの項目が含まれた配列になります。

各項目には、アプリケーションが値を解釈するのに役立つ情報が含まれています。各項目は次の形式になっています。`format` と `value` は常に存在し、`datatype` は任意です。

```
{
  format: 'text' | 'json' | 'xml' | 'binary'
  datatype: string
  value: ...
}
```

datatype プロパティは、ノードタイプ、XSD データ型、または `cts:query` などの任意のサーバータイプになります。報告されるタイプは、実際のタイプよりも一般的になる場合があります。`anyAtomicType` から派生したタイプとしては、`anyURI`、`boolean`、`dateTime`、`double`、および `string` が挙げられます。詳細については、http://www.w3.org/TR/xpath-functions/#datatypes を参照してください。

次の表は、value プロパティ内のデータの表現がどのように判断されるのかを示したものです。

format	datatype	value 表現
json	node()	パース済みの JavaScript オブジェクトまたは配列
text	任意のアトミックタイプ	datatype が文字列からの変換を許容する場合は、JavaScript のブール型、数字、または null 値。それ以外の場合は、文字列値。例えば、datatype が integer の場合、value は数字です。
xml	node()	string
binary		Buffer オブジェクト

例えば、アトミック値（anyAtomicType、anyAtomicType から派生したタイプ、または同等の JavaScript のタイプ）は、integer、string、または date などの明示的なタイプを指定可能な datatype プロパティを持ちます。

コードまたはモジュールが JSON（または JavaScript オブジェクトあるいは配列）を返す場合、値はパース済み JavaScript オブジェクトまたは配列になります。次に例を示します。

```
db.eval('var result = {number: 42, phrase: "hello"};
result;')

==>
[ { format: 'json',
    datatype: 'node()',
    value: { number: 42, phrase: 'hello' }
} ]
```

7.5.5 外部変数値の指定

外部変数を使用して、ランタイム時にアドホッククエリ（または呼び出されたモジュール）に値を渡すことができます。次の形式の JavaScript オブジェクトを使用して、eval および invoke 呼び出しに対して外部変数を指定します。値は、JavaScript プリミティブ型でなければなりません。

```
{ varName1: varValue1, varName2: varValue2, ...}
```

例えば、次のオブジェクトは、word1 および word2 という名前の 2 つの外部変数に対応する値を提供します。

```
{ word1: 'hello', word2: 'world' }
```

XQuery コードを評価または呼び出す場合は、アドホッククエリまたはモジュールで変数を明示的に宣言する必要があります。例えば、次のプロローグは 2 つの外部変数を宣言していますが、これらの値は上記のパラメータオブジェクトによって提供できます。

```
xquery version "1.0-ml";
declare variable $word1 as xs:string external;
declare variable $word2 as xs:string external;
...
```

名前空間内の変数に依存する XQuery コードを評価または呼び出す場合は、変数名で Clark 表記法を使用します。つまり、{*namespaceURI*}*name* という形式の表記法を使用して名前を指定します。

例えば、次のスクリプトでは、名前空間で修飾された外部変数 $my:who を使用します。外部変数の入力パラメータは、Clark 表記法の完全修飾された変数を使用します。例えば、{'{http://example.com}who':'world'} という形式です。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
var db = marklogic.createDatabaseClient(my.connInfo);

db.xqueryEval(
  'xquery version "1.0-ml";' +
  'declare namespace my = "http://example.com";' +
  'declare variable $my:who as xs:string external;' +
  'fn:concat("hello ", $my:who)',
  {'{http://example.com}who' : 'world'}
).result(function(response) {
    console.log(JSON.stringify(response, null, 2));
}, function(error) {
    console.log(JSON.stringify(error, null, 2));
});
```

7.6 modules データベースでのアセットの管理

`DatabaseClient.eval` で使用可能な XQuery および JavaScript モジュール、およびリソースサービス拡張機能および変換機能で使用される依存ライブラリなど、アプリケーションが必要とするサーバーサイドアセットをインストールおよび管理するには、DatabaseClient.config.extlibs インターフェイスを使用します。

このセクションでは、次の内容を取り上げます。

- アセット管理の概要
- アセットのインストールまたは更新
- サーバーサイドコードからのアセットの参照
- アセットの削除
- アセットリストの取得
- アセットの取得

7.6.1 アセット管理の概要

Node.js クライアント API アプリケーションは、変換、リソースサービス拡張の実装、制約結合パーサー、カスタムスニペットジェネレータ、パッチコンテンツジェネレータなど、REST API インスタンスに関連付けられている modules データベースに格納されているいくつかのタイプのユーザー定義コードを使用できます。

これらのアセットクラスのほとんどには、`DatabaseClient.config.resources` や `DatabaseClient.config.query.snippet` といった専用の管理インターフェイスがあります。これにより API によるアセット管理方法の詳細を抽象化します。通常、このようなアセットは汎用的なインターフェイスで管理すべきではありません。専用インターフェイスがないアセットは、`DatabaseClient.config.extlibs` インターフェイスで管理できます。

次の表は、Node.js クライアント API を通じて使用可能なアセット管理インターフェイスを示したものです。

インターフェイス	管理対象
`DatabaseClient.config.extlibs`	`DatabaseClient.invoke` を使用して呼び出すことができる XQuery および JavaScript モジュール。詳細については、「MarkLogic サーバー上にインストールされているモジュールの呼び出し」（270 ページ）を参照してください。 リソースサービス拡張機能や変換機能で必要となる依存ライブラリおよび他のアセット。詳細については、「リソースサービス拡張機能を扱う」（235 ページ）を参照してください。
`DatabaseClient.config.patch.replace`	DatabaseClient.documents.patch の置換コンテンツジェネレータ。詳細については、「MarkLogic サーバーでの置換データの作成」（116 ページ）を参照してください。
`DatabaseClient.config.query.custom`	カスタムのクエリバインドおよびファセットジェネレータ。詳細については、「カスタム制約パーサーの使用」（147 ページ）および「検索ファセットの生成」（184 ページ）を参照してください。
`DatabaseClient.config.query.snippet`	カスタムのスニペットジェネレータ。詳細については、「検索スニペットの生成」（192 ページ）を参照してください。
`DatabaseClient.config.resources`	リソースサービス拡張機能。詳細については、「リソースサービス拡張機能を扱う」（235 ページ）を参照してください。
`DatabaseClient.config.transforms`	読み取り、書き込み、およびクエリの変換機能。詳細については、「コンテンツ変換機能を扱う」（245 ページ）を参照してください。

すべてのアセット管理インターフェイスは、同一のメソッドの基本セットを提供しますが、特定のアセットクラスに適したカスタマイズが行われています。

- `write` : modules データベースにアセットをインストールします。
- `read` : modules データベースからアセットを取得します。
- `list` : すべてのリソースサービス拡張機能またはすべてのファセットジェネレータなど、modules データベースから特定のクラスのすべてのアセットのリストを取得します。
- `remove` : modules データベースからアセットを削除します。

アセットクラス間でインターフェイスを組み合わせてはなりません。例えば、`DatabaseClient.config.query.snippet.write` を使用してスニペッタをインストールし、`DatabaseClient.config.extlibs.remove` を使用してそれを削除してはなりません。アセットの管理は、Java クライアント API および REST クライアント API など、他のクライアント API の同等のインターフェイスを通じて行うことができます。

modules データベースにアセットをインストールしたり、modules データベース内のアセットを更新したりすると、そのアセットはクラスタ全体にわたって自動的にレプリケートされます。ただし、更新してから使用できるようになるまで最大 1 分の遅延が生じることがあります。

MarkLogic サーバーは、ユーザーが関連する拡張機能または変換機能を削除した場合でも、依存アセットを自動的に削除しません。

依存アセットは modules データベースにインストールされるため、modules データベースをインスタンスのティアダウンに含めた場合、REST API インスタンスを削除すると依存アセットが削除されます。詳細については、『REST Application Developer's Guide』の「Removing an Instance」を参照してください。

7.6.2 アセットのインストールまたは更新

このセクションでは、`DatabaseClient.invoke` を使用して呼び出す依存ライブラリまたはモジュールなど、専用のアセット管理インターフェイスがないアセットをインストールまたは更新する方法について説明します。それ以外のアセットクラスには、専用インターフェイスの `write` メソッドを使用します。専用インターフェイスのリストについては、「アセット管理の概要」（275 ページ）を参照してください。

REST API インスタンスに関連付けられている modules データベースにアセットをインストールしたり、modules データベース内のアセットを更新したりするには、`DatabaseClient.config.extlibs.write` を使用します。この場合、モジュールパス、コンテンツタイプ、およびアセットコンテンツを指定する必要があります。アセットは、JSON、XML、テキスト、またはバイナリドキュメントとして modules データベースに挿入できます。ドキュメント形式は、MarkLogic サーバーが判断します。ドキュメントタイプは、コンテンツタイプ、またはモジュールパス URI のファイル拡張子、およびサーバーの MIME タイプマッピングによって判断されます。

インストール時に、ユーザーが指定したモジュールパスの先頭に `/ext/` というプレフィックスが付加されます。このプレフィックスは、`extlibs` インターフェイスを使用してアセットを操作するときは省略できますが、絶対パスを使用するリソースサービス拡張機能 `require` ステートメントや、`DatabaseClient.invoke` を使用してモジュールを呼び出す場合など、他の場所でモジュールを参照する場合は含める必要があります。

次の例は、コンテンツがファイルから読み取られるモジュールをインストールします。このモジュールは、URI が `/ext/extlibs/example.sjs` の modules データベースにインストールされます。

```
var fs = require('fs');
var marklogic = require('marklogic');
var db = marklogic.createDatabaseClient(my.connInfo);

...
db.config.extlibs.write({
  path: '/extlibs/example.sjs',
  contentType: 'application/vnd.marklogic-javascript',
  source: fs.createReadStream('./example.sjs')
})...
```

その他の例については、「MarkLogic サーバー上にインストールされているモジュールの呼び出し」（270 ページ）または GitHub の `marklogic/node-client-api` プロジェクトの `test-basic/extlibs.js` を参照してください。

7.6.3 サーバーサイドコードからのアセットの参照

`DatabaseClient.extlibs.write` でインストールした依存ライブラリを、拡張機能、変換機能、または呼び出されたモジュールから使用するには、依存ライブラリのインストール先と同じ URI を `/ext/` プレフィックスを含めて使用します。

例えば、`db.config.extlibs.write({path: '/my/domain/lib/`*`myasset`*`', ...})` を使用して依存アセットをインストールした場合、modules データベース内の URI は `/ext/my/domain/`*`myasset`* です。

このアセットを使用している JavaScript 拡張機能、変換機能、または呼び出されたモジュールは、次のようにこれを参照できます。

```
var myDep = require('/ext/my/domain/lib/myasset');
```

このライブラリを使用している XQuery 拡張機能、変換機能、または呼び出されたモジュールには、次の形式の import を含めることができます。

```
import module namespace dep="mylib" at
"/ext/my/domain/lib/myasset";
```

7.6.4 アセットの削除

`DatabaseClient.config.extlibs.write` を使用してインストールしたアセットを modules データベースから削除するには、`DatabaseClient.config.extlibs.remove` を使用します。拡張機能や変換機能など、専用インターフェイスがあるアセットの場合は、`DatabaseClient.config.resources.remove` といった専用インターフェイスの `remove` メソッドを使用します。

アセットの削除は、冪等（べきとう）操作（何回繰り返しても、1 回だけ実行したときと同じ結果になる操作）です。つまり、アセットが存在するかどうかにかかわらず同じ応答を返します。

特定のディレクトリ内のすべてのアセットを削除するには、特定のアセットのパスではなく、アセットが含まれているディレクトリの名前を指定します。

例えば、アセットが次のようにインストールされているとします。

```
db.config.extlibs.write({
  path: '/invoke/example.sjs',
  contentType: 'application/vnd.marklogic-javascript',
  source: ...
})
```

この場合、そのアセットは次のような呼び出しで削除できます。

```
db.config.extlibs.remove('/invoke/example.sjs');
```

/ext/invoke/ にインストールされているすべてのアセットを削除するには、次のような呼び出しを使用します。

```
db.config.extlibs.remove('/invoke/');
```

7.6.5 アセットリストの取得

DatabaseClient.config.extlibs.write を使用してインストールしたアセットのリストを取得するには、DatabaseClient.config.extlibs.list を使用します。拡張機能や変換機能など、専用インターフェイスがあるアセットの場合は、DatabaseClient.config.transforms.list といった専用インターフェイスの list メソッドを使用します。

応答の形式は次のとおりです。

```
{ "assets": [
    { "asset": "/ext/invoke/example.sjs" },
    { "asset": "/ext/util/dep.sjs" },
    { "asset": assetModulePath }, ...
]}
```

7.6.6 アセットの取得

DatabaseClient.config.extlibs.write を使用してインストールしたアセットを取得するには、DatabaseClient.config.extlibs.read を使用します。拡張機能や変換機能など、専用インターフェイスがあるアセットの場合は、DatabaseClient.config.transforms.read といった専用インターフェイスの read メソッドを使用します。

アセットをインストールするのに使用したのと同じモジュールパスを使用してアセットを取得します。次に例を示します。

```
db.config.extlibs.read('/invoke/example.sjs')
```

8.0 REST API インスタンスの管理

Node.js クライアント API は、サーバーと通信を行い、データベースにアクセスするために、MarkLogic サーバー上の REST クライアント API インスタンスを必要とします。この章では、インスタンスの作成方法と管理方法について説明します。

ここでは、次の内容を取り上げます。

8.1 REST API インスタンスとは

Node.js クライアント API の実装は、『REST Application Developer's Guide』で説明している REST クライアント API を使用して MarkLogic サーバーと通信します。そのため、Node.js クライアント API を通じて MarkLogic サーバーにリクエストを行うには、REST API インスタンスが存在している必要があります。REST API インスタンスは、REST クライアント API リクエストを処理できるように特別に設定された HTTP アプリケーションサーバー、デフォルトのコンテンツデータベース、および modules データベースで構成されています。

> **注：** 各 REST API インスタンスがホストできるのは、1 つのアプリケーションだけです。REST API アプリケーションが複数存在する場合は、それぞれのアプリケーションごとに 1 つのインスタンスを作成し、それぞれが独自の modules データベースを持っている必要があります。

MarkLogic サーバーをインストールすると、事前設定された REST API インスタンスがポート 8000 で使用可能になります。このインスタンスは、MarkLogic サーバーのインストール後に使用できるようになります。追加の手順は不要です。このインスタンスは、デフォルトのコンテンツデータベースとして documents データベースを使用し、モジュールデータベースとして modules データベースを使用します。

ポート 8000 のインスタンスは最初は便利ですが、通常は実稼動用にその後、専用インスタンスを作成します。この章では、独自のインスタンスを作成および管理する方法について説明します。

`marklogic.createDatabaseClient` を使用して `DatabaseClient` オブジェクトを作成する場合は、REST API インスタンスへの接続を作成します。また、`DatabaseClient` を作成する場合は、インスタンスに関連付けられているデフォルトのコンテンツデータベース以外のコンテンツデータベースを指定できます。代替のデータベースを使用するには、追加のセキュリティ権限が必要です。詳細については、「異なるデータベースに対するリクエストの評価」（17 ページ）を参照してください。

REST API インスタンスに関連付けられたデフォルトのコンテンツデータベースは、インスタンスの作成時に自動的に作成されます。また、インスタンスを作成する前に個別に作成することもできます。インスタンスには、任意のコンテンツデータベースを関連付けることができます。コンテンツデータベースは、管理画面、XQuery または JavaScript Admin API、あるいは REST 管理 API を使用して通常どおりに管理します。

REST インスタンスの modules データベースはインスタンスの作成時に自動的に作成されます。また、インスタンスを作成する前に個別に作成することもできます。modules データベースを事前に作成する場合は、インスタンス間でこれを共有しないでください。インスタンスの作成時は、特別なコードが modules データベースに挿入されます。modules データベースは、永続的なクエリオプション、拡張機能、コンテンツ変換機能、カスタムパーサー、および `DatabaseClient.config` インターフェイスを使用してインストールしたその他のアセットも保持します。

この章で説明しているインスタンスプロパティを除き、インスタンスに関連付けられたアプリケーションサーバーを事前に設定することはできません。ただし、インスタンスの作成後に、管理画面、XQuery または JavaScript Admin API、あるいは REST 管理 API を使用して、リクエストのタイムアウトなどのプロパティをカスタマイズできます。

8.2 インスタンスの作成

MarkLogic サーバーをインストールすると、事前設定された REST API インスタンスがポート 8000 で使用可能になります。ただし、REST クライアント API を使用して独自のインスタンスを作成することもできます。

新しい REST インスタンスを作成するには、次の形式の URL でポート 8002 の `/rest-apis` サービスに POST リクエストを送信します。

```
http://host:8002/version/rest-apis
```

version には、キーワード LATEST または最新バージョンを使用できます。また、POST ボディには、JSON または XML インスタンスの設定を含める必要があります。設定には、少なくとも名前を含める必要がありますが、ポート番号、コンテンツ、modules データベース名、およびその他のインスタンスプロパティを含めることもできます。

例えば、次のコマンドは、cURL コマンドラインツールを使用して、デフォルトのポート、データベース、プロパティで「RESTstop」という名前のインスタンスを作成します。

```
curl --anyauth --user user:password -X POST -i \
  -d '{"rest-api": {"name":"RESTstop" }}' \
  -H "Content-type: application/json" \
  http://localhost:8002/LATEST/rest-apis
```

詳細および例については、『REST Application Developer's Guide』の「Creating an Instance」を参照してください。Node.js ライブラリを使用してインスタンスを作成する例については、GitHub 上の Node.js クライアント API ソースコードプロジェクトの `etc/test-setup.js` を参照してください。

8.3 インスタンスプロパティの設定

インスタンスの作成後は、複数のインスタンスプロパティを確認および変更できます。例えば、`document-transform-out` プロパティを使用して、デフォルトの読み取り変換機能を指定できます。

インスタンスプロパティの読み取りおよび書き込みを行うには、`DatabaseClient.config.serverprops` インターフェイスを使用します。このインターフェイスを使用するには、`rest-admin` ロールまたは同等の権限を持っている必要があります。使用可能なプロパティの説明については、『REST Application Developer's Guide』の「Instance Configuration Properties」を参照してください。

現在の設定を取得するには、`read` メソッドを使用します。応答は、すべてのプロパティが含まれているオブジェクトになります。次に例を示します。

```
db.config.serverprops.read()
```

プロパティを設定するには、write メソッドを使用して、変更する各インスタンスプロパティのオブジェクトプロパティが含まれているオブジェクトを渡します。read によって返されるオブジェクトは、write への入力として適切です。

```
db.config.serverprops.write(props)
```

例えば、次のスクリプトは現在のインスタンスプロパティを読み取り、結果を使用して `debug` プロパティの値を切り替え、`debug` 設定のみが含まれているプロパティディスクリプタを使用して元の値に戻します。

```
var marklogic = require('marklogic');
var my = require('./my-connection.js');
```

```
var db = marklogic.createDatabaseClient(my.connInfo);

db.config.serverprops.read().result()
  .then(function(props) {
    console.log("Current instance properties:");
    console.log(props);
    // flip the debug property setting
    props.debug = !props.debug;
    return db.config.serverprops.write(props).result();
  }).then(function(response) {
    console.log("Props updated: " + response);
    // demonstrate the setting changed
    return db.config.serverprops.read().result();
  }).then(function(props) {
    console.log("Debug setting is now: " + props.debug);
    // flip the setting back using sparse properties
    var newProps = {};
    newProps.debug = !props.debug;
    return db.config.serverprops.write(newProps).result();
  }).then(function(response) {
    return db.config.serverprops.read().result();
  }).then(function(props) {
    console.log("Debug setting is now: " + props.debug);
  });
```

スクリプトを実行すると、次のような出力が表示されます。実際のインスタンスプロパティ値は異なることもあります。

```
{ 'content-versions': 'none',
  'validate-options': true,
  'document-transform-out': '',
  debug: false,
  'document-transform-all': true,
  'update-policy': 'merge-metadata',
  'validate-queries': false }
Props updated: true
Debug setting is now: true
Debug setting is now: false
```

8.4 設定情報の取得

ポート 8002 の `/rest-apis` サービスに対して GET リクエストを使用すると、ホスト上のすべての REST API インスタンスに関する設定情報や、インスタンス名またはコンテンツデータベースによって指定した特定のインスタンスに関する設定情報を取得できます。

詳細については、『REST Application Developer's Guide』の「Retrieving Configuration Information」を参照してください。

8.5 インスタンスの削除

REST クライアント API のインスタンスを削除するには、ポート 8002 の `/rest-apis` サービスに DELETE リクエストを送信します。このとき、コンテンツデータベースを変更するかどうかを選択できます。

警告 通常、この手順はポート 8000 にある事前設定された REST API インスタンスには適用しないでください。これを行うと、XDBC、Query Console、および REST 管理 API など、そのポートの他のサービスが無効になることがあります。

詳細および例については、『REST Application Developer's Guide』の「Removing an Instance」を参照してください。