Microsoft

SQL Server 2017

SQL Server 2017 自習書シリーズ No.2

SQL Server 2017 on Linux

Published: 2017 年 11 月 1 日

有限会社エスキューエル・クオリティ

SQLQuality
データベース コンサルティング

目次

STEP 1. SQL Server 2017 on Linux の概要

この STEP では、SQL Server の最新バージョンである「**SQL Server 2017**」で提供された **Linux** 対応機能の概要を説明します。

この STEP では、次のことを学習します。

- ✓ SQL Server 2017 の主な新機能
- ✓ SQL Server 2017 on Linux の概要

1.1 SQL Server 2017 で提供された主な新機能

SQL Server の最新バージョンである「**SQL Server 2017**」は、2017 年 10 月に発売されました。1 つ前のバージョンである SQL Server 2016 の発売は 2016 年 6 月でしたので、わずか 1 年半弱でのバージョン アップになりますが、SQL Server 2017 には非常に多くの新機能が提供されています。その主なものは、次のとおりです。

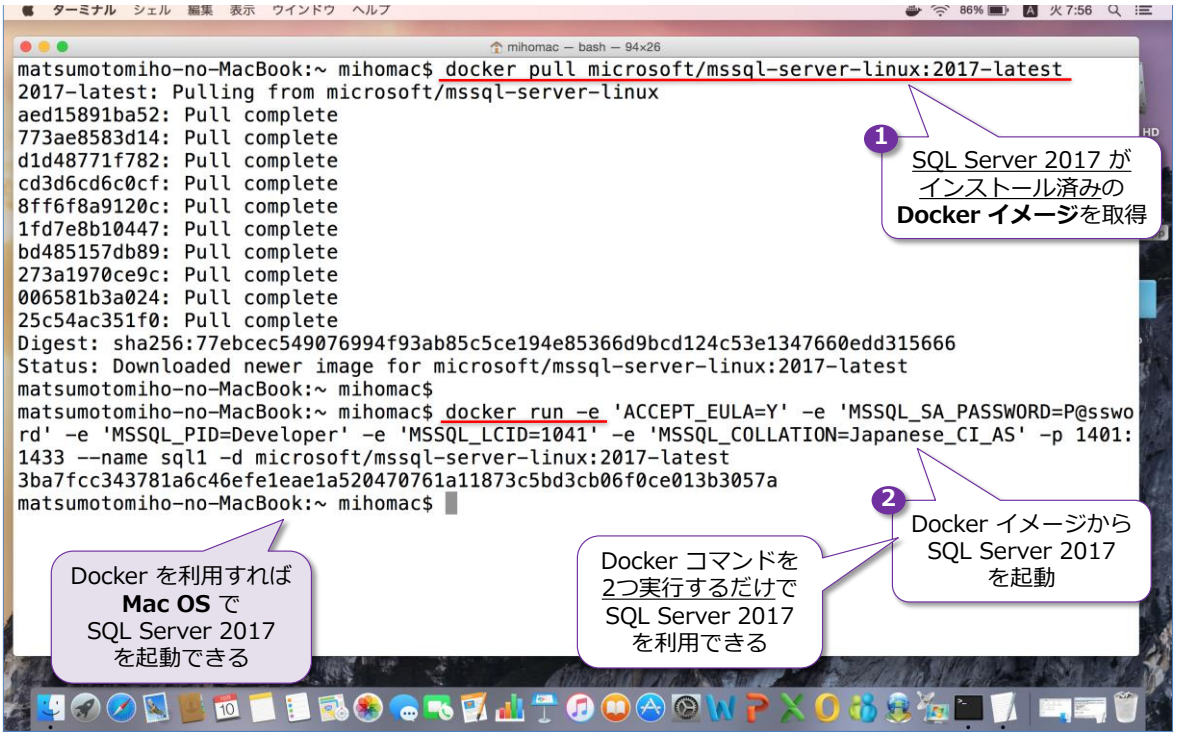
	SQL Server 2017 からの主な新機能
Linux 対応	・**SQL Server 2017 on Linux** サポート プラットフォーム - Red Hat Enterprise Linux 7.3／7.4 Workstation, Server, and Desktop - SUSE Enterprise Linux Server v12 SP2 - Ubuntu 16.04 LTS - Docker Engine 1.8 以上 ・**on Linux で利用できる機能** - データベース エンジンに関するほぼすべての機能（詳細は 2章参照） - SQL Server Agent ジョブ（Transact-SQL の定期実行） - SSIS パッケージ（.dtsx）の実行
ビルトイン AI 機械学習	・**Python 統合**（Machine Learning Services） Python 言語をデータベース エンジンに統合（ビルトイン） Python における**ディープ ラーニング**（Deep Learning : 深層学習）の 定番フレームワークである CNTK（Microsoft Cognitive Toolkit）や Chainer、TensorFlow、Caffe、Theano などを利用可能。GPU にも対応 ・**ネイティブ スコアリング**（PREDICT 関数による予測の実行）
注目の新機能	・**グラフ データベース**（Graph processing） ・**自動チューニング**（Automatic Tuning） ・**Adaptive Query Processing**（適応型クエリ処理） ・**クエリ ストア**の強化、**DTA** の強化、Scan/Read Ahead アルゴリズム改善 ・**列ストア インデックス**の強化（オンライン再構築や LOB 対応、性能向上etc） ・**インメモリ OLTP** の強化（制限緩和の追加や性能向上etc） ・**再開可能**なオンライン インデックス再構築（Resumable Rebuild Index） ・**AlwaysOn 可用性グループ**の強化（DTC 対応、クラスター レス構成など）
その他の新機能	・**Transact-SQL の強化** 新しい関数（TRIM、CONCAT_WS、TRANSLATE、STRING_AGG）、 SELECT INTO でのファイル グループ指定、IDENTITY_CACHE オプション のサポート、日本語向けの新しい照合順序の追加など ・**セットアップ時の変更点**（tempdb ファイルの設定） ・**スマート バックアップ**、バックアップ性能の向上 ・**新しい DMV**（動的管理ビュー） ・**テンポラル テーブルの強化**（保持ポリシーの追加など） ・**XE プロファイラー**の提供、実行プランの検索機能 ・**SQL CLR** の**セキュリティ強化**
BI 関連の強化	・**Analysis Services の強化** Power Query Formula Language（M 言語）対応、ドリルスルー データ での DAX 指定、Ragged 階層対応、Object レベル セキュリティ、 DAX 強化、DMV 強化、DISCOVER_CALC_DEPENDENCY など ・**Reporting Services の強化** レポート コメント、DAX エディター for SSDT、REST API 対応、 軽量のインストーラー など ・**Integration Services の強化** SSIS パッケージのスケールアウト実行、Linux 対応 など ・**MDS（マスター データ サービス）の強化**

この中でも、目玉の新機能は、やはり**マルチ プラットフォーム**（**Linux** や **Mac OS** に対応）と、**ビルトイン AI**（AI 機能を SQL Server データベース エンジンに統合）、**自動チューニング**（Automatic Tuning）機能の搭載、**グラフ データベース**対応などです。ビルトイン AI や自動チューニングなどの新機能については、本自習書シリーズの No.1「**SQL Server 2017 の新機能の概要**」編で詳しく説明しているので、こちらもぜひご覧いただければと思います。

1.2 SQL Server 2017 on Linux の概要

SQL Server 2017 では、ついに **Linux**（&**Docker**）をサポートしたので、Windows 環境ではもちろんのこと、**Linux**（Ubuntu や Red Hat Enterprise Linux）でも、**Mac OS**（Docker を利用）でも動作させられるようになりました。

Docker を利用する場合は、既に **SQL Server 2017** がインストールされた状態の Docker イメージが提供されているので、**Docker コマンドを 2 つ実行**するだけで、SQL Server 2017 をインストールする必要なく、簡単に SQL Server 2017 を試すことができます。



Docker での SQL Server 2017 の利用方法については、第 2 章で詳しく説明しているので、ぜひ試してみてください。**Linux** 環境への SQL Server 2017 のインストール方法についても、第 2 章で詳しく説明していますが、これも **5 個のコマンド**を実行するだけで、簡単にインストールすることができます。こちらもぜひ試してみてください。

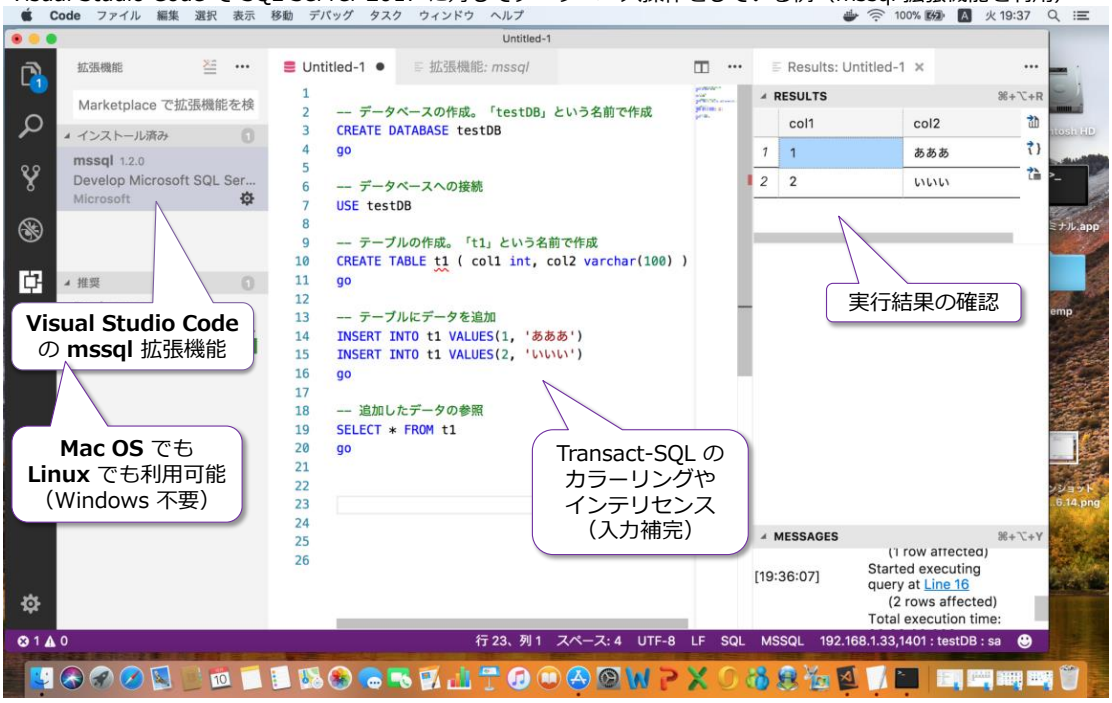
SQL Server 2017 on Linux でできること

SQL Server 2017 on Linux は、ただ単に Linux 上で SQL Server を動かせるようにしただけではなく、ほとんどの SQL Server の機能（データベース エンジンに関する機能）を、Windows 上の SQL Server とまったく同じように利用できます。データベースに関する**基本操作**（データベースの作成やテーブル作成、データの追加／更新／削除、ビューやストアド プロシージャ、トリガー、インデックスの作成）ができることはもちろん、性能向上を実現できる**列ストア インデックス**や、**インメモリ OLTP**、**データ パーティション**、**データ圧縮**も利用できます。

Visual Studio Code などのアプリケーション開発ツールを利用すれば、Windows を利用するこ

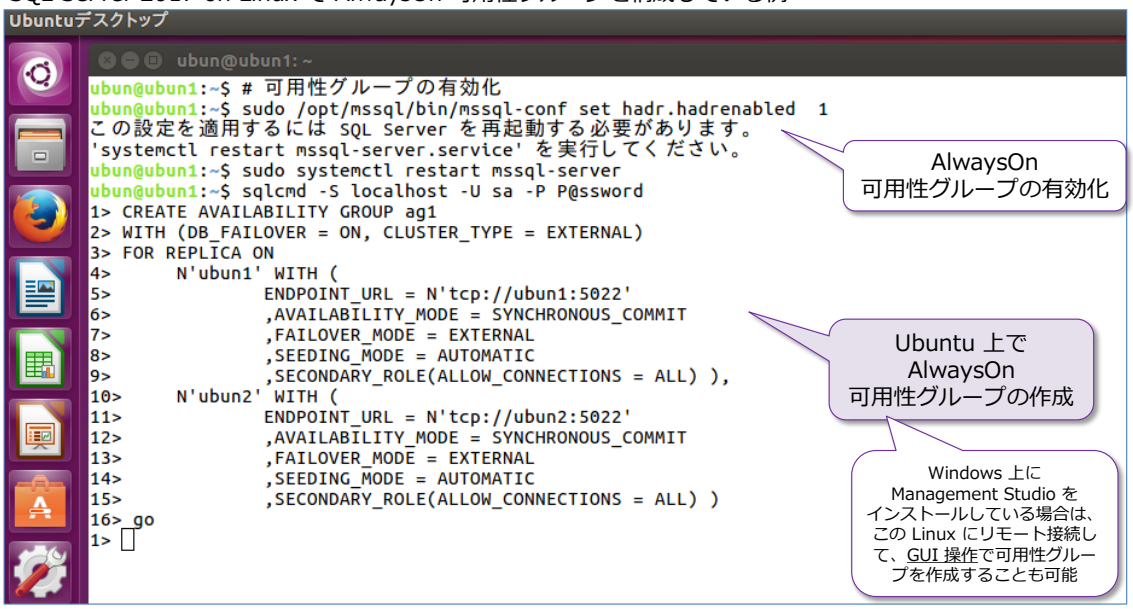
となく、Mac OS や Linux だけで、SQL Server 2017 に関する操作をグラフィカルに行うことができ、合わせてアプリケーション開発（C# や ASP.NET、Java、Python、node.js など）も可能です（詳しくは、次章以降に説明します）。

Visual Studio Code で SQL Server 2017 に対してデータベース操作をしている例（mssql 拡張機能を利用）



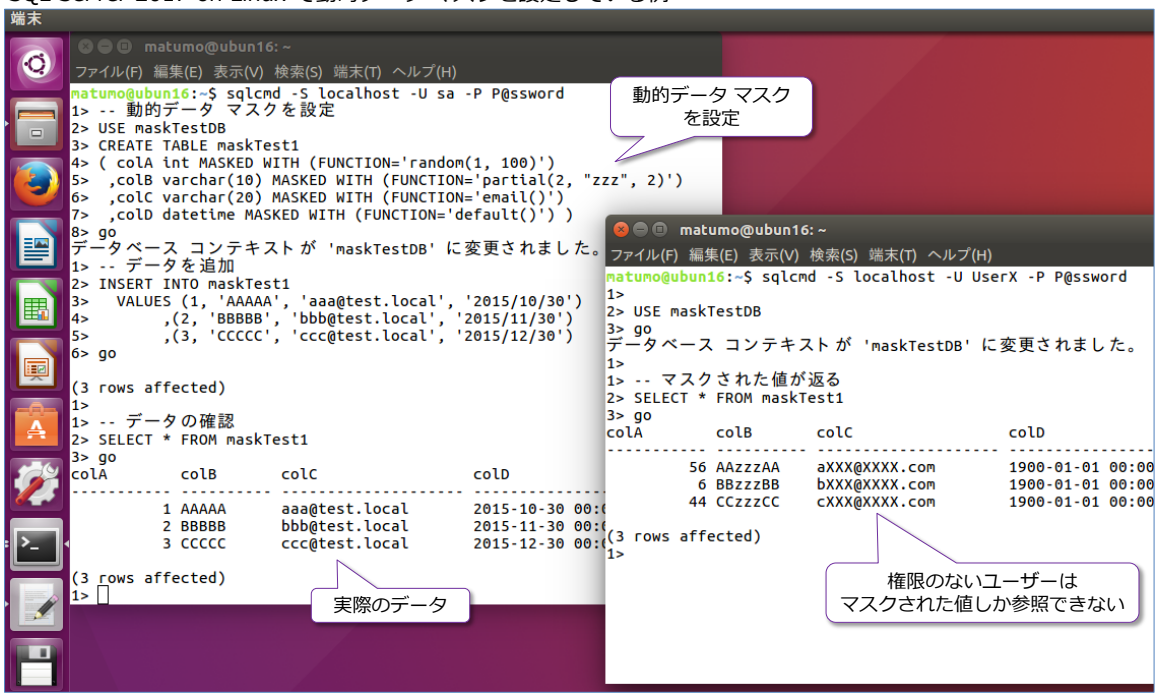
SQL Server 2017 on Linux では、**自動チューニング**機能や、**クエリ ストア**、**グラフ データベース**、**AlwaysOn 可用性グループ**、**スマートバックアップ**などについても、Windows 上の SQL Server を操作するのとまったく同じように利用できます。

SQL Server 2017 on Linux で AlwaysOn 可用性グループを構成している例

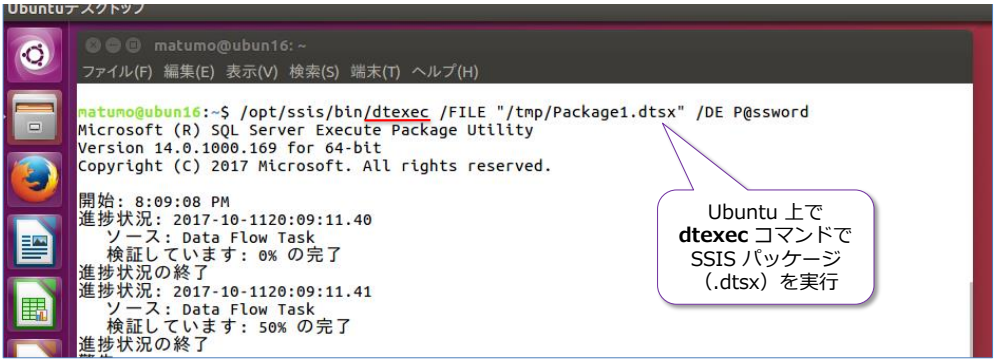


また、SQL Server が標準で搭載している**セキュリティ機能**（ユーザー作成やオブジェクト権限の設定だけでなく、監査や行レベル セキュリティ、動的データ マスク、TDE：透過的なデータ暗号化、バックアップ暗号化、ネットワーク接続の暗号化、Always Encrypted、テンポラル テーブル、包含データベースなど）についても、on Linux で同じように利用することできます。

SQL Server 2017 on Linux で動的データ マスクを設定している例



SQL Server 2017 on Linux では、その他のシステムとの連携に役立つ**リンク サーバー**や **bcp**、**BULK INSERT** なども利用することができ、**Integration Services**（SSIS）のパッケージ（.dtsx）を実行することもできます。



また、SQL Server におけるバックアップのスケジュールなどで定番となっている **SQL Server Agent ジョブ**機能もサポートしているので、バックアップやインデックスの再構築といった各種のメンテナンス系の SQL をスケジュール実行することもできます。**データベース メール**機能もサポートしているので、ジョブの成功や失敗をメールで通達するといったことも行えます。

さらには、**Machine Learning Services** で**機械学習**したモデルを利用した**予測**（**PREDICT** 関数によるスコアリング）を行うこともできます。

Windows 上の SQL Server 環境で取得したバックアップを、Linux 環境にリストアすることも、何の問題もなく行えるので、**移行**（マイグレーション）も簡単です（リストア時の考慮事項は、Windows 環境でのリストアの場合と全く同様です）。

これらの Linux 上での SQL Server の機能について、第 2 章以降でステップ バイ ステップ形式で画面ショット満載で紹介しているので、ぜひ、皆さんも実際に試しながらこの自習書を読み進めていただければと思います。

1.3 SQL Server 2017 on Linux の性能（TPC-H など）

SQL Server 2017 on Linux は、性能に関しても結果が出ています。例えば、次の画面ショットは、執筆時点（2017 年 10 月）での **TPC-H ベンチマーク**での Non-Clustered カテゴリの **1TB**（1,000GB）の結果です。

TPC-H - Top Ten Performance Results - Non-Clustered
Version 2 Results As of 15-Oct-2017 at 4:15 PM [GMT]

Printer Version

Note 1: The TPC believes that comparisons of TPC-H results measured against different database sizes are misleading and discourages such comparisons. The TPC-H results shown below are grouped by database size to emphasize that only results within each group are comparable.
Note 2: The TPC believes it is not valid to compare prices or price/performance of results in different currencies.

All Active Results | Active Clustered Results | Active Non-Clustered Results **Currency:** All | Include Historical Results

1,000 GB Results

Rank	Company	System	QphH	Price/QphH	Watts/KQphH	System Availability	Database	Operating System	Date Submitted
1	Hewlett Packard Enterprise	HPE Proliant DL380 Gen9	717,101	.61 USD	NR	10/19/17	Microsoft SQL Server 2017 Enterprise Edition	Red Hat Enterprise Linux Server 7.3	04/17/17
2	Hewlett Packard Enterprise	HPE Proliant DL380 Gen9	678,492	.64 USD	NR	07/31/16	Microsoft SQL Server 2016 Enterprise Edition	Microsoft Windows Server 2012 R2 Standard Edition	03/24/16
3	Cisco	Cisco UCS C460 M4 Server	588,831	.97 USD	NR	12/16/14	Microsoft SQL Server 2014 Enterprise Edition	Microsoft Windows Server 2012 R2 Standard	12/15/14
4	Hewlett Packard Enterprise	HPE Proliant DL380 Gen9	543,102	.69 USD	NR	07/31/16	Microsoft SQL Server 2016 Enterprise Edition	Microsoft Windows Server 2012 R2 Standard Edition	03/09/16

＊ 執筆時点（2017年 10月）での TPC-H Non-Clustered 1,000GB テストの結果

TPC-H は、データ ウェアハウス／意志決定支援システム向けの**ベンチマーク テスト**として有名なものですが、1 位は **Red Hat Enterprise Linux Server 7.3** 上で動作している **SQL Server 2017** で、2 位は **Windows Server 2012 R2** 上で動作している **SQL Server 2016** です。

2 位の SQL Server 2016 on Windows が **678,492 QphH**（1 時間あたりの DWH クエリ実行数）であるのに対して、1 位の SQL Server 2017 on Linux では **717,101 QphH**、1 クエリあたりのコスト（**Price/QphH**）に関しては、**0.64** USD（US ドル）が **0.61** USD に下がっており、性能向上とともにコスト削減も実現しています。

なお、TPC-H ベンチマークは、SQL Server 2016 のときに **10TB** テストで**世界記録**（ワールドレコード）を更新しましたが、SQL Server 2017 on Windows でさらに記録を更新しています。

10,000 GB Results

Rank	Company	System	QphH	Price/QphH	Watts/KQphH	System Availability	Database	Operating System	Date Submitted
1	Lenovo	Lenovo ThinkSystem SR950	1,336,109	.92 USD	NR	10/19/17	Microsoft SQL Server 2017 Enterprise Edition	Microsoft Windows Server 2016 Standard Edition	07/09/17
2	Cisco	Cisco UCS C460 M4 Server	1,115,298	.87 USD	NR	11/28/16	Microsoft SQL Server 2016 Enterprise Edition	Microsoft Windows Server 2016 Standard Edition	11/28/16
3	Lenovo	Lenovo System x3850 X6	1,106,832	.89 USD	NR	09/30/16	Microsoft SQL Server 2016 Enterprise Edition	Microsoft Windows Server 2016 Standard Edition	07/11/16
4	Hewlett Packard Enterprise	HPE Proliant DL580 Gen9	1,047,243	1.07 USD	NR	09/30/16	Microsoft SQL Server 2016 Enterprise Edition	Microsoft Windows Server 2016 Standard Edition	06/27/16
5	Hewlett Packard Enterprise	HP Integrity Superdome X	780,346	2.27 USD	NR	02/03/16	Microsoft SQL Server 2014 Enterprise Edition	Microsoft Windows Server 2012 R2 Standard Edition	02/02/16
6	Hewlett Packard Enterprise	HP Integrity Superdome X	680,841	2.35 USD	NR	10/31/15	Microsoft SQL Server 2014 Enterprise Edition	Microsoft Windows Server 2012 R2 Standard Edition	10/30/15

＊ 執筆時点（2017年 10月）での TPC-H Non-Clustered 10TB テストの結果

SQL Server 2016 で **1,115,298 QphH** だったところを、SQL Server 2017 では **1,336,109 QphH** を達成しました。SQL Server 2017 では、**Adaptive Query Processing**（適応型クエリ処理）や、スキャン／Read Ahead アルゴリズムの改善、インメモリ OLTP／列ストア インデックスの強化など、**データベース エンジンの性能向上**も図っているので、その成果が出ています。

480 スレッドでもスケール（SQL Server 2017 on Linux）

昨今は、1 個の CPU あたりのコア数が非常に多い "**メニー コア**" 時代になりましたが、コア数が増えてもスケールしないデータベースが存在する中（NUMA に対応していなかったり、64 コアを超えるプロセッサー グループ環境に対応していないデータベースがあったりする中）、SQL Server はメニー コア環境でもスケールします。今年の 9 月に開催された Microsoft Ignite 2017 では、**480 スレッド**分もの **CPU をフル活用**して、スケールしているデモがありました（以下）。

SQL Server 2017 on Linux で 480 スレッドの CPU をフル活用している様子。メモリは 12TB 搭載

https://myignite.microsoft.com/videos/54946 ：Microsoft Ignite 2017 Session「Microsoft SQL Server 2017 deep dive」より引用

こうしたメニー コアへの対応は、Windows 版ではお馴染みでしたが、Linux 版でも見事に対応しています（そうした結果が前述の TPC-H の結果に繋がっています）。

STEP 2. SQL Server 2017 on Linux

この STEP では、Linux 環境での SQL Server 2017 に関して、「**Docker での利用方法**」や「**Linux へのインストール方法**」、「**Linux 版で利用できる機能／利用できない機能**」などを説明します。

この STEP では、次のことを学習します。

- ✓ Docker を利用した SQL Server 2017 on Linux
- ✓ Linux 環境への SQL Server 2017 のインストール
- ✓ SQL Server 2017 on Linux でのセキュリティ
- ✓ SQL Server 2017 on Linux で利用できる機能／利用できない機能

2.1 Docker を利用した SQL Server on Linux

SQL Server 2017 は、**Docker イメージ**としても提供されているので、Docker を利用することで、Windows 環境ではもちろんのこと、**Linux** でも、**Mac OS** でも簡単に SQL Server 2017 を動作させることができます。

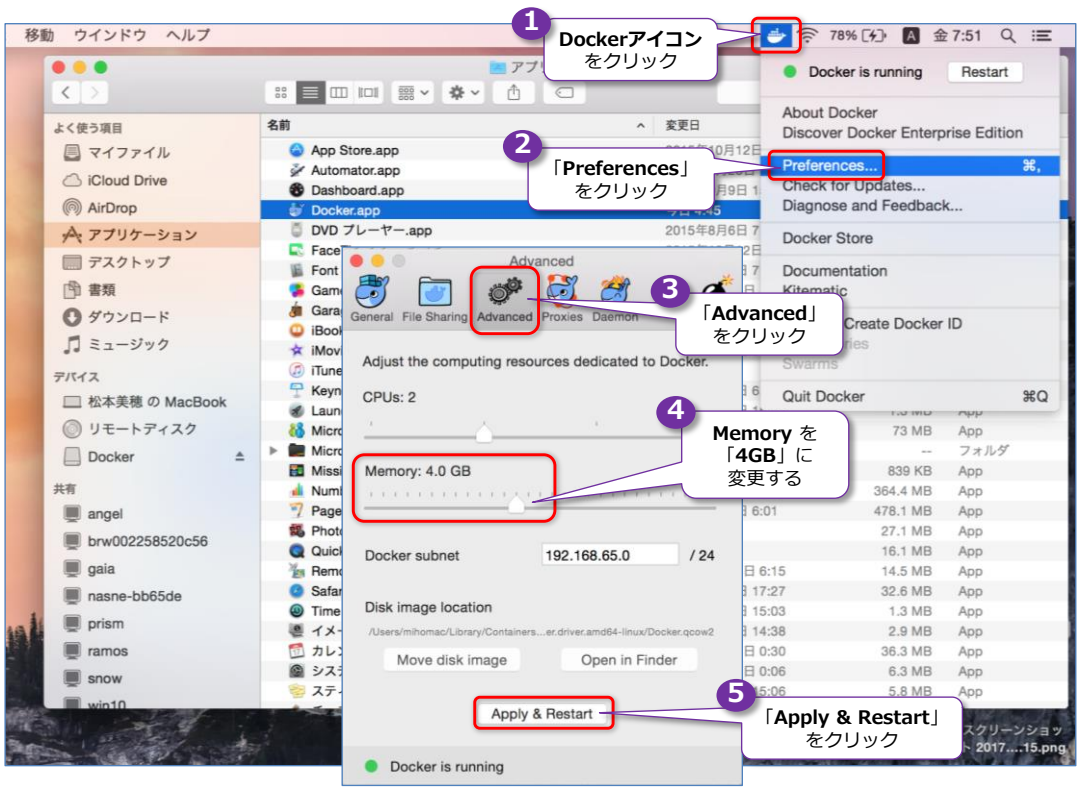
Docker 上で SQL Server 2017 を動作させるための要件は、次のとおりです。

- **Docker Engine** のバージョンに **1.8** 以上を利用する
- 最低 **4GB** のディスク領域
- Docker 用の**メモリを 4GB** 以上に設定する（Mac OS と Windows 版の Docker の場合は、既定値が **2GB** に設定されているので **4GB** に変更する）

Let's Try

それでは、これを試してみましょう。ここでは、**Mac OS X**（**Docker CE for Mac**）を利用した場合の手順を説明しますが、**Linux** 環境でもほとんど同じように試すことができます。Mac OS でも、Linux でも、**Docker コマンドを 2 つ実行**するだけで、SQL Server 2017 を起動することができ、非常に簡単に試すことができるので、ぜひ試してみてください。

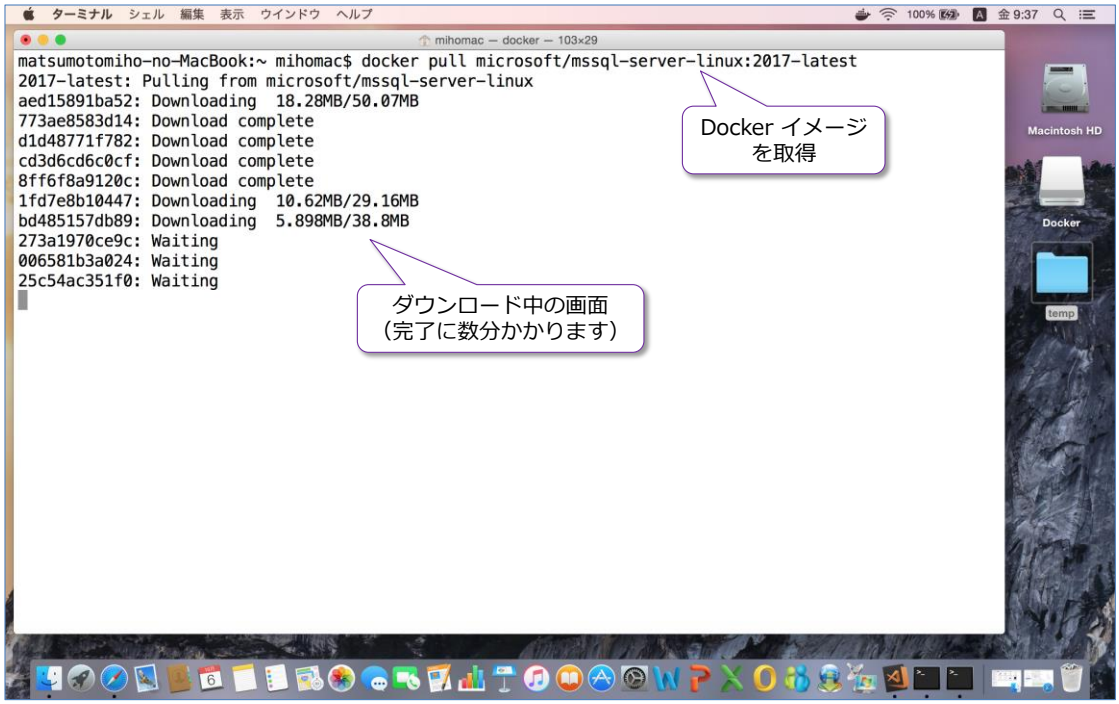
1. **Mac OS X** では、「**Docker CE for Mac**」をインストールすることで、Docker を利用することができますが、既定では Docker 用のメモリが **2GB** に設定されています。まずは、これを **4GB** 以上に変更する必要があります。メモリを変更するには、次のように「**Docker アイコン**」をクリックして、［**References...**］をクリックします。



［**References**］ダイアログが表示されたら、［**Advanced**］ページを開いて、［**Memory**］セクションのスライダーを動かして「**4GB**」に変更します。変更後、［**Apply & Restart**］ボタンをクリックします。これで Docker が再起動されて、4GB の設定が有効になります。

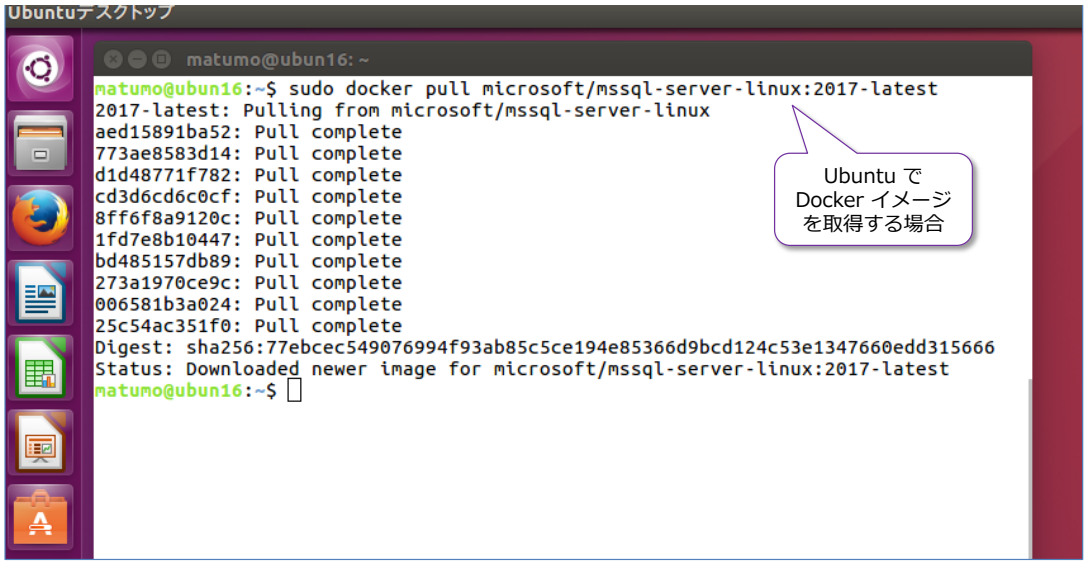
2. Docker の再起動が完了したら、次は、**SQL Server 2017** の **Docker イメージを取得**（pull）します。これを行うには、**ターミナル**を起動して、次のように **Docker** コマンドの **pull** で、イメージ名に「**microsoft/mssql-server-linux:2017-latest**」を指定します。

```
docker pull microsoft/mssql-server-linux:2017-latest
```



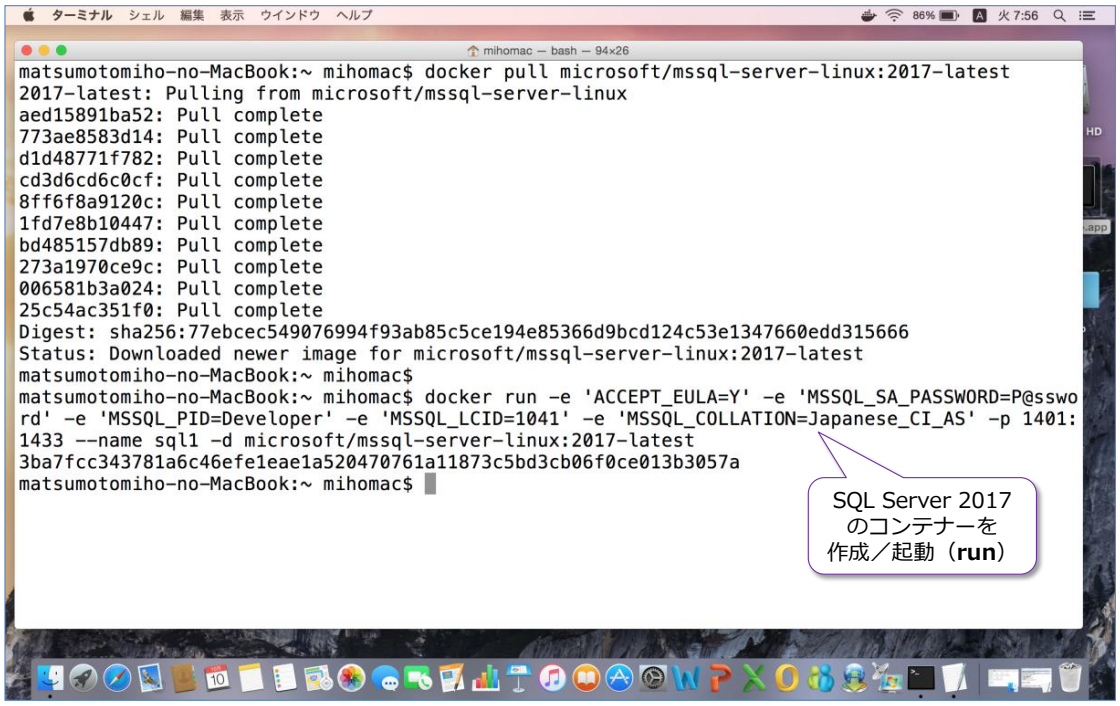
イメージのダウンロードには数分かかるので、完了するまで少し待ちます。

なお、**Linux** を利用して、イメージを取得する場合には、必要に応じて、先頭に「**sudo**」を付けて実行するようにします。



3. SQL Server 2017 のイメージの取得が完了したら、次は**イメージからコンテナーを作成／起動（run）**します。これを行うには、次のように **Docker** コマンドを実行します。以下のコマンドには改行が入っていますが、実際にコマンドを入力するときは、改行せずに 1 行で記述するようにしてください（あるいは、改行を入れる場合には、行末に**バック スラッシュ**を入れるようにしてください）。

```
docker run -e 'ACCEPT_EULA=Y'
 -e 'MSSQL_SA_PASSWORD=任意の複雑なパスワード'
 -e 'MSSQL_PID=Developer'
 -e 'MSSQL_LCID=1041' -e 'MSSQL_COLLATION=Japanese_CI_AS'
 -p 1401:1433 --name sql1
 -d microsoft/mssql-server-linux:2017-latest
```



-e オプションでは、SQL Server 2017 に関する**環境変数**を設定（どのように動作させるのかを指定）しますが、「**-e '変数名=値'**」という形で、**単一引用符**で囲む必要があります（なお、Windows 版の Docker を利用している場合には、単一引用符ではなく、二重引用符で囲むようにします）。

環境変数の「**ACCEPT_EULA=Y**」は、使用許諾契約書（ライセンス条項）の確認になりますが、同意する場合には **Y** を指定します。

「**MSSQL_SA_PASSWORD=**」では、SQL Server の管理者アカウントである「**sa**」に対する任意のパスワードを設定しますが、8 文字以上の複雑なパスワード（大文字、小文字、数字、記号の 4 種類のうち、いずれかの 3 種類の文字を含めるもの）を指定する必要があります。

「**MSSQL_PID=Developer**」では、利用する SQL Server のエディションとして、**Developer**（開発者向けのエディション）を指定しています。有償のエディションを購入している場合には、**Enterprise** や **Standard**、**Web** などを指定することができます。

「**MSSQL_LCID=1041**」では、利用する言語（ロケール ID）を「**日本語**」（**1041**）に設定しています（これは省略可能な環境変数ですが、省略した場合は、英語：**1033** に設定されます）。

「**MSSQL_COLLATION=Japanese_CI_AS**」では、SQL Server の**照合順序**（**Collation**）を設定しますが、「**Japanese_CI_AS**」は日本語版の SQL Server を利用する場合の既定値になっています（これも省略可能な環境変数ですが、省略した場合には、英語版の SQL Server の既定値である **SQL_Latin1_General_CP1_CI_AS** に設定されます）。

「**-p 1401:1433**」では、SQL Server のデータベース エンジンで利用する**ポート番号**を設定しますが、**1401:1433** に設定する場合は、コンテナー内の SQL Server は、**1433** ポートでリッスンして、ホスト側のポートは **1401** を利用します。したがって、別のマシンから、コンテナー内の SQL Server にアクセスするには、ホスト側のポート番号を利用するので、**1401** を指定することになります。なお、「**1433:1433**」と指定した場合には、ホスト上の **1433** ポートを利用して、アクセスさせることもできます（なお、**1433** は、SQL Server における既定のポート番号になります）。

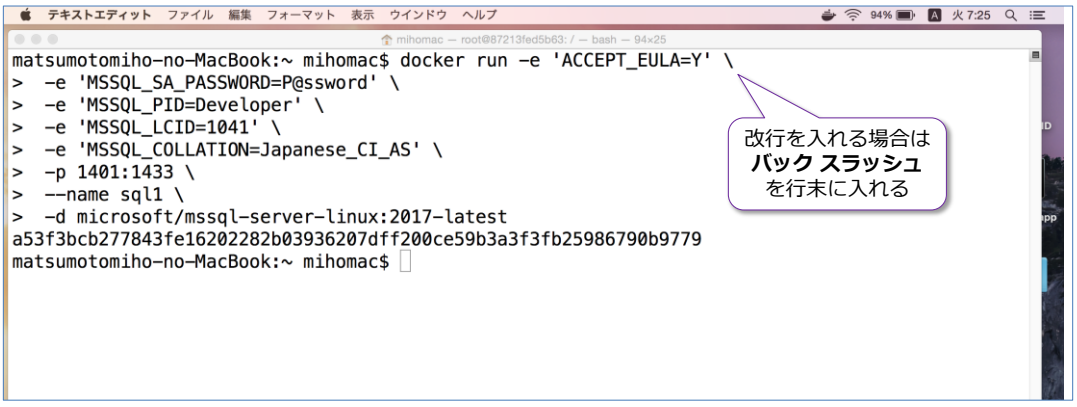
「**--name sql1**」（**name** の前はハイフンが 2 つであることに注意）では、作成するコンテナーに対して任意の名前を付けることができます（ここでは **sql1** という名前を指定していますが、別の名前を指定しても大丈夫です）。

「**-d microsoft/mssql-server-linux:2017-latest**」では、前の手順で取得した SQL Server 2017 の Docker イメージの名前を指定します。

以上で、SQL Server 2017 の起動が完了（コンテナーの作成と起動が完了）です。

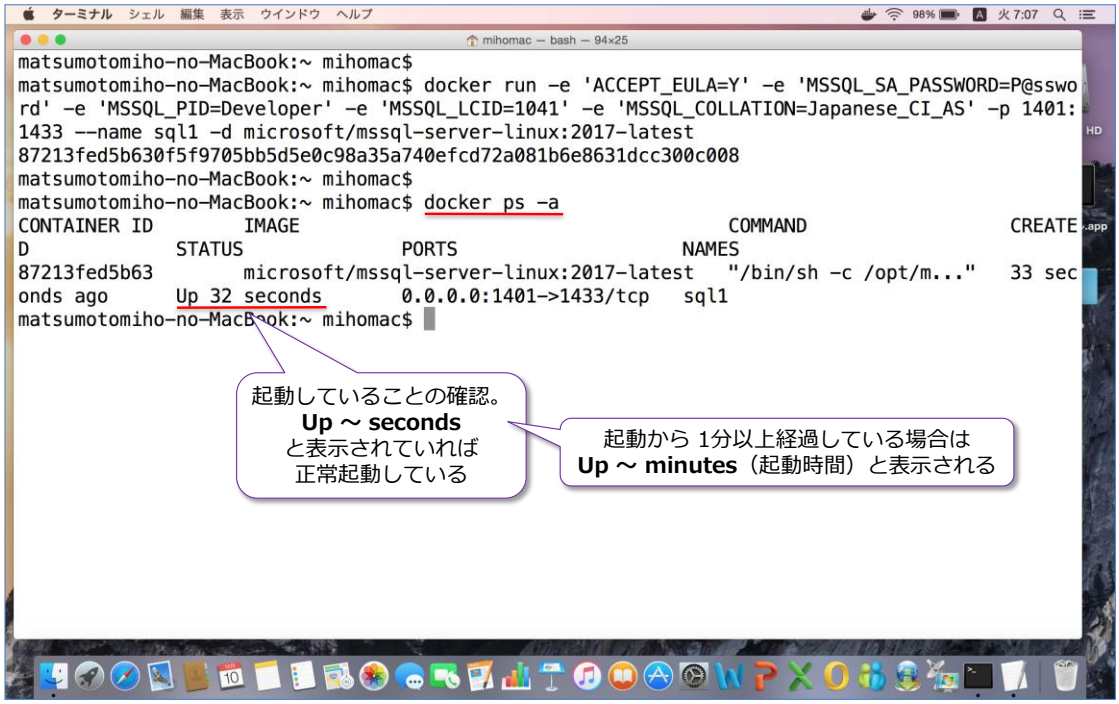
Note： Docker コマンドを複数行で記述する場合

Docker コマンドを、1 行ではなく、改行を入れて複数行で記述したい場合には、次のように**バック スラッシュ**を行末に入れるようします。

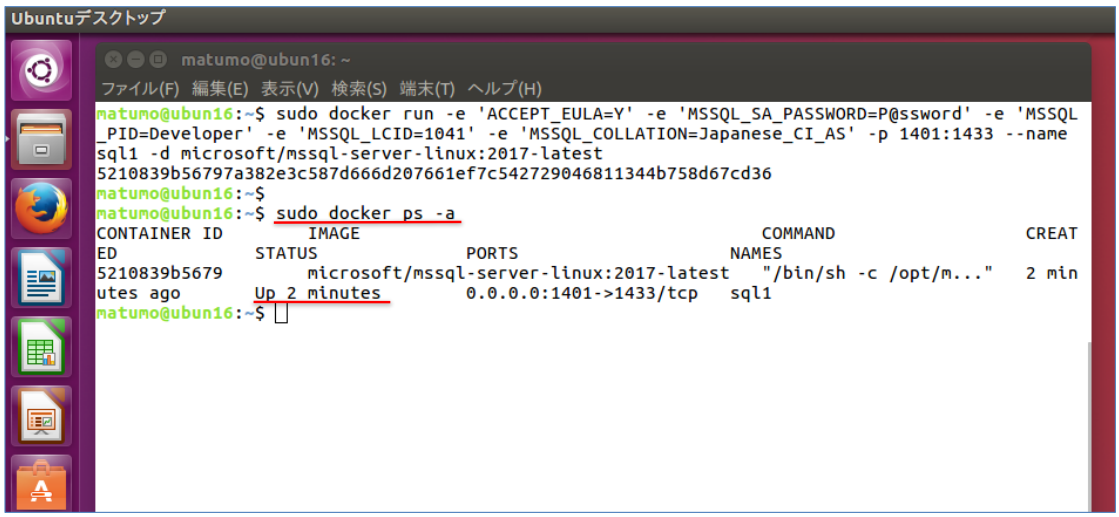


4. 次に、コンテナーが起動していることを確認するために、次のように **Docker** コマンドを実行します。

```
docker ps -a
```



Ubuntu の場合の画面

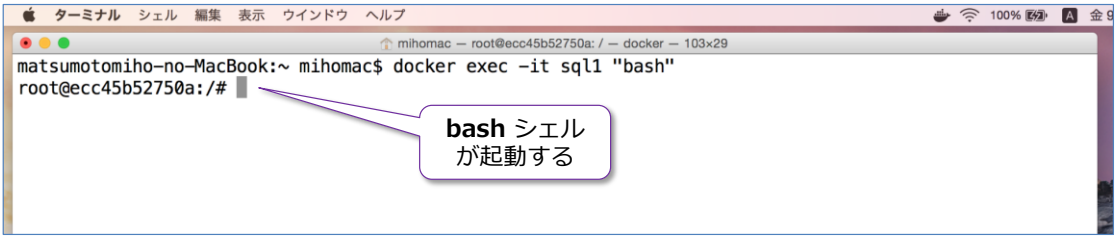


STATUS（状態）に **Up** ～ **seconds** や **Up** ～ **minutes** などと表示されれば、コンテナーが正常に起動しています（**Up** は起動していること、seconds や minutes で起動時間を確認できます）。

SQL Server への接続（sqlcmd ツール）

次に、Docker コンテナー内の **SQL Server 2017 に接続**してみましょう。SQL Server に接続するには、**sqlcmd** ツールを利用しますが、このツールを利用するには、まず、次のように **Docker** コマンドを実行（**exec**）して、コンテナー内の **bash** シェルを起動するようにします。

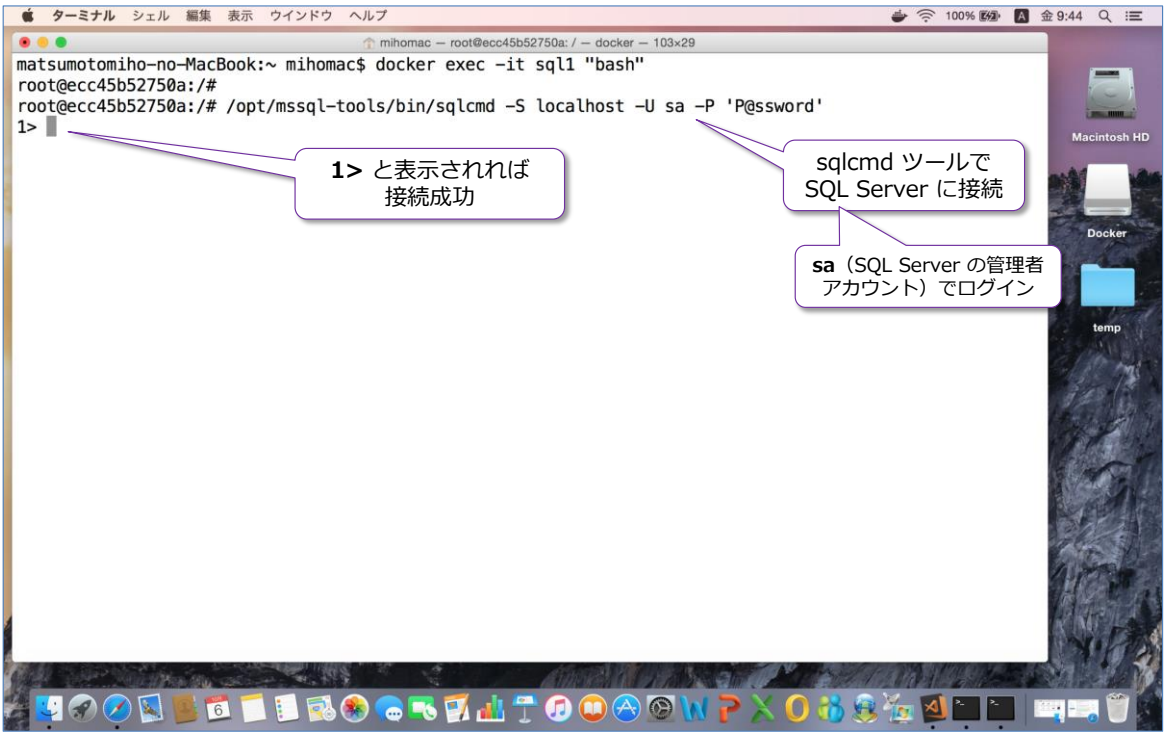
```
docker exec -it sql1 "bash"
```



「**-it**」には、**run** をしたときに「**--name**」で指定したコンテナー名の「**sql1**」を与えて、「"**bash**"」と記述することで、コンテナーの **bash** シェルを起動することができます。

bash シェルが起動したら、次に **sqlcmd** ツール（SQL Server を操作するためのコマンドライン ツール）を利用して、SQL Server に接続してみます。このツールは、コンテナー内の「**/opt/mssql-tools/bin**」ディレクトリに格納されているので、次のように実行できます。

```
/opt/mssql-tools/bin/sqlcmd -S localhost -U sa -P 'saに設定したパスワード'
```

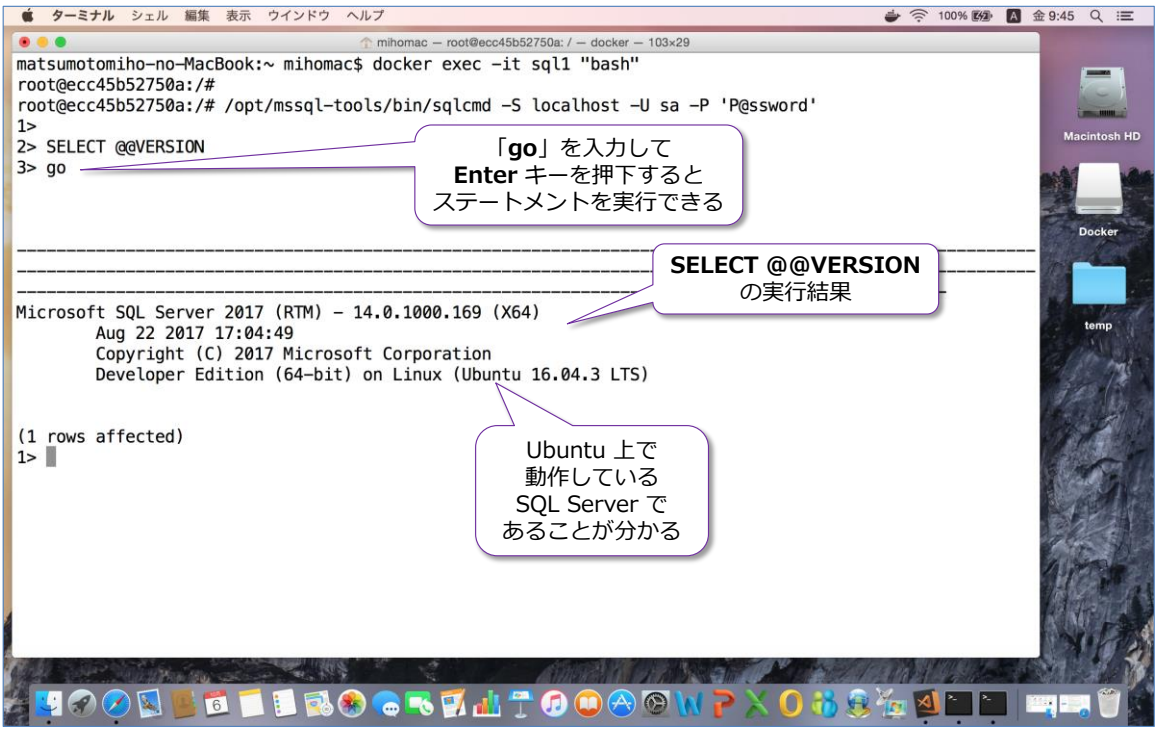


-S オプションでは、接続先となる SQL Server を指定しますが、コンテナー内（自分自身）の SQL Server に接続するので「**localhost**」と指定できます。**-U** オプションでは、接続ユーザーを指定しますが、SQL Server の管理者アカウントである「**sa**」を指定して、**-P** オプションでは、前の手順で Docker run したときに設定した **sa** のパスワードを入力します。

接続が完了すると「**>1**」と表示されて、任意の Transact-SQL ステートメントが実行できるようになります。

次に、SQL Server のバージョンを取得することができる「**@@VERSION**」関数を利用してみましょう。次のように **SELECT** ステートメントを記述して、**Enter** キーで改行し、次に「**go**」を付けて **Enter** キーを押下します。

```
SELECT @@VERSION
go
```



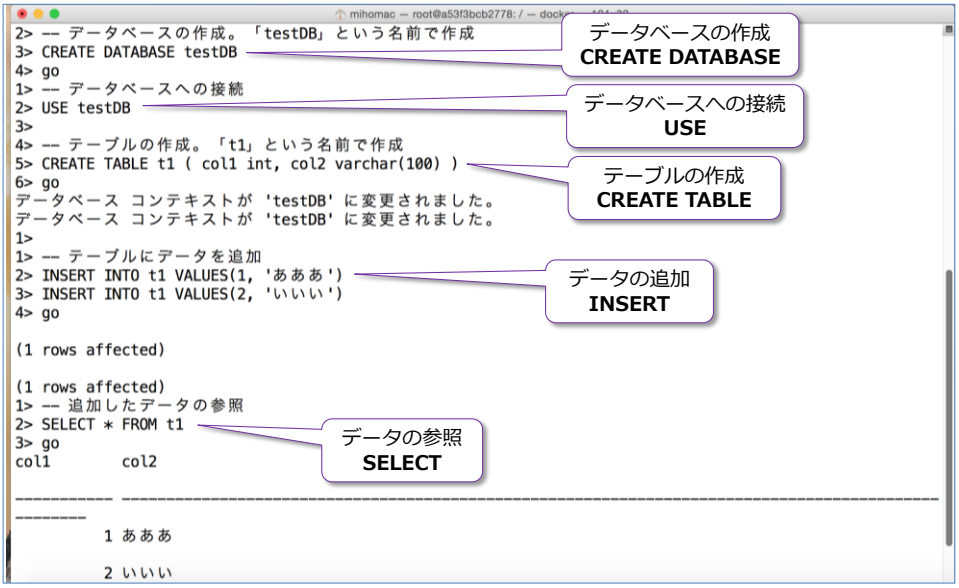
sqlcmd ツールでは、「**go**」を記述することで、そこまでに記述した Transact-SQL ステートメントを実行することができます。

データベースやテーブルの作成例

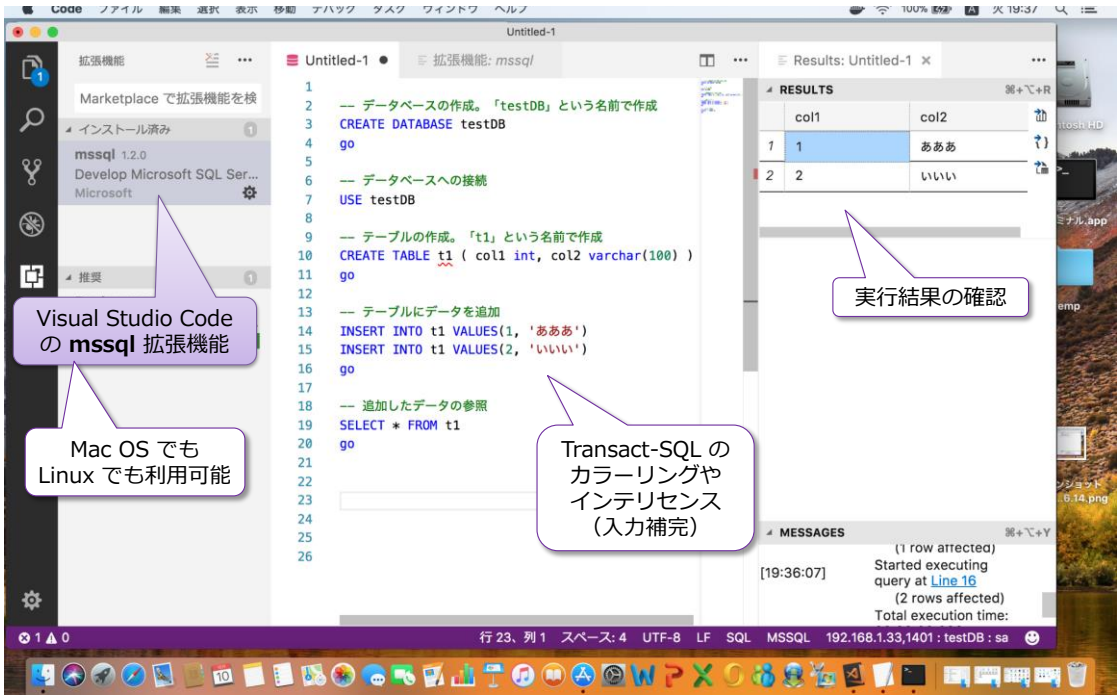
SQL Server 上にデータベースを作成（**CREATE DATABASE**）したり、その中にテーブルを作成（**CREATE TABLE**）したりするには、次のように記述します。

```
-- データベースの作成。「testDB」という名前で作成
CREATE DATABASE testDB
go
-- データベースへの接続。SQL Server では USE でデータベースに接続する
USE testDB

-- テーブルの作成。「t1」という名前で作成
CREATE TABLE t1 ( col1 int, col2 varchar(100) )
go
```

```
-- テーブルにデータを追加
INSERT INTO t1 VALUES(1, 'あああ')
INSERT INTO t1 VALUES(2, 'いいい')
go

-- 追加したデータの参照
SELECT * FROM t1
go
```



このように **sqlcmd** ツールを利用すれば、SQL Server 2017 に対して Transact-SQL ステートメントを実行できるようになります。ただ、このツールは、コマンドライン ツールなので使いづらい部分もあります。そこで、お勧めになるツールが、**Mac OS** や **Linux** でも利用することができる **Visual Studio Code** の **mssql** 拡張機能です。

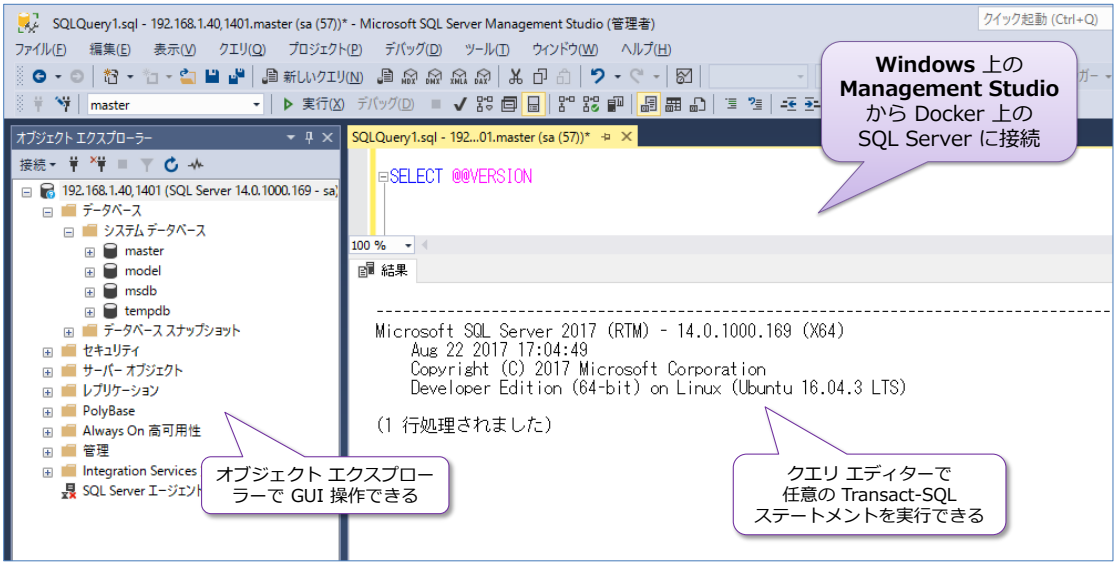


Visual Studio Code は、無償で利用できる開発ツールで、Windows はもちろんのこと、Mac OS でも Linux でも利用することができます。これを利用すれば、さまざまな言語（Java や Python、C#、node.js、ASP.NET など）を利用して、アプリケーション開発ができるので、SQL Server にアクセスするアプリケーションを開発する際にもお勧めのツールになります。

Visual Studio Code には、**mssql** 拡張機能をインストールすることで、上の画面のように、Transact-SQL ステートメントをグラフィカルに実行できるようになります（カラーリングやインテリセンスが効きます）。Visual Studio Code のインストール方法や、mssql 拡張機能の利用方法については、本章の Step 2.4 で詳しく説明します。

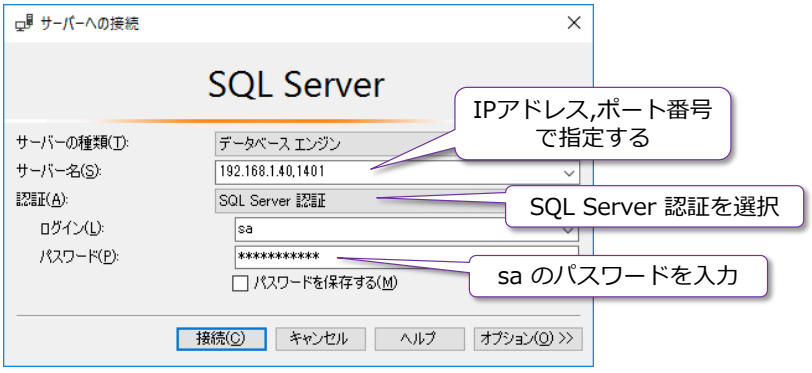
Note： Windows の Management Studio から Docker 上の SQL Server に接続

Docker 上の SQL Server コンテナーには、Windows 上の **Management Studio** ツール（SQL Server の管理ツール）から接続することも可能です。



これを利用すれば、Windows 上の SQL Server の操作と同様、オブジェクト エクスプローラーを利用して GUI で SQL Server を操作することができます。

Management Studio で Docker 上の SQL Server に接続する際には、次のようにポート番号に **1401** を指定するようにします（ホスト側でファイアウォールを有効化している場合には、ファイアウォールで **1401** ポートを解放しておく必要があります）。



「**サーバー名**」に「**IP アドレス,ポート番号**」と入力して、ホストの IP アドレスに続けて**カンマ**を記述し、それに続けてポート番号「**1401**」を指定します（これは、コンテナーを **run** したときに「**-p 1401:1433**」という形で指定したポート番号です）。

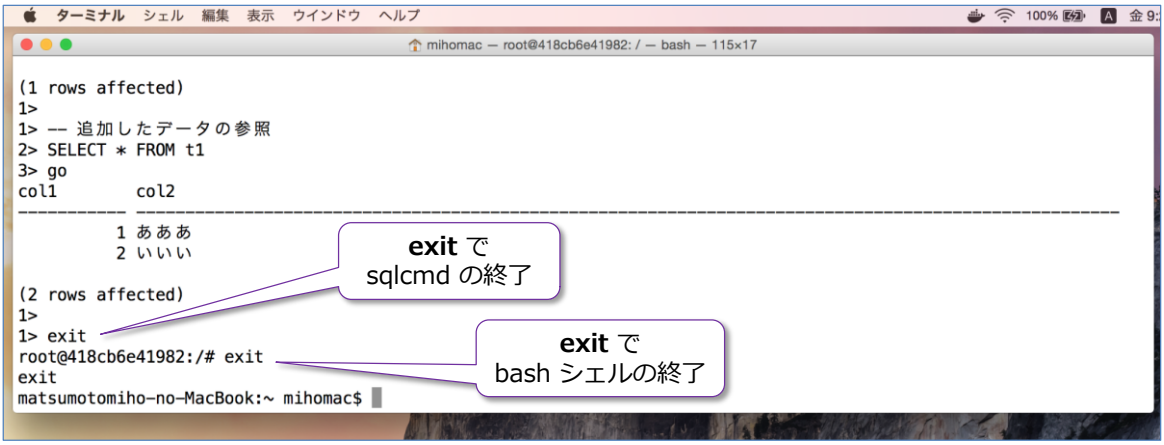
「**認証**」では「**SQL Server 認証**」を選択して、「**ログイン**」に「**sa**」、「**パスワード**」には sa のパスワードを入力すれば、SQL Server に接続することができます。

あとは、Windows 上の SQL Server を操作するのと同じように Docker 上の SQL Server を操作することができます。

sqlcmd の終了、bash シェルの終了 ～exit～

sqlcmd ツールを終了して、bash シェルに戻るには、**exit** と記述します。

```
exit
```



また、コンテナー内の **bash** シェルを終了する場合にも、**exit** と記述します。

コンテナーの停止、削除

コンテナーを停止（**stop**）したい場合には、次のように **Docker** コマンドを実行します。

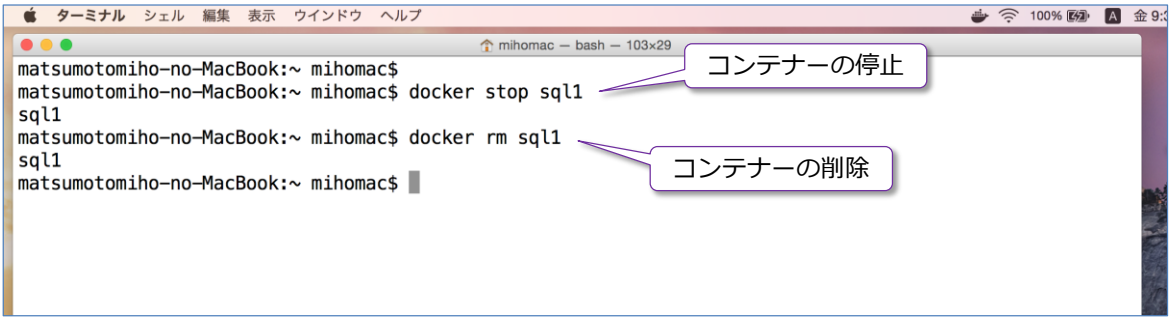
```
docker stop sql1
```

なお、停止したコンテナーを再開（開始）するには、次のように **start** を指定します。

```
docker start sql1
```

コンテナーを削除したい場合には、次のように **rm** を指定しますが、これを実行するには、事前にコンテナーが停止（stop）済みである必要があります。

```
docker rm sql1
```



以上のように、SQL Server 2017 は、**Docker** イメージが提供されるようになったので、**Mac OS** や **Linux** 上でも簡単に試せるようになりました。SQL Server 2017 をインストールすることなく、SQL Server を利用することができるので、開発環境を作成する場合などに大変便利です。

Docker イメージ（コンテナー）に対して行った修正（データベースの作成や、既存のデータベースのリストアなど）は、**commit**（Docker コマンドの commit を実行）をすることで**新しいイメージ**として作成（SQL Server 2017 もデータベースも全て構成された状態のイメージを作成）することができるので、これを Docker Hub や Docker Registory、Azure Container Registory などのイメージの共有が可能なレジストリに配置しておけば、社内の開発メンバーに、同じデータベース環境を簡単に共有することができます（レジストリから pull するだけで、開発に必要なデータベースをすぐに利用することができます）。

Note： イメージを削除したい場合

ダウンロードした SQL Server 2017 イメージを削除したい場合には、次のようにイメージの一覧を取得して、イメージ ID を確認します。

```
docker images
```

「**microsoft/mssql-server-linux:2017-latest**」のイメージ ID を確認したら、次のように **rmi** を指定することで、イメージを削除することができます。

```
docker rmi イメージ ID
```

2.2 Linux 環境への SQL Server 2017 のインストール

次に、Docker ではなく、Linux 環境に直接 SQL Server 2017 をインストールする方法を説明します。SQL Server 2017 がサポートしているプラットフォームは、次のとおりです。

- **Red Hat Enterprise Linux 7.3** または **7.4** Workstation, Server, and Desktop (ファイル システム: XFS または EXT4)
- **SUSE Enterprise Linux Server v12 SP2**（ファイル システム: EXT4）
- **Ubuntu 16.04 LTS**（ファイル システム: EXT4）

インストールのためのハードウェア要件は、次のとおりです。

- メモリ： **3.25GB**
- ディスク領域： **6GB**
- CPU： **x64** と互換性のあるもののみで、**2GHz**、**2 コア**以上

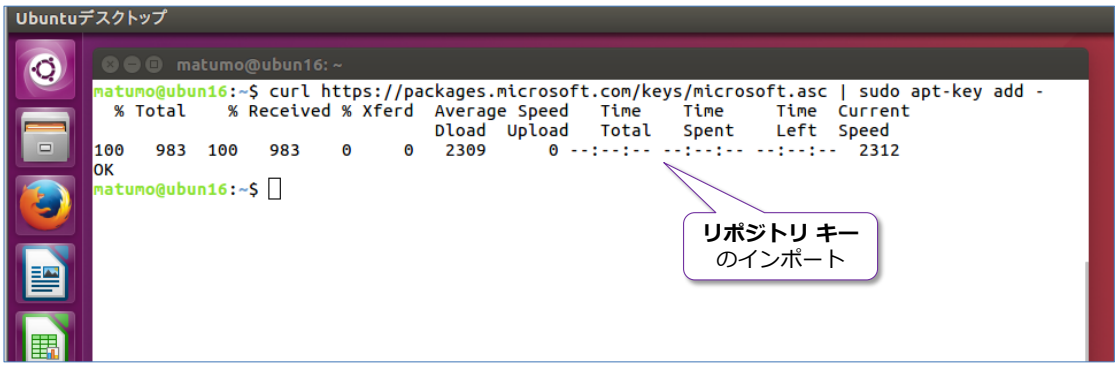
Let's Try

それでは、Linux 環境への SQL Server 2017 のインストールを試してみましょう。ここでは、**Ubuntu 16.04 LTS** 上に SQL Server 2017 をインストールする手順を例に説明しますが、**RHEL**（Red Hat Enterprise Linux）や **SUSE Enterprise Linux** を利用する場合でもほとんど同じようにインストールすることができます（パッケージ マネージャとして apt-get を利用するか、yum を利用するか、zypper を利用するかの違い程度です）。

Ubuntu への SQL Server 2017 のインストールは、**5 個のコマンド**を実行するだけで簡単に完了するので、ぜひ試してみてください。

1. Ubuntu に SQL Server 2017 をインストールするには、まず、次のようにリポジトリ キーをインポートします。

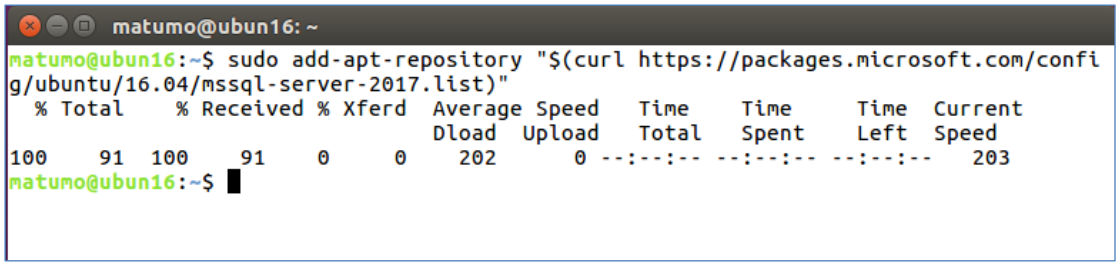
```
curl https://packages.microsoft.com/keys/microsoft.asc | sudo apt-key add -
```



2. 次に、リポジトリを登録します（以下のコマンドには改行が入っていますが、実際にコマンド

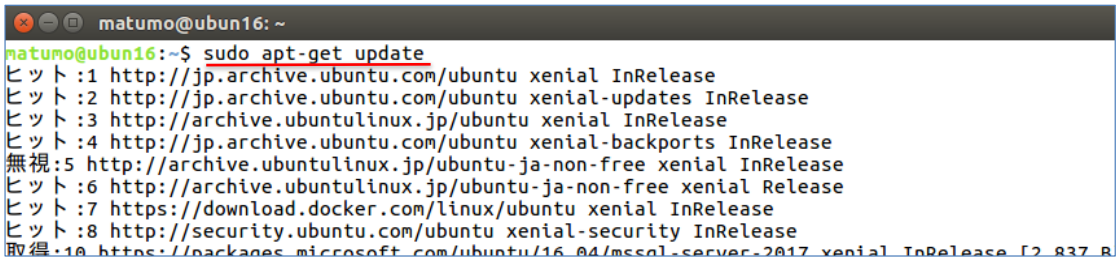
を入力するときは、改行せずに 1 行で記述するようにしてください)。

```
sudo add-apt-repository "$(curl https://packages.microsoft.com/config/ubuntu/16.04/mssql-server-2017.list)"
```



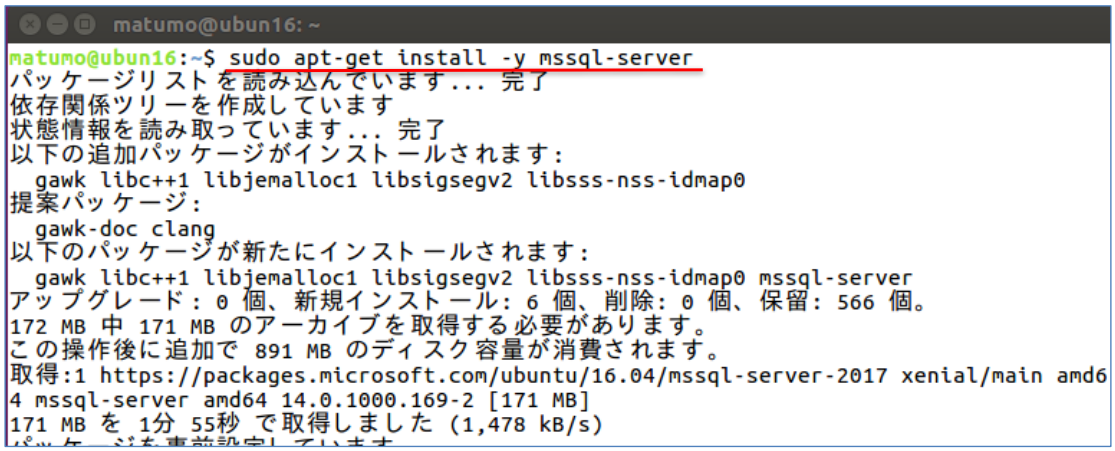
3. 次に、パッケージのアップデートを実行します。

```
sudo apt-get update
```

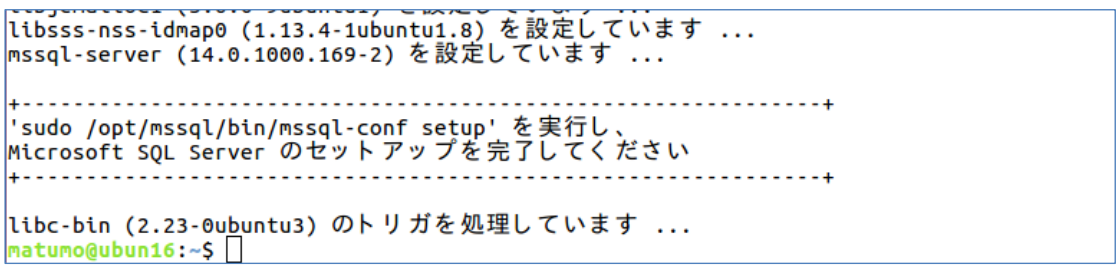


4. パッケージのアップデートが完了したら、次は SQL Server 2017 のインストールです。これを行うには、次のように「**mssql-server**」を指定します。

```
sudo apt-get install -y mssql-server
```

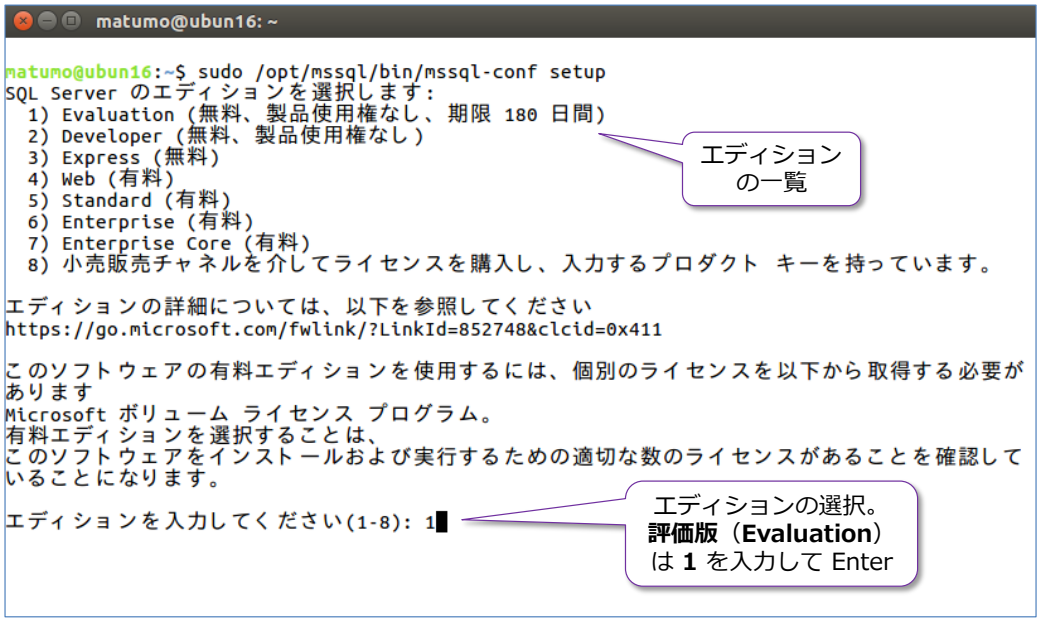


:



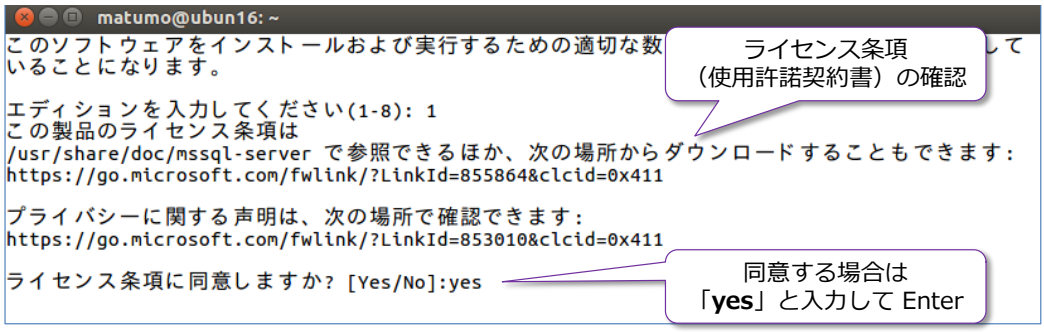
5. 最後に、**SQL Server のセットアップ**（エディションの選択や言語の選択、管理者アカウントへのパスワード設定など）を行います。これを行うには、次のように「**/opt/mssql/bin**」ディレクトリの「**mssql-conf setup**」を実行します。

```
sudo /opt/mssql/bin/mssql-conf setup
```

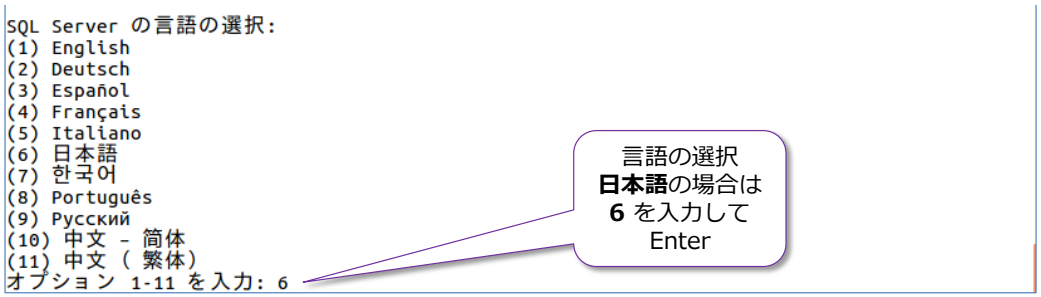


最初にエディションを選択しますが、**評価版**（**Evaluation**）を利用する場合は「**1**」を入力して、**Enter** キーを押下します。

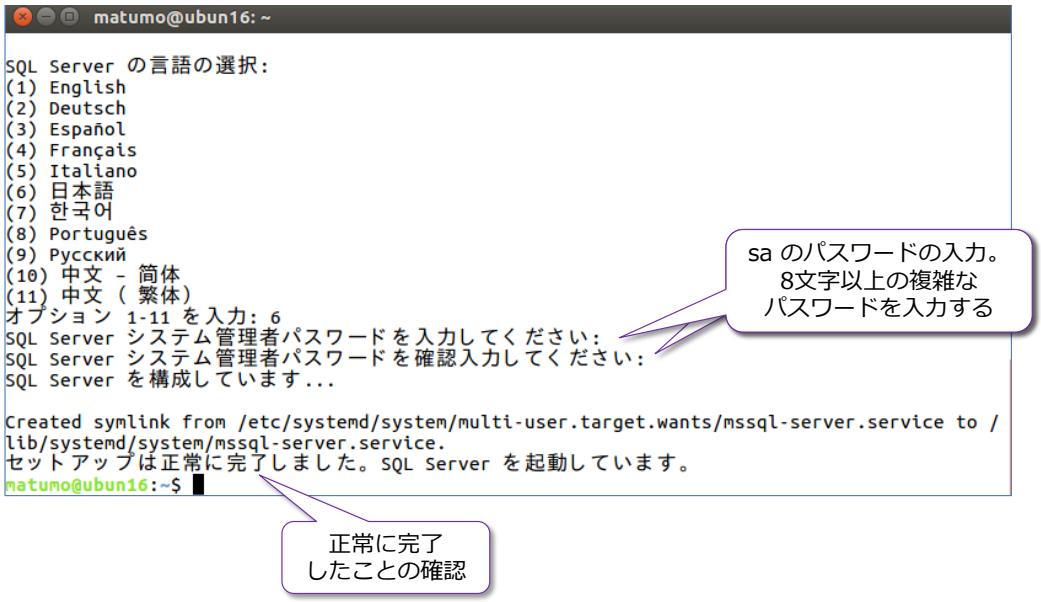
次に、**ライセンス条項**（使用許諾契約書）に関する情報が表示されるので、内容を確認した上で、同意する場合は「**yes**」と入力します。



次に、**言語の選択**が表示されるので、**日本語**を利用する場合は「**6**」を入力します。



最後に **SQL Server の管理者アカウント**である「**sa**」に対する任意のパスワードを設定しますが、パスワードは 8 文字以上の複雑なもの（大文字、小文字、数字、記号の 4 種類のうち、いずれかの 3 種類の文字を含めるもの）に設定する必要があります。

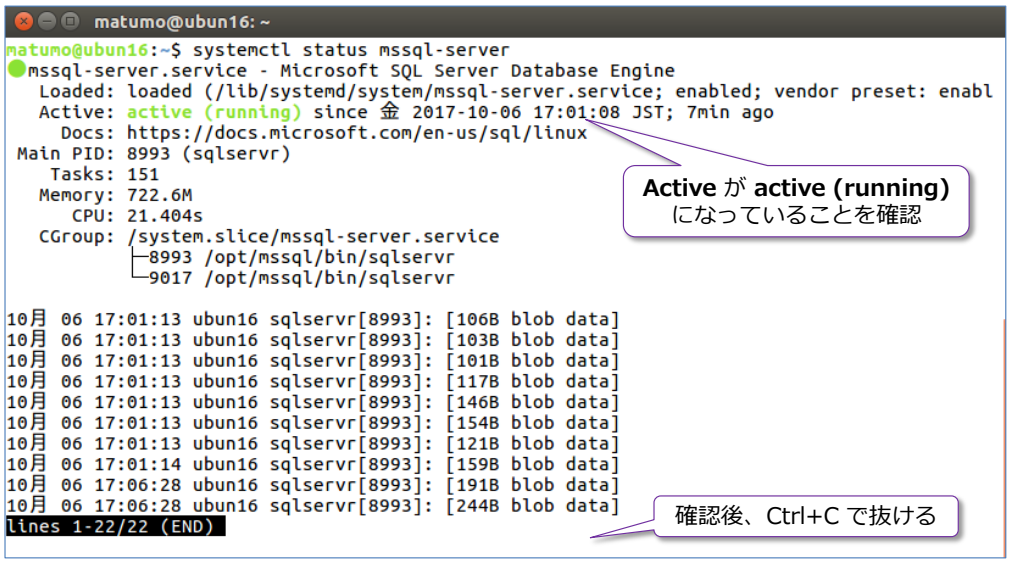


以上で、最後に「**セットアップは正常に完了しました**」と表示されれば、SQL Server のインストールが完了です。

インストールの確認

1. インストールが完了したことを確認するには、次のように **systemctl** で「**mssql-server**」の **status** を確認します。

```
systemctl status mssql-server
```



Active が **active (running)** になっていれば、SQL Server は正常に動作しています。

sqlcmd ツールのインストール

次に、SQL Server に接続するために、**sqlcmd** ツールをインストールします。

1. **sqlcmd** ツールをインストールするには、まず、リポジトリ キーをインポートします。

```
curl https://packages.microsoft.com/keys/microsoft.asc | sudo apt-key add -
```

```
matumo@ubun16: ~
matumo@ubun16:~$ curl https://packages.microsoft.com/keys/microsoft.asc | sudo apt-key add -
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   983  100   983    0     0   2122      0 --:--:-- --:--:-- --:--:--  2118
OK
matumo@ubun16:~$
```

2. 次に、リポジトリを登録します。

```
sudo add-apt-repository "$(curl https://packages.microsoft.com/config/ubuntu/
16.04/prod.list)"
```

```
matumo@ubun16: ~
matumo@ubun16:~$ sudo add-apt-repository "$(curl https://packages.microsoft.com/config/ubuntu/16.0
4/prod.list)"
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100    79  100    79    0     0    175      0 --:--:-- --:--:-- --:--:--   175
matumo@ubun16:~$
```

3. 次に、パッケージのアップデートを実行します。

```
sudo apt-get update
```

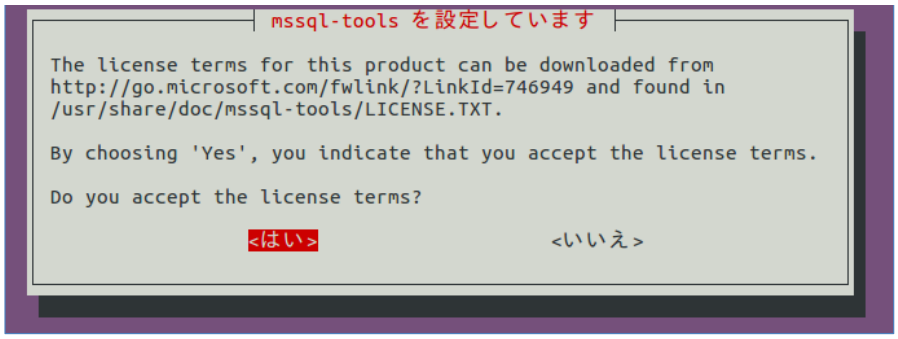
```
matumo@ubun16: ~
matumo@ubun16:~$ sudo apt-get update
ヒット:1 http://jp.archive.ubuntu.com/ubuntu xenial InRelease
取得:2 http://jp.archive.ubuntu.com/ubuntu xenial-updates InRelease [102 kB]
ヒット:3 http://archive.ubuntulinux.jp/ubuntu xenial InRelease
無視:4 http://archive.ubuntulinux.jp/ubuntu-ja-non-free xenial InRelease
ヒット:5 http://archive.ubuntulinux.jp/ubuntu-ja-non-free xenial Release
取得:6 http://jp.archive.ubuntu.com/ubuntu xenial-backports InRelease [102 kB]
ヒット:7 https://download.docker.com/linux/ubuntu xenial InRelease
取得:9 http://security.ubuntu.com/ubuntu xenial-security InRelease [102 kB]
ヒット:10 https://packages.microsoft.com/ubuntu/16.04/mssql-server-2017 xenial InRelease
取得:11 https://packages.microsoft.com/ubuntu/16.04/prod xenial InRelease [2,845 B]
取得:12 https://packages.microsoft.com/ubuntu/16.04/prod xenial/main amd64 Packages [17.0 kB]
326 kB を 1秒 で取得しました (226 kB/s)
*** Error in `appstreamcli': double free or corruption (fasttop): 0x000000000220e590 ***
======= Backtrace: =========
/lib/x86_64-linux-gnu/libc.so.6(+0x77725)[0x7f798a3ec725]
/lib/x86_64-linux-gnu/libc.so.6(+0x7ff4a)[0x7f798a3f4f4a]
/lib/x86_64-linux-gnu/libc.so.6(cfree+0x4c)[0x7f798a3f8abc]
/usr/lib/x86_64-linux-gnu/libappstream.so.3(as_component_complete+0x439)[0x7f798a770d19]
```

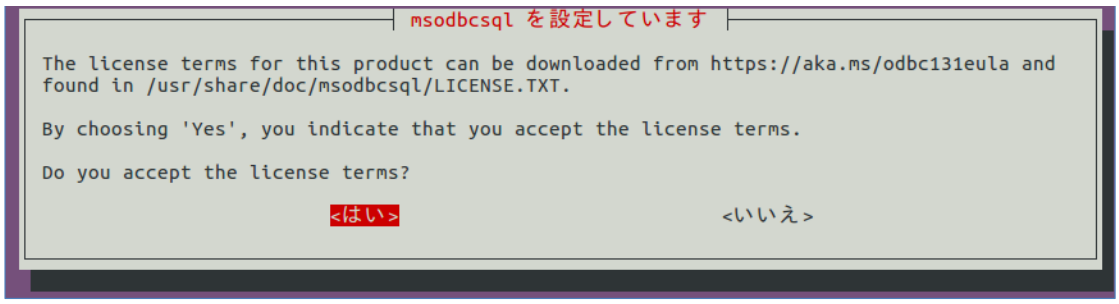
4. パッケージのアップデートが完了したら、次は **sqlcmd** ツールのインストールを行います。これは、次のように「**mssql-tools**」と「**unixodbc-dev**」を指定します。

```
sudo apt-get install -y mssql-tools unixodbc-dev
```

```
matumo@ubun16: ~

matumo@ubun16:~$ sudo apt-get install -y mssql-tools unixodbc-dev
パッケージリストを読み込んでいます... 完了
依存関係ツリーを作成しています
状態情報を読み取っています... 完了
以下の追加パッケージがインストールされます:
  autotools-dev libltdl-dev libodbc1 libtool msodbcsql odbcinst odbcinst1debian2 unixodbc
提案パッケージ:
  libtool-doc libmyodbc odbc-postgresql tdsodbc unixodbc-bin autoconf automaken gfortran
  | fortran95-compiler gcj-jdk
以下のパッケージが新たにインストールされます:
  autotools-dev libltdl-dev libodbc1 libtool msodbcsql mssql-tools odbcinst odbcinst1debian2
  unixodbc unixodbc-dev
アップグレード: 0 個、新規インストール: 10 個、削除: 0 個、保留: 566 個。
5,770 kB のアーカイブを取得する必要があります。
この操作後に追加で 4,704 kB のディスク容量が消費されます。
取得:1 http://jp.archive.ubuntu.com/ubuntu xenial/main amd64 autotools-dev all 20150820.1 [39.8 kB
]
取得:2 http://jp.archive.ubuntu.com/ubuntu xenial/main amd64 libltdl-dev amd64 2.4.6-0.1 [162 kB]
取得:3 http://jp.archive.ubuntu.com/ubuntu xenial/main amd64 libodbc1 amd64 2.3.1-4.1 [180 kB]
取得:4 http://jp.archive.ubuntu.com/ubuntu xenial/main amd64 libtool all 2.4.6-0.1 [193 kB]
```

インストールの途中では、次のように **license terms**（**使用許諾契約書**）の確認が求められるので、内容を確認した上で、同意する場合は「**はい**」を選択します。





以上で **sqlcmd** ツールのインストールが完了です。

5. **sqlcmd** ツールは、「**/opt/mssql-tools/bin**」ディレクトリに格納されているので、次のように実行することができます。

```
/opt/mssql-tools/bin/sqlcmd -S localhost -U sa -P 'saに設定したパスワード'
```

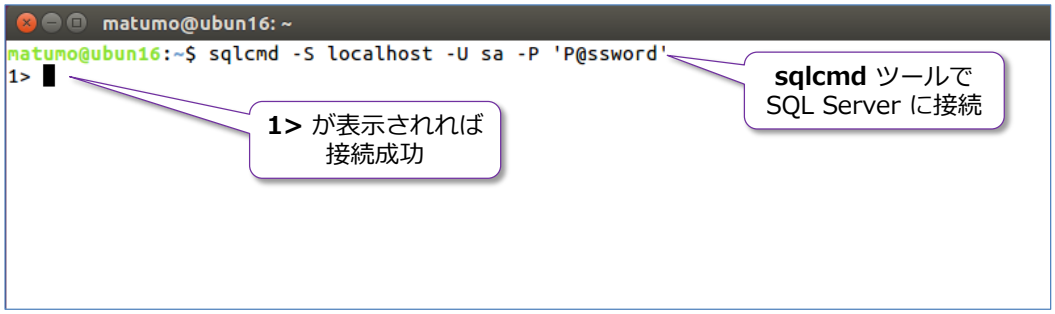
6. 上の手順のように毎回パスを記述するのが面倒な場合は、次のように **PATH** を設定しておくと便利です。

```
echo 'export PATH="$PATH:/opt/mssql-tools/bin"' >> ~/.bash_profile
echo 'export PATH="$PATH:/opt/mssql-tools/bin"' >> ~/.bashrc
source ~/.bashrc
```

```
matumo@ubun16: ~
matumo@ubun16:~$ echo 'export PATH="$PATH:/opt/mssql-tools/bin"' >> ~/.bash_profile
matumo@ubun16:~$ echo 'export PATH="$PATH:/opt/mssql-tools/bin"' >> ~/.bashrc
matumo@ubun16:~$ source ~/.bashrc
matumo@ubun16:~$
```

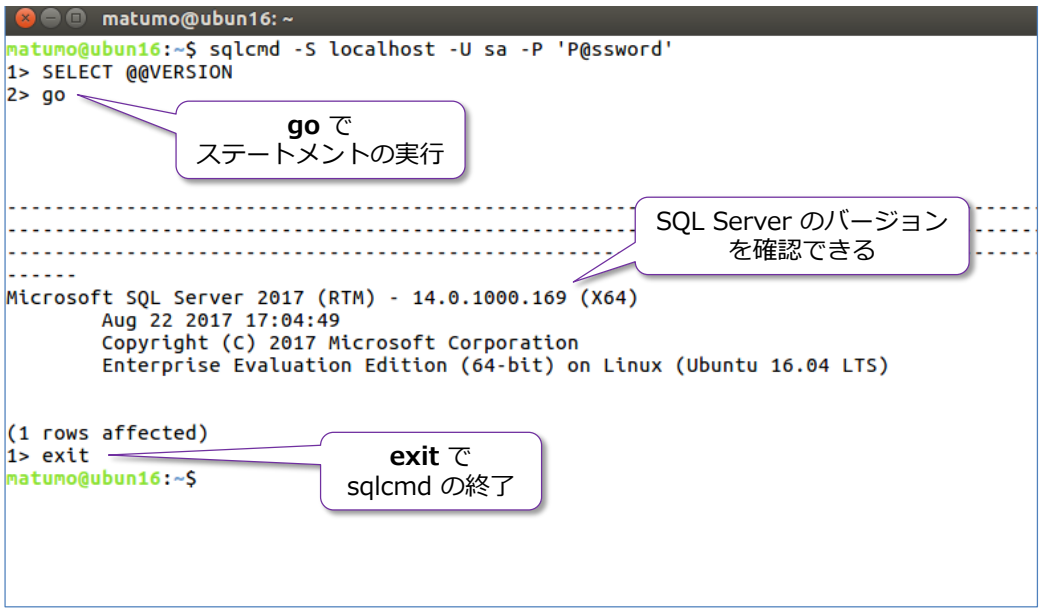
7. **PATH** を設定した後は、次のように **sqlcmd** と記述するだけで利用できるようになります。

```
sqlcmd -S localhost -U sa -P 'saに設定したパスワード'
```



sqlcmd で Transact-SQL ステートメントを実行するには「**go**」を付けます。

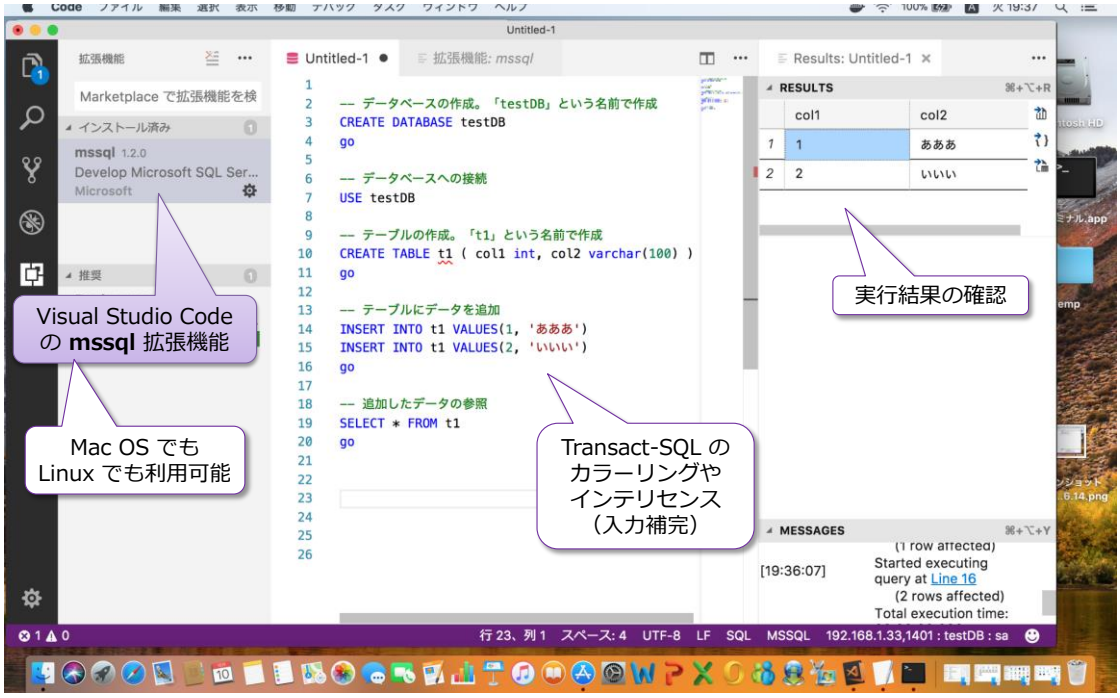
ここでは、SQL Server のバージョンを取得する SELECT ステートメントを「**SELECT @@VERSION**」と入力して、改行（Enter キー）、次に「**go**」を付けて、ステートメントを実行してみましょう。



sqlcmd を終了するには「**exit**」と入力します。

このように **sqlcmd** ツールを利用すれば、SQL Server 2017 に対して Transact-SQL ステートメントを実行できるようになります。また、合わせて、同じパス（**/opt/mssql-tools/bin**）には **bcp** ツールもインストールされるので、テキスト ファイル（CSV ファイルなど）を一括インポート（ファイルのデータをデータベース内のテーブルに格納）したい場合に利用できます。

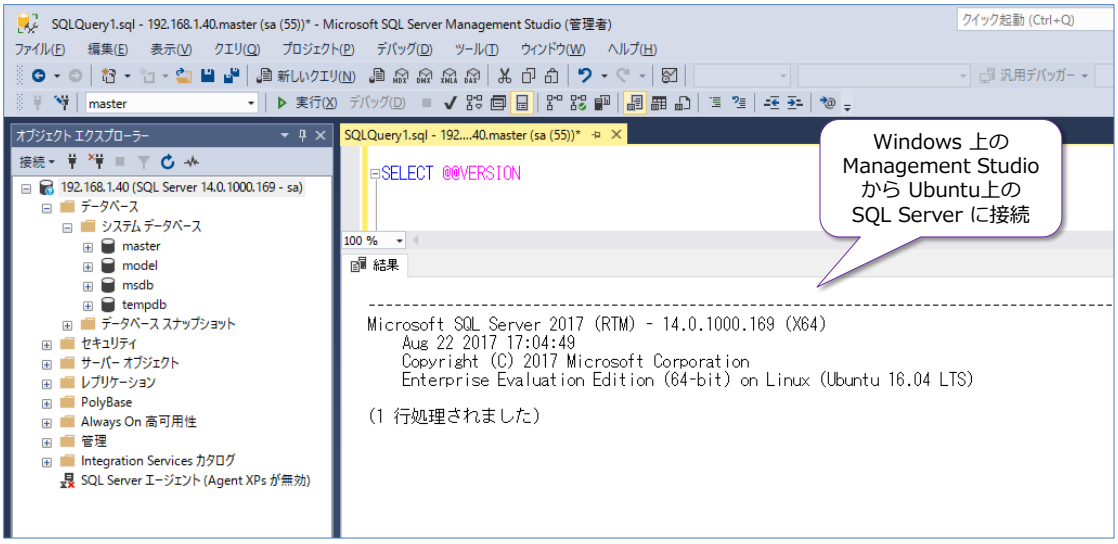
Docker のところで紹介しましたが、**sqlcmd** ツールは、コマンドライン ベースで使いづらい部分があるので、お勧めのツールなのが **Visual Studio Code** の **mssql 拡張機能**です



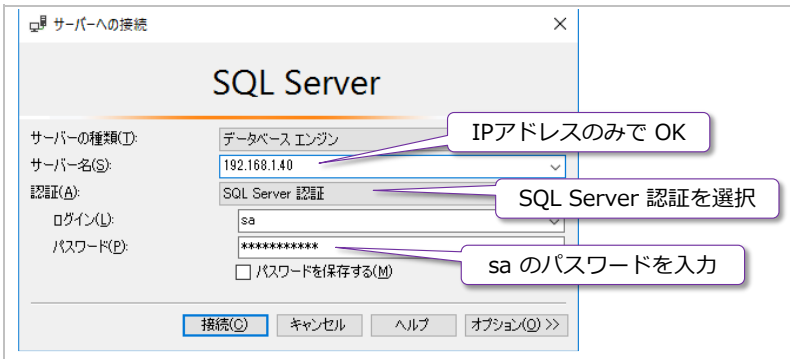
このツールの利用方法や、Visual Studio Code のインストール方法については、本書の Step 2.4 で詳しく説明します。

Note： Windows の Management Studio から Linux 上の SQL Server に接続

Linux 上の SQL Server 2017 には、Windows 上の **Management Studio** ツール（SQL Server の管理ツール）から接続することも可能です。



Management Studio を利用して Linux 上の SQL Server に接続する場合は、次のように行えます。



本文中で試したインストール手順では、SQL Server 2017 は、ポート番号「**1433**」(SQL Server の既定のポート番号) を利用するようにインストールされるので、「**サーバー名**」には IP アドレスを入力するだけで大丈夫です (ポート番号は、**1433** の場合は省略できます)。なお、Linux 上でファイアウォールを有効化している場合には、ファイアウォールで **1433** ポートを解放しておく必要があります。また、hosts や DNS を利用している場合には、IP アドレスではなく、ホスト名で接続することもできます。

あとは、Windows 上の SQL Server を操作するのと同じように Linux 上の SQL Server を GUI で操作することができます。

RHEL や SUSE への SQL Server 2017 のインストール

再掲になりますが、SQL Server 2017 がサポートしている Linux プラットフォームは、次のとおりです。

- **Red Hat Enterprise Linux 7.3** または **7.4** Workstation, Server, and Desktop (ファイル システム: XFS または EXT4)
- **SUSE Enterprise Linux Server v12 SP2** (ファイル システム: EXT4)
- **Ubuntu 16.04 LTS** (ファイル システム: EXT4)

Red Hat Enterprise Linux や SUSE Enterprise Linux Server にインストールする手順については、以下の URL が参考になります。

Red Hat Enterprise Linux への SQL Server 2017 のインストール
https://docs.microsoft.com/ja-jp/sql/linux/quickstart-install-connect-red-hat

SUSE Enterprise Linux Server への SQL Server 2017 のインストール
https://docs.microsoft.com/ja-jp/sql/linux/quickstart-install-connect-suse

2.3 SQL Server 2017 on Linux で利用できる機能／利用できない機能

ここでは、SQL Server 2017 on Linux で利用できる機能と利用できない機能について説明します。

SQL Server 2017 on Linux で利用できる機能

SQL Server on Linux で利用できる機能は、次のとおりです。

SQL Server データベース エンジンに関して

SQL Server on Linux では、データベースに関する**基本操作**（DB 作成、テーブル作成、データの追加／更新／削除、ビューやストアド プロシージャ、トリガー、インデックスの作成）ができることはもちろん、性能向上を実現できる**列ストア インデックス**や、**インメモリ OLTP**、**データ パーティション**、**データ圧縮**も利用できます。

また、第 4 章で説明する**クエリ ストア**や**自動チューニング**、**Adaptive Query Processing**（適応型 JOIN など）、**グラフ データベース**、**非クラスター化列ストア インデックスのオンライン再構築**、**再開可能なインデックス再構築**、**スマート バックアップ**なども利用することができます。

SQL Server on Linux の **Standard** エディションでは、利用できる **CPU** に関して 4 ソケットまたは 24 コアのいずれかの小さい方に制限されますが、**Enterprise** エディションであれば OS がサポートできる最大ソケット数／コア数まで利用することができます。また、**メモリ**に関しては、**Standard** エディションでは **128GB** に制限（列ストア セグメントのキャッシュとインメモリ OLTP のメモリ最適化テーブルは **32GB** に制限）されますが、**Enterprise** エディションであれば OS がサポートできる最大容量のメモリまで利用することができます。

Enterprise エディションでのみ利用できる主な機能としては、自動チューニングや、適応型 JOIN、非クラスター化列ストア インデックスのオンライン再構築、再開可能なインデックス再構築、並列インデックス処理、パーティションのパラレル処理、リソース ガバナー、NUMA 対応のラージ ページ メモリ、I/O リソース管理、分散パーティション ビュー、パラレル整合性チェック、TDE (透過的なデータ暗号化) などがあります。どの機能がどのエディションで利用できるのかについては、以下の URL が参考になります。

エディションと SQL Server 2017 on Linux のサポートされる機能
https://docs.microsoft.com/ja-jp/sql/linux/sql-server-linux-editions-and-components-2017

SQL Server on Linux では、SQL Server で利用できる**標準のセキュリティ機能**をそのまま利用することができます。ユーザー作成やオブジェクト権限の設定だけでなく、監査や行レベル セキュリティ、動的データ マスク、TDE、バックアップ暗号化、ネットワーク接続の暗号化、Always Encrypted、テンポラル テーブル、包含データベースなども利用できます（第 3 章で説明）。

また、その他のシステムとの連携に役立つ**リンク サーバー**や **bcp**、**BULK INSERT** なども利用することができます（Integration Services のパッケージ実行も可能です）。

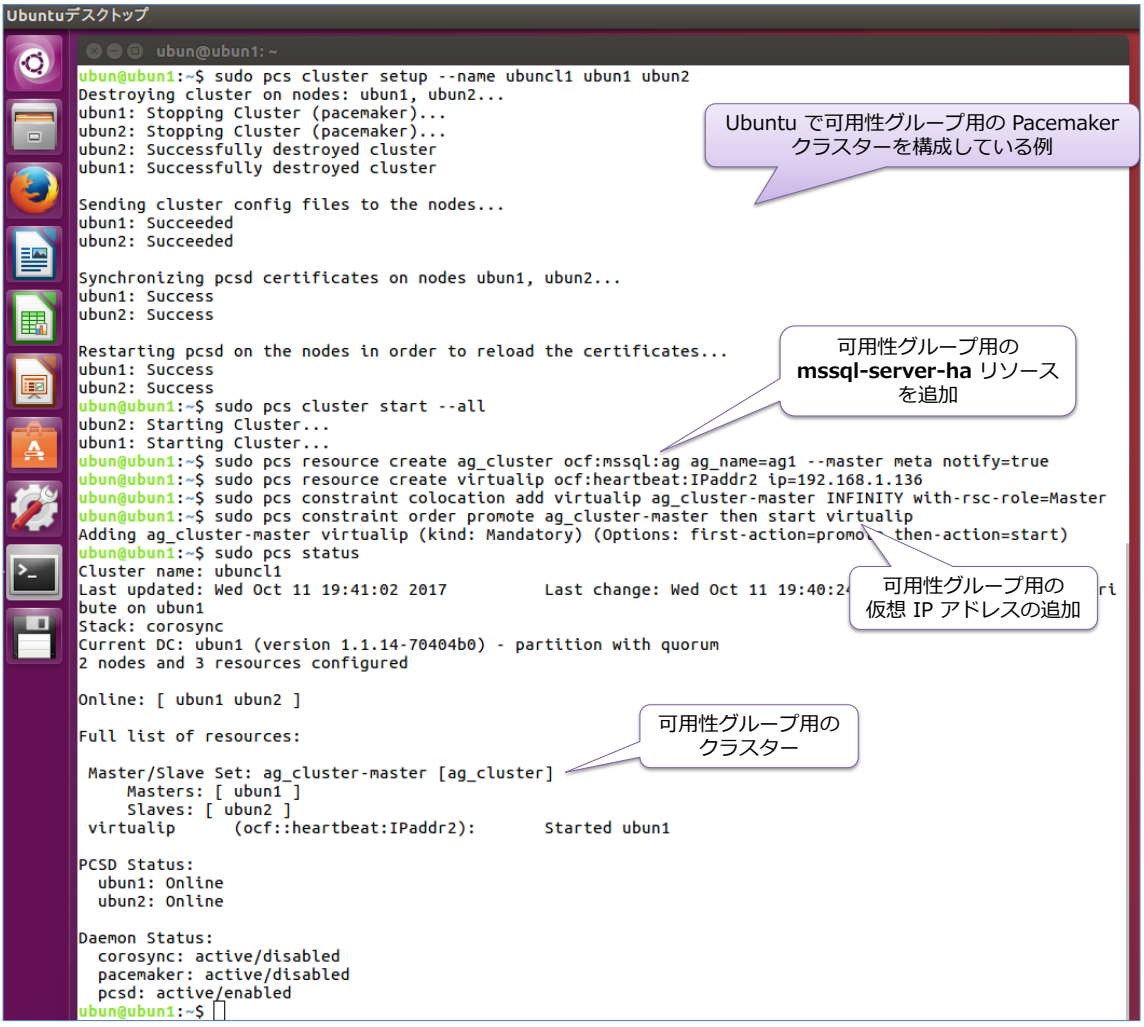
SQL Server でのバックアップのスケジュールなどで定番となっている **SQL Server Agent ジョブ**機能についても、Transact-SQL ステートメントの定期実行がサポートされているので、バックアップやインデックスの再構築といった各種のメンテナンス系の SQL をスケジュール実行することができます。また、**データベース メール**機能もサポートしているので、ジョブの成功や失敗をメールで通達するといったことも行えます。

Machine Learning Services（ML Services）については未サポートになりますが、**PREDICT 関数はサポート**しているので、この関数を利用して ML Services の RevoScalePy や RevoScaleR で機械学習したモデルにアクセスして、予測を実行することができます。Machine Learning Services や PREDICT 関数については、本自習書シリーズの No.3「**Machine Learning Services**」編（現在制作中）で詳しく説明しているので、こちらもぜひご覧いただければと思います。。

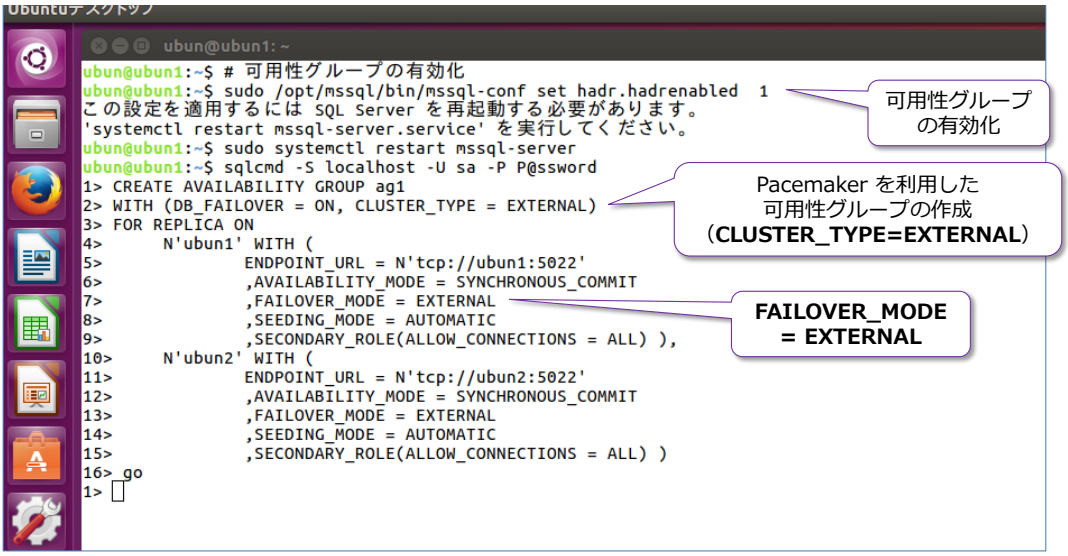
AlwaysOn 可用性グループ（Availability Group）**による冗長構成に関して**

Linux 環境でも **AlwaysOn 可用性グループ**を構成できます。Windows 環境の場合は、WSFC（Windows Server フェールオーバー クラスター）を利用して可用性グループを構成しますが、Linux 環境の場合は、**Pacemaker** を利用します。

Ubuntu で可用性グループ用の Pacemaker クラスターを作成している例

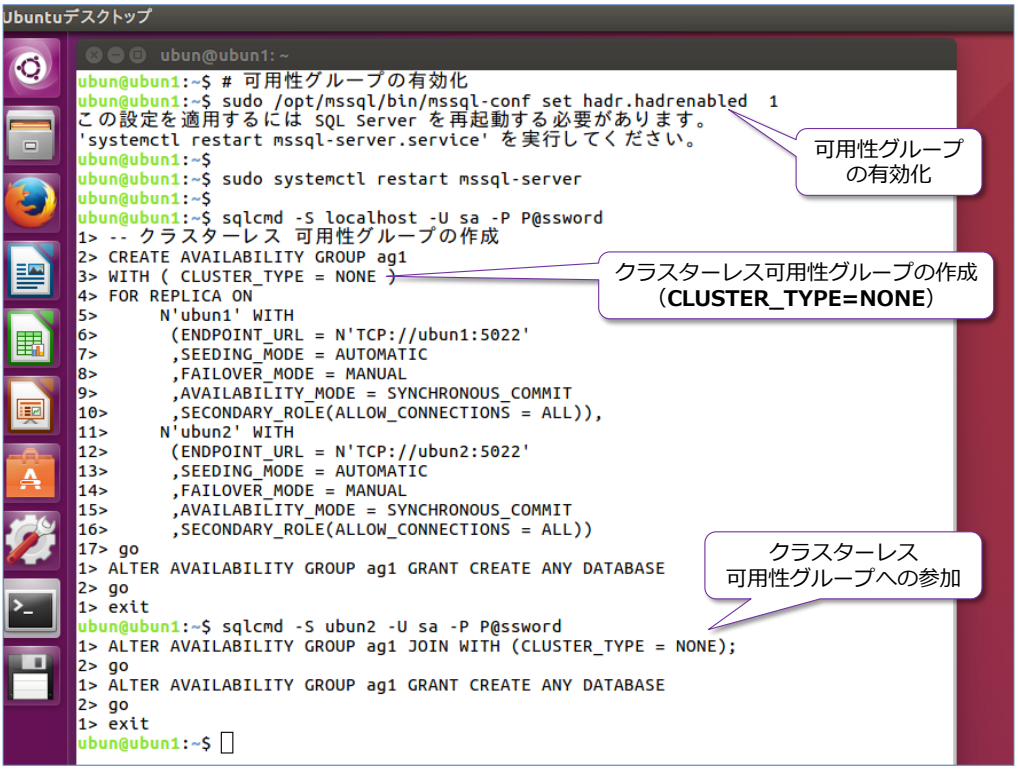


可用性グループを作成するときは、次のように「**CLUSTER_TYPE=EXTERNAL**」と指定することで Pacemaker を利用した可用性グループを利用できるようになります。



可用性グループでは、**クラスター レス**（クラスター不要）の構成もサポートして、この場合は Pacemaker は必要ありません。ただし、これは高可用性が目的のものではなく、**読み取り専用スケール**（読み取り性能を向上させる目的）として可用性グループを利用することになります。

Ubuntu でクラスター レス可用性グループを作成している例



可用性グループは、Windows と Linux にまたがって構成することもできます。また、SQL Server 2016 からの新機能である**分散可用性グループ**（Distributed Availability Group）に、Windows と Linux の可用性グループを含めることもできます。

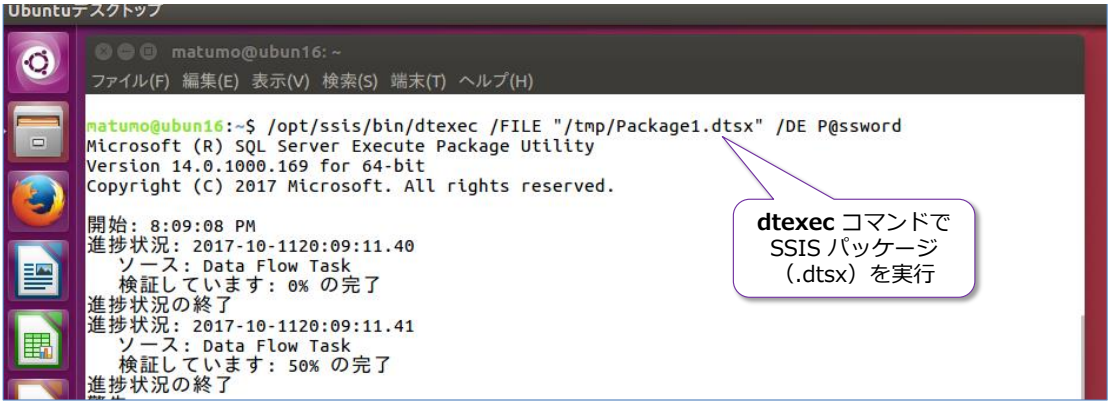
Windows 環境から Linux 環境への移行に関して

Windows 上の SQL Server 環境で取得したバックアップを、Linux 環境にリストアすることも、何の問題もなく行えるので、**移行**（マイグレーション）も簡単です（リストア時の考慮事項は、Windows 環境でのリストアの場合と全く同様です）。

既存の Windows 環境に対して、Linux を含めた可用性グループを構成すれば、まったく同じデータベース（ミラー化した複製データベース）を Linux 上に作成することができるので、段間的な移行用途として可用性グループを利用することもできます。

Integration Services（SSIS）に関して

Integration Services（SSIS）は、SQL Server 2017 on Linux でも利用することができます。ただし、SSIS パッケージの作成は、従来どおり Windows 上の **SSDT**（SQL Server Data Tools）を利用して行って、作成したパッケージ ファイル（**.dtsx**）を Linux 上にコピーして、**dtexec** コマンドで実行するという形になります。



ただし、Integration Services に関して、以下の機能については利用することができません。

- SSIS カタログ データベース
- SQL Server Agent ジョブでのパッケージ実行ジョブの登録
- Windows 認証、サードパーティ コンポーネント、CDC（変更データキャプチャ）
- SSIS スケールアウト、Azure Feature Pack for SSIS、Hadoop／HDFS サポート、Microsoft Connector for SAP BW

SQL Server 2017 on Linux では利用できない機能

SQL Server 2017 on Linux では利用できない機能は、次のとおりです。

	利用できない機能
データベース エンジン	トランザクション／マージ レプリケーション ストレッチ データベース、Polybase、サードパーティ接続の分散クエリ、拡張システム ストアド プロシージャ（xp_cmdshell など）、Filetable、バッファ プール拡張、 CLR アセンブリでの EXTERNAL_ACCESS／UNSAFE 権限
SQL Server Agent	ジョブのサブシステム： CmdExec、PowerShell、Queue Reader、SSIS、SSAS、SSRS 警告（Alert）、ログ リーダー エージェント、CDC、 Managed Backup
可用性	データベース ミラーリング（DBM）
セキュリティ	拡張キー管理 リンク サーバー／可用性グループでの Active Directory 認証
その他のサービス	SQL Server Browser、 Machine Learning Services (ただし、PREDICT 関数は利用可能。3章で説明)、 StreamInsight、 Analysis Services（SSAS）、 Reporting Services（SSRS）、 Data Quality Services（DQS）、 Master Data Services（MDS）

こうした on Linux での未サポート機能については、以下のリリースノートが参考になります。

Release notes for SQL Server 2017 on Linux
Unsupported features and services
https://docs.microsoft.com/en-us/sql/linux/sql-server-linux-release-notes#Unsupported

2.4 Visual Studio Code のインストール、mssql 拡張機能

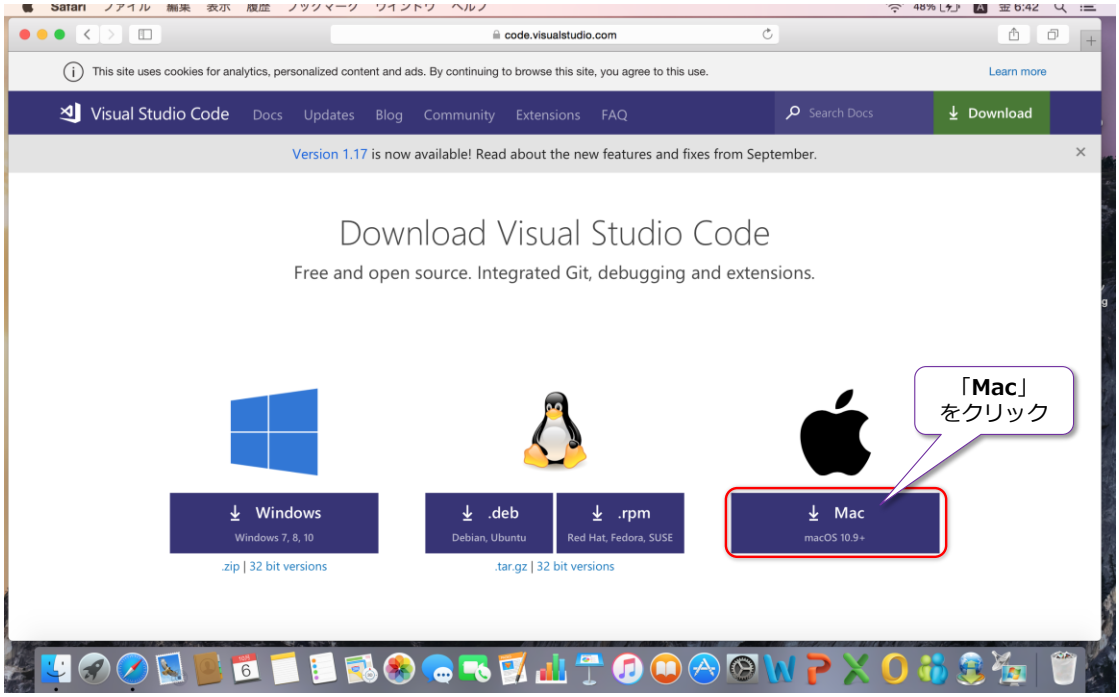
ここでは、Mac OS や Linux に **Visual Studio Code** をインストールする方法や、Visual Studio Code に **mssql 拡張機能**を追加する方法、mssql 拡張機能を利用する方法などについて説明します。

Mac OS への Visual Studio Code のインストール

まずは、**Mac OS X** に **Visual Studio Code** をインストールする手順を説明します。

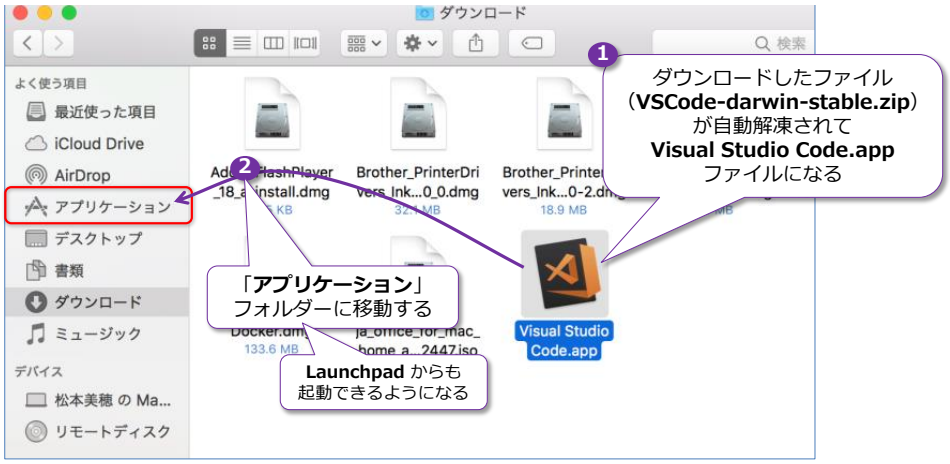
1. **Mac OS X** に **Visual Studio Code** をインストールするには、次の URL からファイルをダウンロードします。

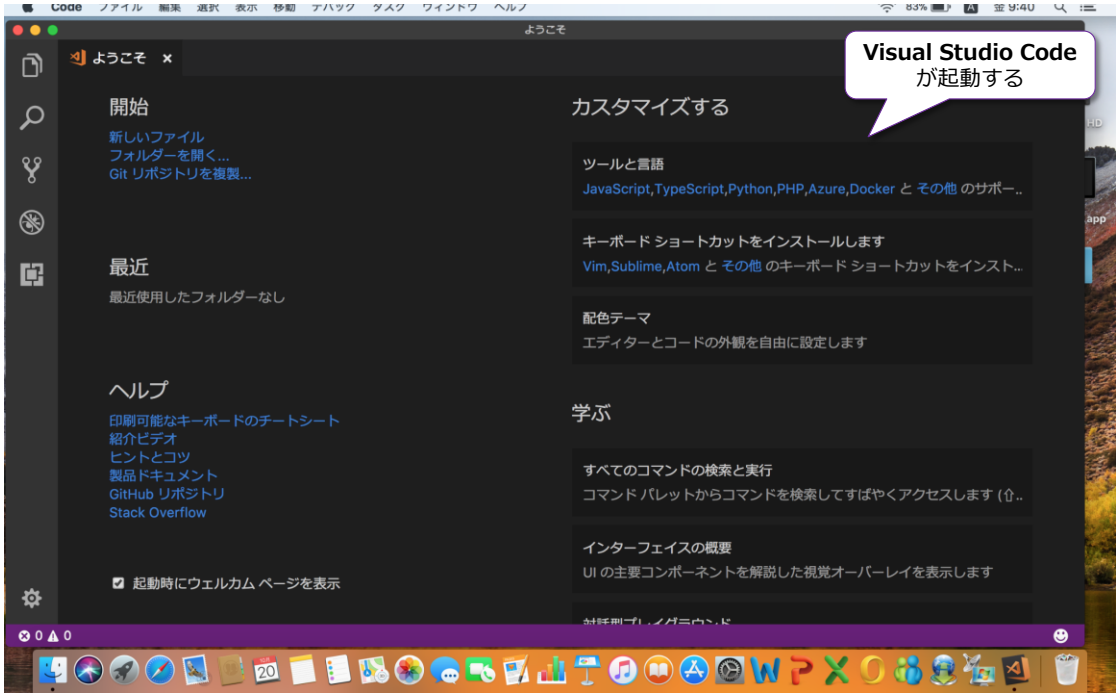
   Download Visual Studio Code
   https://code.visualstudio.com/download



2. ファイル（**VSCode-darwin-stable.zip**）のダウンロードが完了すると、.zip が自動解凍されて、**Visual Studio Code.app** ファイルが出来上がります（自動解凍されない場合は、.zip を解凍しておいてください）。

   この **Visual Studio Code.app** ファイルを「**アプリケーション**」フォルダーなどに移動しておけば、Visual Studio Code のインストールが完了です。後は、この .app ファイルをダブルクリックすれば、Visual Studio Code を起動することができます（アプリケーション フォルダーに移動することで、Launchpad から起動することもできます）。





Mac OS X への mssql 拡張機能のインストール（Mac OS X 10.12 以上が必要）

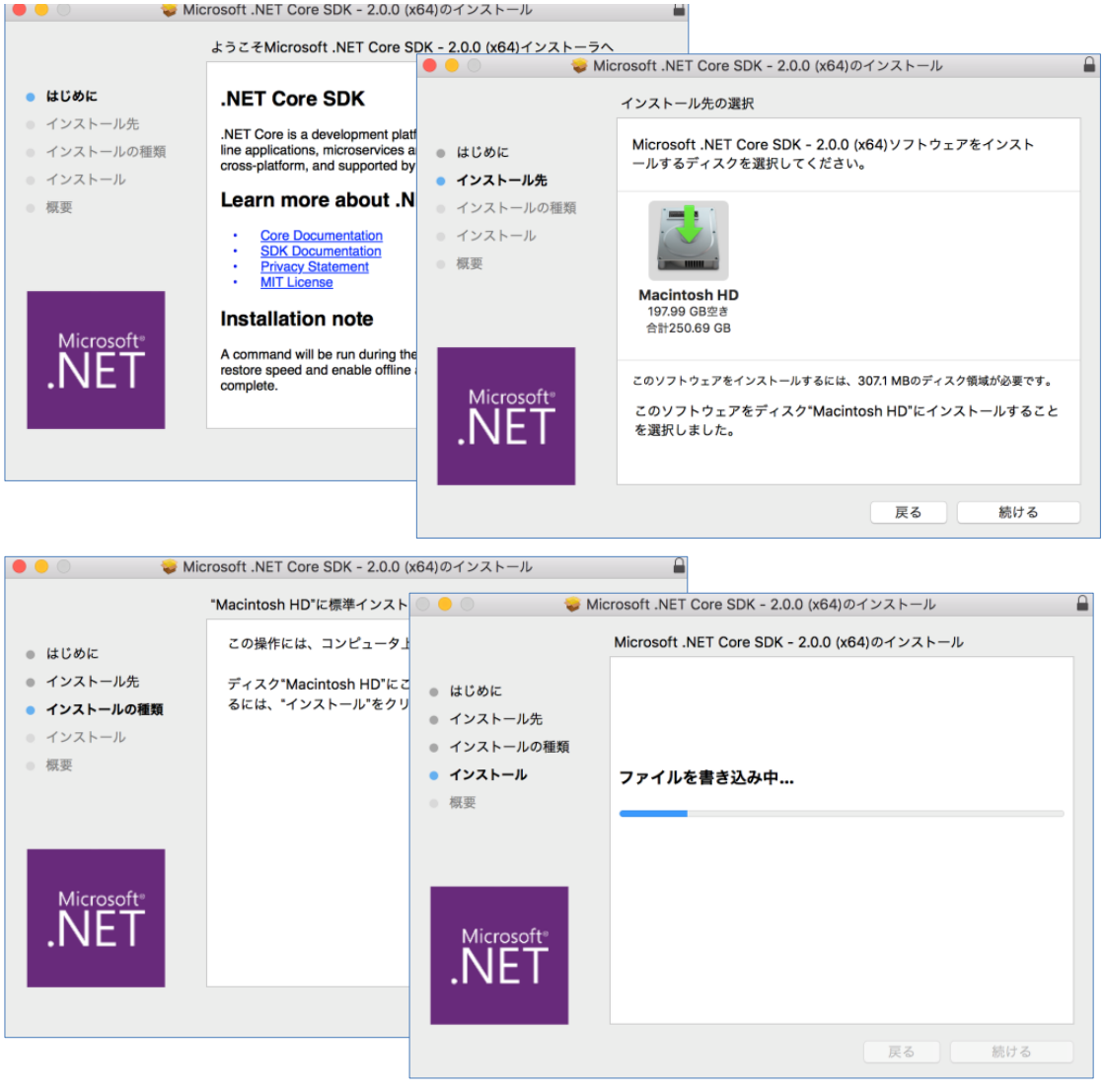
Mac OS X で Visual Studio Code の「**mssql 拡張機能**」を利用するためには、次のソフトウェア要件があります。

- **.NET Core SDK** のインストールが必要
  （.NET Core SDK のインストールには、**Mac OS X** の **10.12** 以上が必須要件）
- **OpenSSL** のインストールが必要

1. **.NET Core SDK** は、次の URL からインストーラーをダウンロードすることができます。

   https://www.microsoft.com/net/core#macos

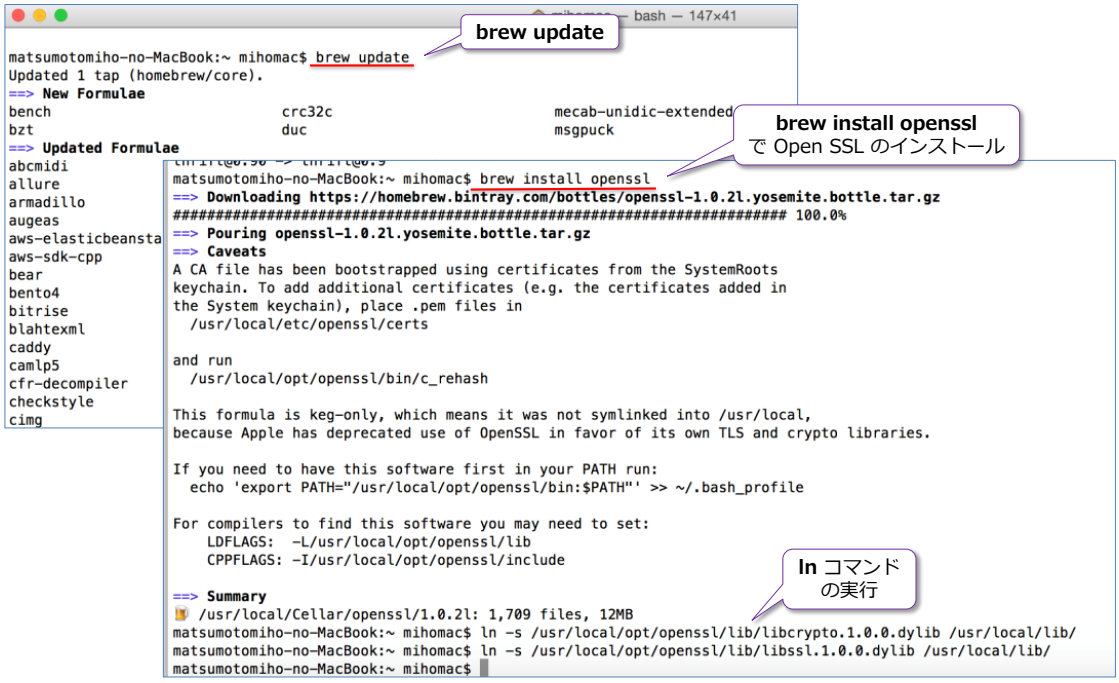
インストーラー（**dotnet-sdk-2.0.0-osx-gs-x64.pkg**）のダウンロードが完了したら、これをダブルクリックしてインストールを開始します。



2. .NET Core SDK のインストールが完了したら、次は、**Open SSL** をインストールします。こ

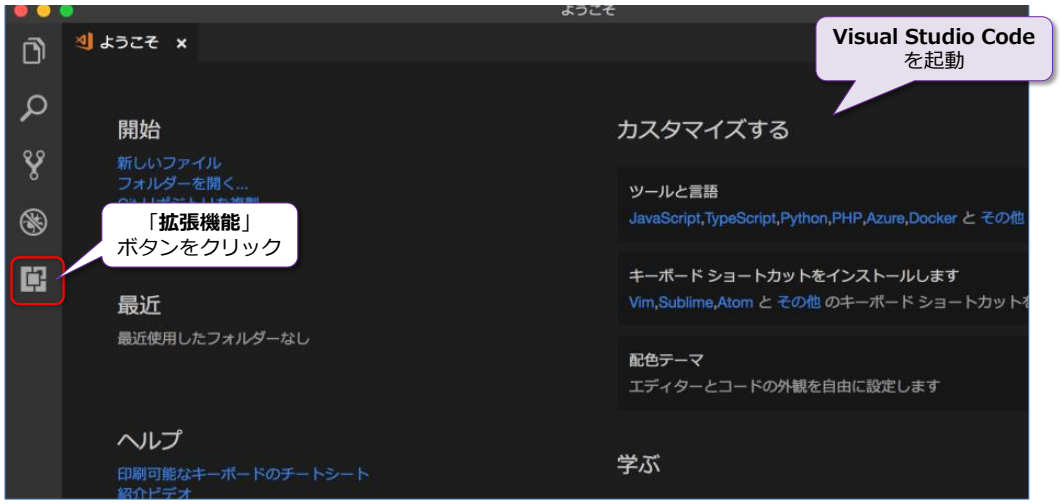
れは、Mac OS 用のパッケージ マネージャーである「**Homebrew**」を利用して、次のように行います（**ターミナル**を起動して、**brew** コマンドを実行します）。

```
brew update
brew install openssl
ln -s /usr/local/opt/openssl/lib/libcrypto.1.0.0.dylib /usr/local/lib/
ln -s /usr/local/opt/openssl/lib/libssl.1.0.0.dylib /usr/local/lib/
```



以上で、**mssql 拡張機能**のインストールをための準備が完了です。

3. **mssql 拡張機能**をインストールするには、まず **Visual Studio Code** を起動します。Visual Studio Code が起動したら、画面左側の「**拡張機能**」ボタンをクリックします。



4. 次のように「**拡張機能**」サイドバーが表示されたら、「**検索**」ボックスで「**mssql**」と入力します。

「**mssql**」で検索すると、**mssql 拡張機能**（執筆時点でのバージョンは **mssql 1.2.0**）が表示されるので、これをクリックします。これで、右ペインに mssql 拡張機能の詳細が表示されます。ここに「**インストール**」ボタンがあるので、これをクリックすれば、mssql 拡張機能をインストールすることができます。

5. インストールが完了すると、次のように［**再読み込み**］ボタンが表示されるので、これをクリックします。

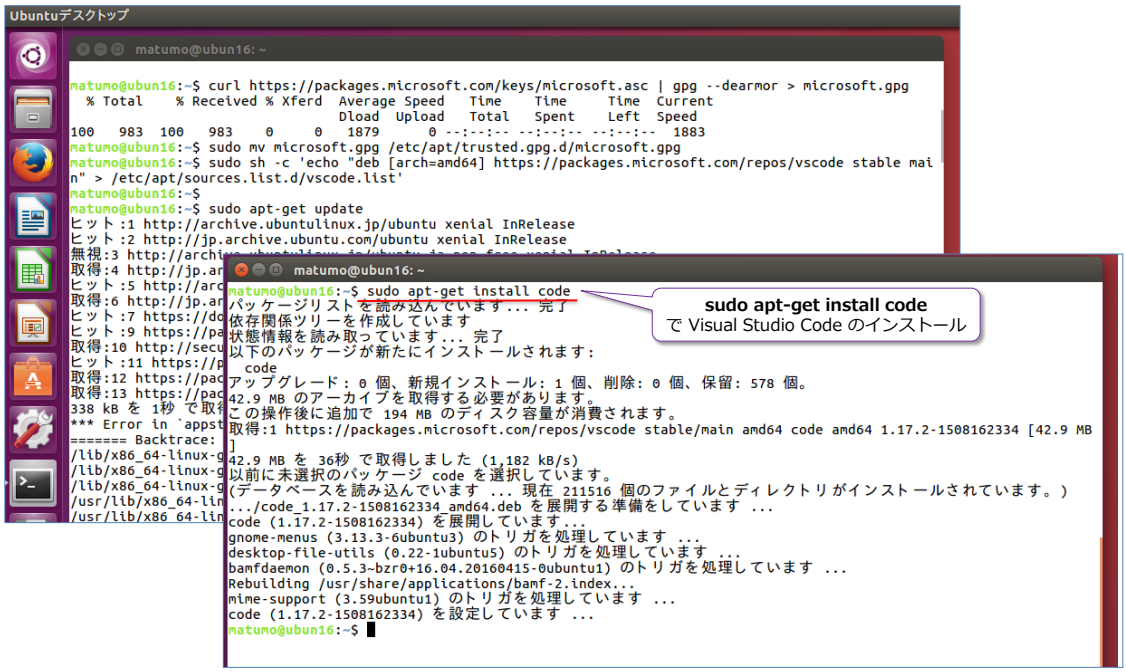
以上で、**mssql 拡張機能**のインストールが完了です。

Linux への Visual Studio Code／mssql 拡張機能のインストール

次に、Linux 環境（**Ubuntu** や **Red Hat Enterprise Linux**）に、**Visual Studio Code** および **mssql 拡張機能**をインストールする手順を説明します。

1. **Ubuntu** や **Debian** を利用している場合には、次のように実行することで **Visual Studio Code** をインストールすることができます（4 行目には改行が入っていますが、実際にコマンドを入力するときは、改行なしで記述するようにしてください）。

```
curl https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor > microsoft.gpg
sudo mv microsoft.gpg /etc/apt/trusted.gpg.d/microsoft.gpg

# 以下は改行なしで １行で実行してください
sudo sh -c 'echo "deb [arch=amd64] https://packages.microsoft.com/repos
/vscode stable main" > /etc/apt/sources.list.d/vscode.list'

sudo apt-get update
sudo apt-get install code
```



コマンドの入力が面倒な場合は、以下の URL（Visual Studio Code の公式ページ）から、コマンドをコピーするのがお勧めです。

Running VS Code on Linux
https://code.visualstudio.com/docs/setup/linux

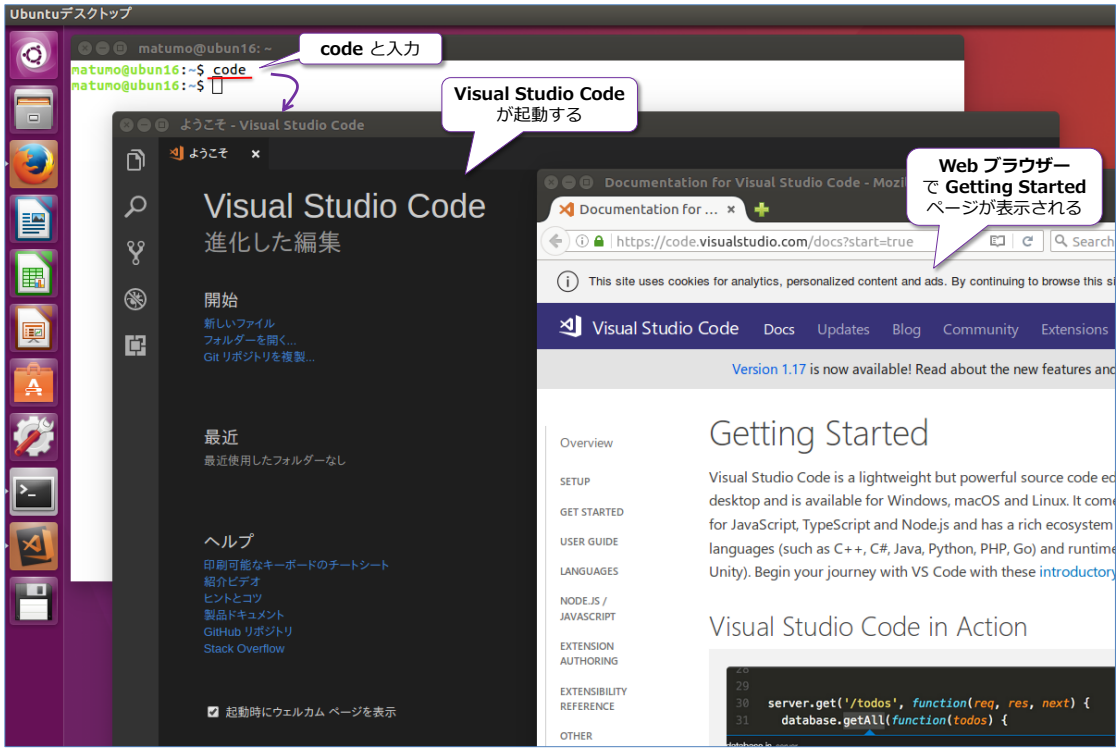
RHEL（Red Hat Enterprise Linux）や **Fedora**、**CentOS** を利用している場合には、次のようにコマンドを入力することで **Visual Studio Code** をインストールすることができます。

```
sudo rpm --import https://packages.microsoft.com/keys/microsoft.asc
```

```
# 以下は改行なしで １行で実行してください
sudo sh -c 'echo -e "[code]¥nname=Visual Studio Code¥nbaseurl=https://packages.microsoft.com/yumrepos/vscode¥nenabled=1¥ngpgcheck=1¥ngpgkey=https://packages.microsoft.com/keys/microsoft.asc" > /etc/yum.repos.d/vscode.repo'

yum check-update
sudo yum install code
```

2. Visual Studio Code のインストールが完了すると、**/usr/bin/** ディレクトリに実行ファイル（**code** という名前）がインストールされているので、次のように「**code**」と入力、**PATH** が切られていない場合は「**/usr/bin/code**」と入力することで、Visual Studio Code を起動することができます。

```
# Visual Studio Code の起動
code
```



以上で Visual Studio Code のインストールが完了です。

Linux への mssql 拡張機能のインストール

1. **Linux** 環境で **mssql 拡張機能**をインストールするには、まず **Visual Studio Code** の画面左側の「**拡張機能**」ボタンをクリックします。

「**拡張機能**」サイドバーが表示されたら、「**検索**」ボックスで「**mssql**」と入力します。

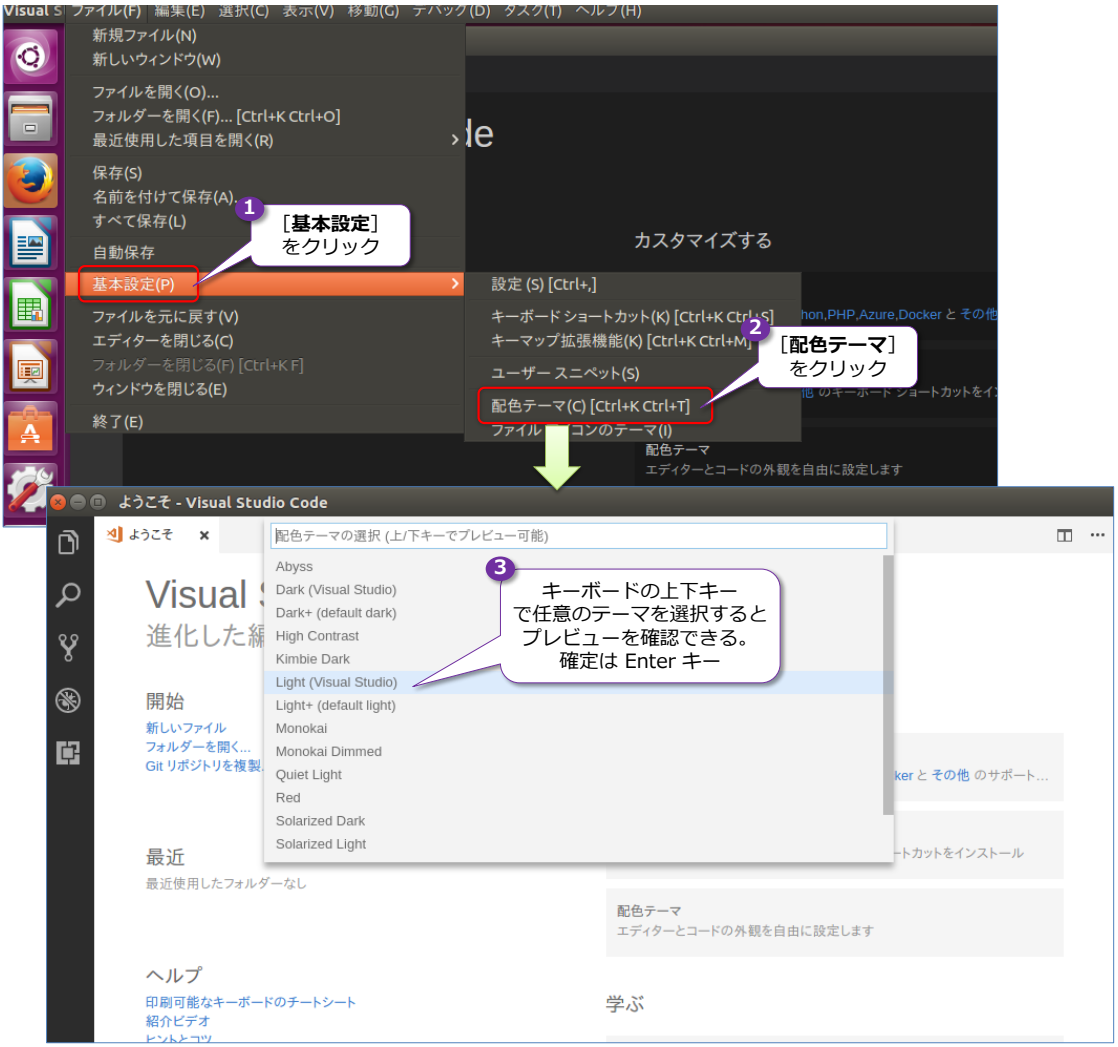
mssql 拡張機能が表示されたら、これをクリックして、右ペインに mssql 拡張機能の詳細を表示します。ここの「**インストール**」ボタンをクリックすれば、mssql 拡張機能をインストールすることができます。

2. インストールが完了すると、次のように［**再読み込み**］ボタンが表示されるので、これをクリックします。

以上で、**mssql 拡張機能**のインストールが完了です。

Visual Studio Code の配色テーマの変更

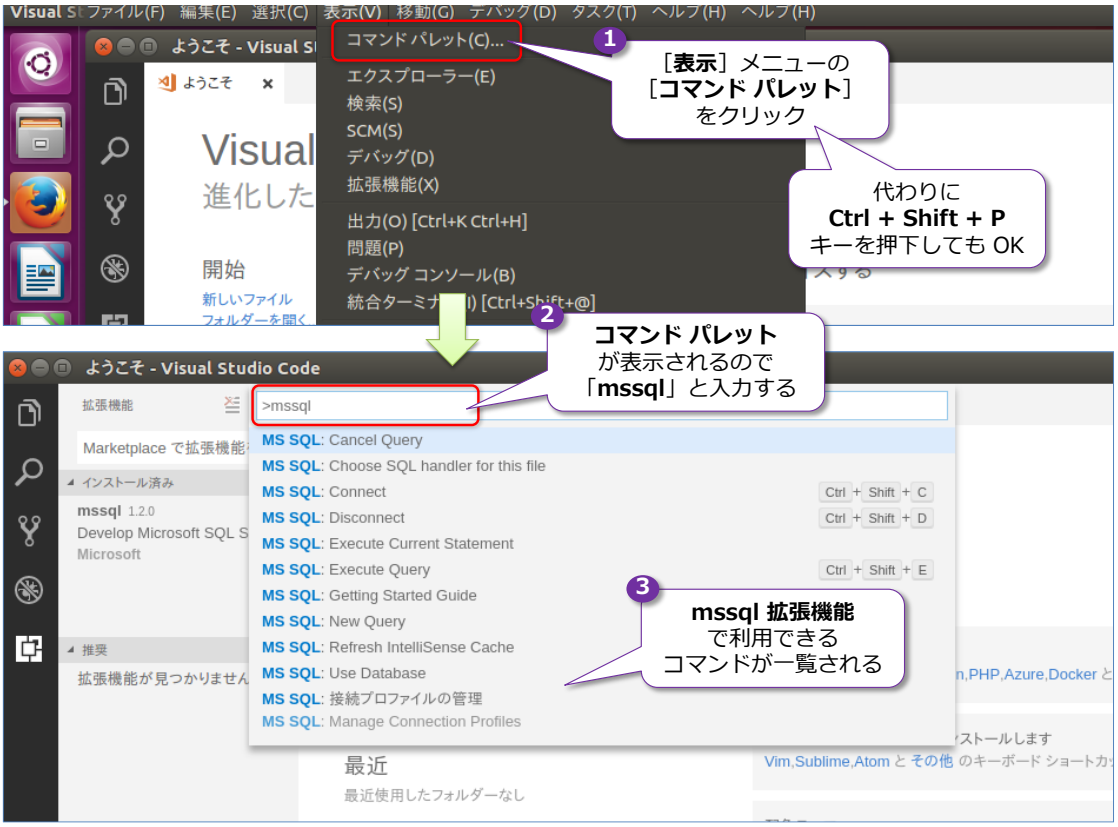
Visual Studio Code は、［**ファイル**］メニュー（Mac OS の場合は［**Code**］メニュー）の［**基本設定**］から［**配色テーマ**］をクリックすることで、全体的な配色を変更することができます。



以降、本文内では、配色テーマを「**Light（Visual Studio）**」に設定した状態の画面ショットで説明します。

2.5 mssql 拡張機能で SQL Server に接続（Visual Studio Code）

次に、**Visual Studio Code** の **mssql 拡張機能**を利用して、SQL Server に接続する手順を説明します。mssql 拡張機能を利用するには、[**表示**] メニューの [**コマンド パレット**] をクリックして、コマンド パレットを開き、「**mssql**」と入力します。



コマンド パレットは、**Ctrl + Shift + P** キーを押下しても開くことができます（パレットの **P** と覚えると、ショートカット キーを押しやすくなります）。

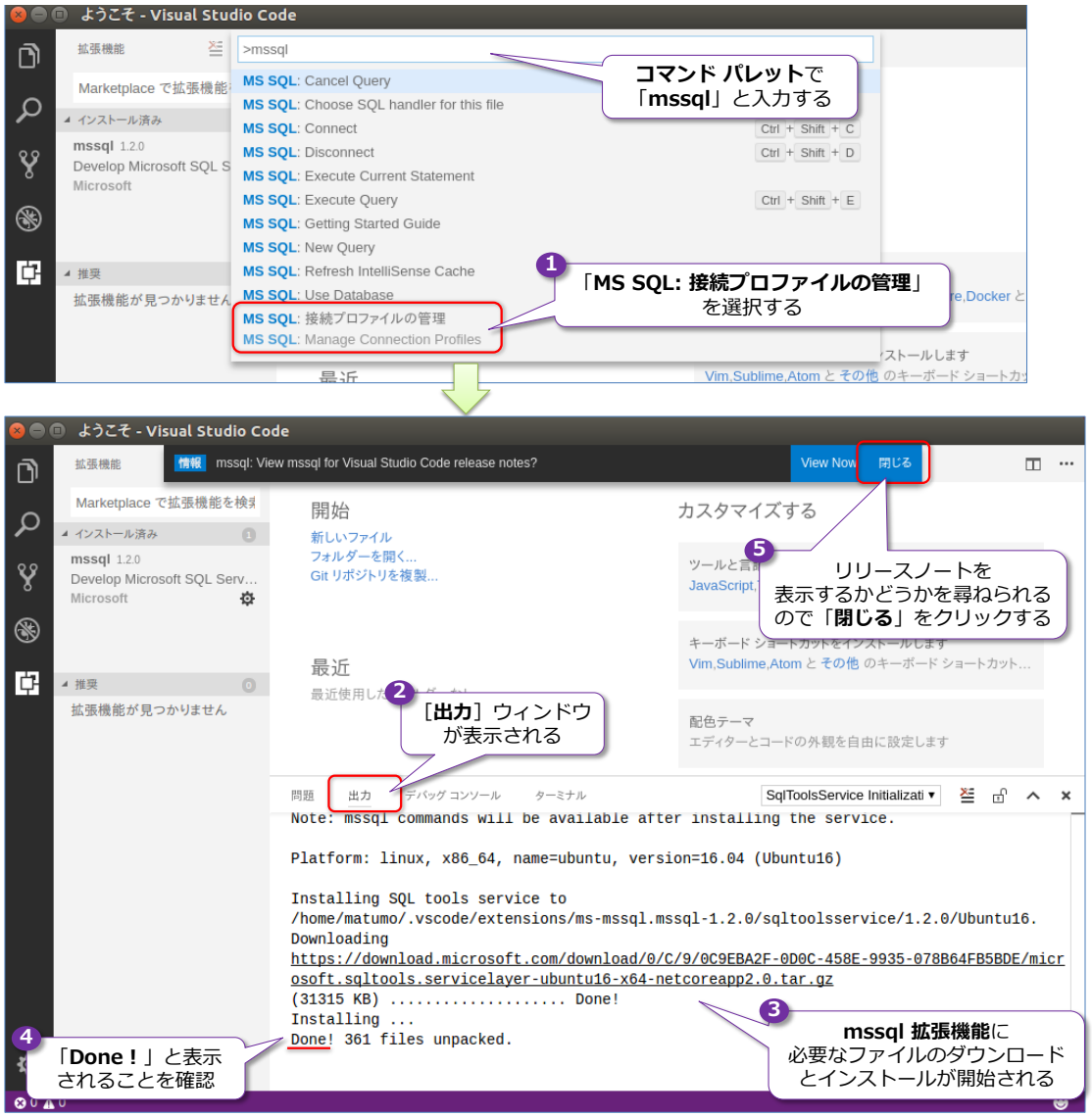
コマンド パレットで「**mssql**」と入力すると、「**MS SQL: Connect**」や「**MS SQL: Execute Query**」、「**MS SQL: New Query**」、「**MS SQL: 接続プロファイルの管理**」など、mssql 拡張機能で利用できるコマンドが一覧表示されます。

このうち、SQL Server に対して対話的に Transact-SQL ステートメントを実行するには、「**MS SQL: 接続プロファイルの管理**」でプロファイルを作成して、「**MS SQL: New Query**」で新しいクエリを記述して実行する、といった流れになります。

Let's Try

それでは、これも試してみましょう。

1. まずは、コマンド パレットで「**mssql**」と入力して、「**MS SQL: 接続プロファイルの管理**」を選択します。

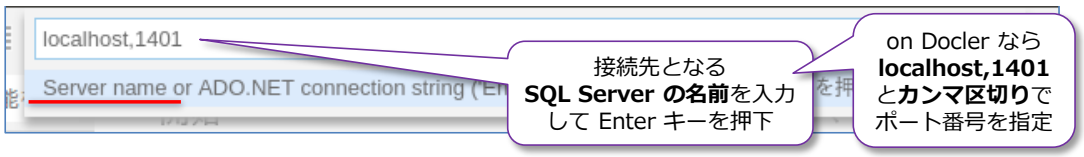


mssql 拡張機能では、いずれかのコマンドを選択したときに、動作に必要なファイルのダウンロードとインストールが追加で行われて、この経過が［**出力**］ウィンドウに表示されます。ダウンロード中は「**Downloading**」、インストール中は「**Installing...**」と表示されて、最後に「**Done !**」と表示されればインストールが完了です。

インストールが完了すると、画面上部に、リリースノートを表示するかどうかを尋ねられるので、「**閉じる**」をクリックします（表示したい場合は「**View Now**」をクリックします）。

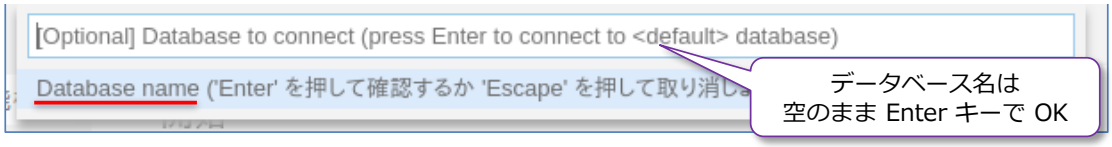
2. 続いて、もう一度、［**表示**］メニューから［**コマンド パレット**］をクリック、または **Ctrl + Shift + P** キーを押下して、コマンド パレットを開いて、「**最近使用したもの**」から「**MS SQL: 接続プロファイルの管理**」を選択します。コマンド パレットでは、直近に利用したコマンドを「**最近使用したもの**」から選択できます。

3. **接続プロファイルの管理**では、接続先となる SQL Server の名前やデータベース名（オプション）、認証方法、認証ユーザー名などを指定しますが、まずは、次のように**接続先となる SQL Server の名前**が聞かれます。

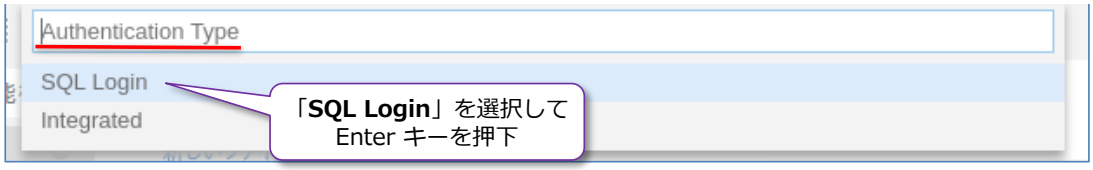


ここでは、ローカル環境にインストールした SQL Server の場合は「**localhost**」と記述して、**Docker** 上の SQL Server の場合は「**localhost,1401**」のようにサーバー名の後にカンマを付けて、ポート番号（Docker で **run** をしたときに **-p 1401:1433** としている場合は **1401**）を指定するようにします。

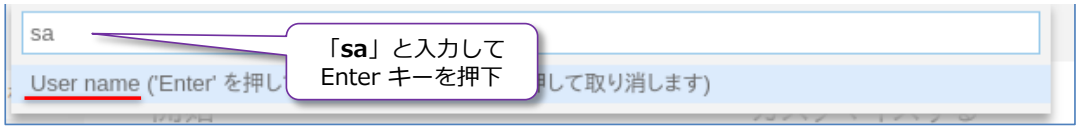
4. 次に、接続先となるデータベース名を尋ねられますが、これはオプションなので、何も入力せずに、Enter キーを押下すれば大丈夫です。



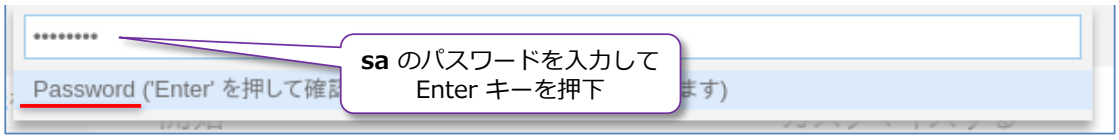
5. 次に、認証方法が尋ねられますが、「**SQL Login**」を選択して、SQL Server 認証（SQL Server 上に登録されたログイン アカウントを利用してログイン認証する方法）を利用します。



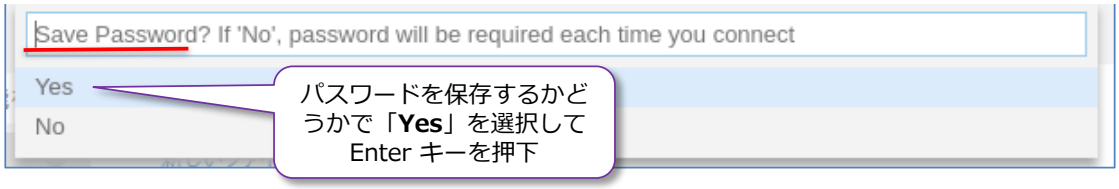
6. 次に、ログイン ユーザー名が尋ねられるので、SQL Server の管理者アカウントである「**sa**」と入力して、Enter キーを押下します。



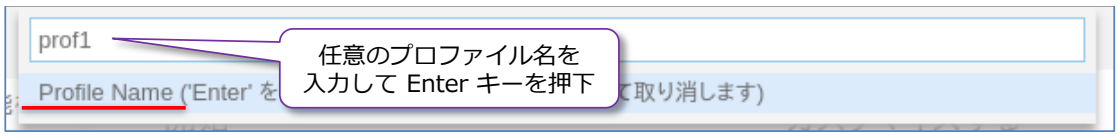
7. 次に、**sa** のパスワードを入力して、Enter キーを押下します。



8. 次に、パスワードを保存するかどうかを尋ねられるので、「**Yes**」を選択して、Enter キーを押下します。

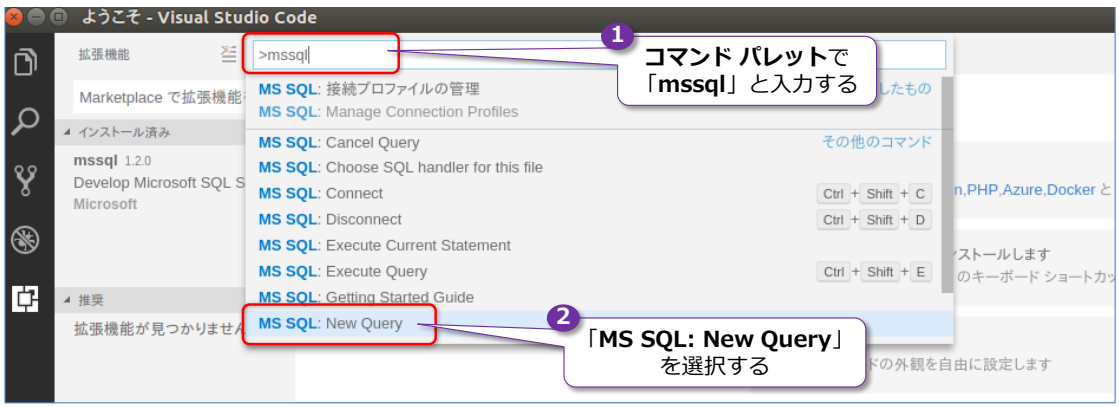


9. 次に、**プロファイル名**の入力を求められるので、任意の名前（画面は **prof1**）を入力して、Enter キーを押下します。



以上で接続プロファイルの作成が完了です。プロファイルの作成が完了すると、画面上部に「**Profile created successfully**」と表示されるので、[**閉じる**] をクリックします。

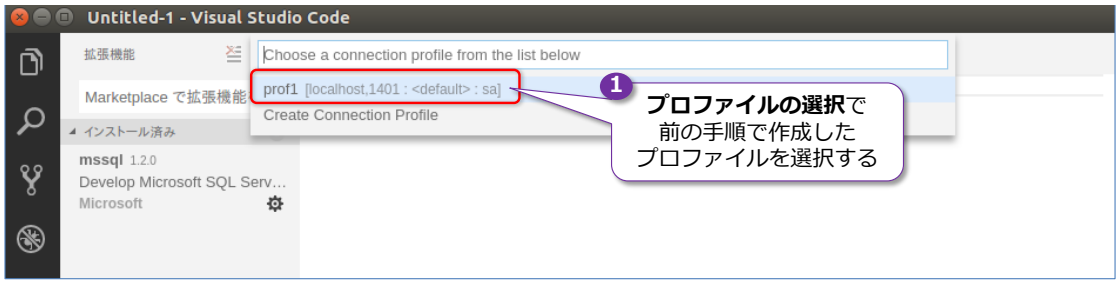
10. 次に、SQL ステートメントを実行するために、[**表示**] メニューから [**コマンド パレット**] をクリックまたは **Ctrl + Shift + P** キーを押下して、コマンド パレットを開いて、「**mssql**」と入力します。



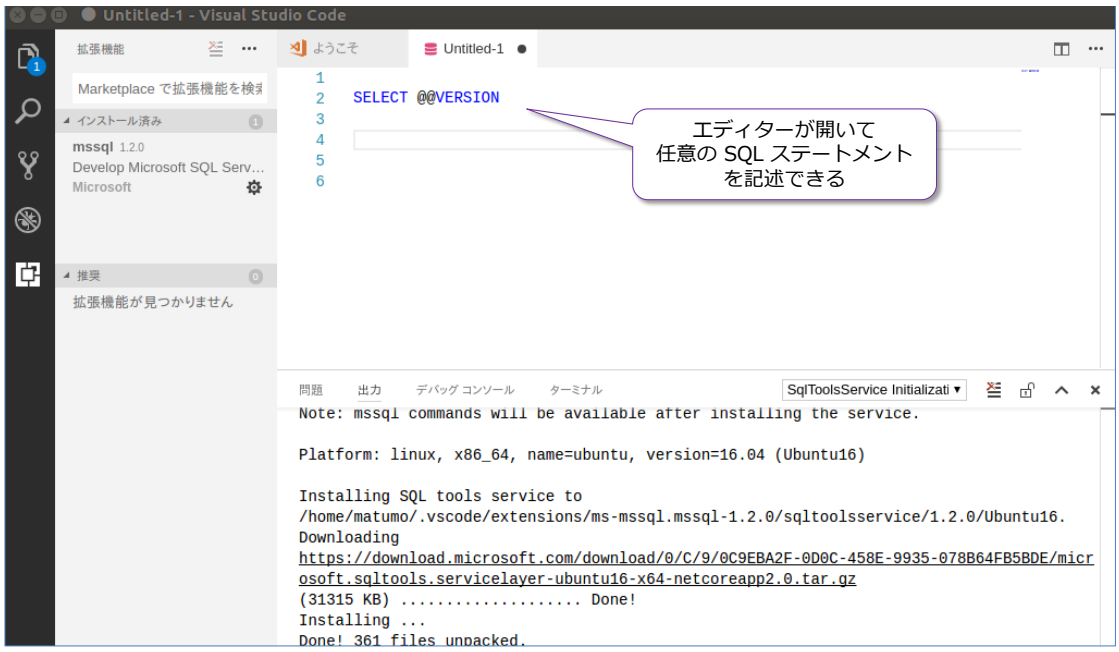
mssql 拡張機能の一覧が表示されたら、「**MS SQL: New Query**」（クエリを実行するための

コマンド）を選択します。

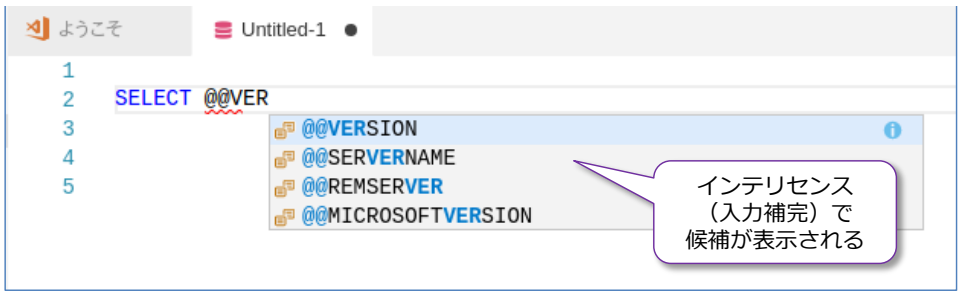
11. 次に、接続プロファイルの選択が求められるので、前の手順で作成したプロファイル（画面は **prof1**）を選択します。



12. これで任意の SQL ステートメント（クエリ）を記述できるエディターが表示されます。



このエディターでは、Visual Studio でお馴染みの**インテリセンス**（入力補完）機能が働くので、SQL ステートメントが非常に入力しやすくなっています。

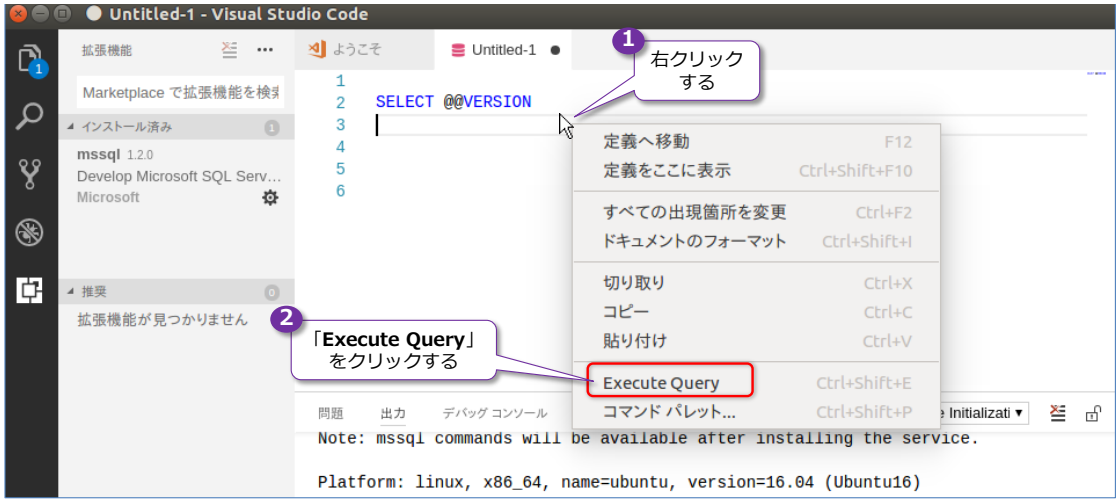


13. 次に、SQL Server のバージョンを取得するための SELECT ステートメントを「**SELECT @@VERSION**」と入力してみましょう。

```
SELECT @@VERSION
```

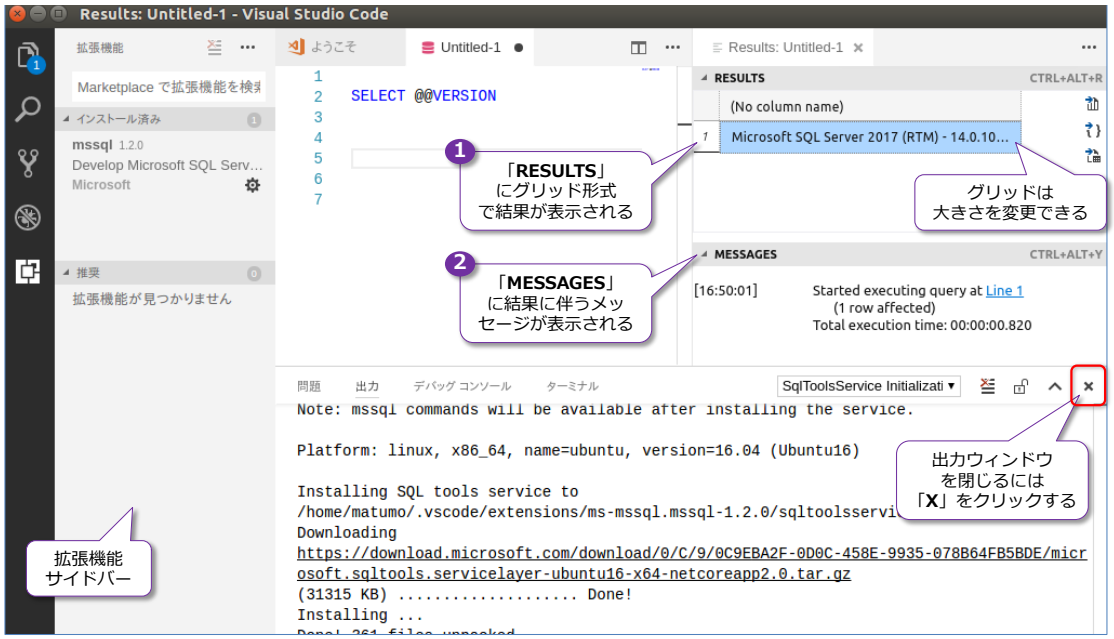
前掲の **sqlcmd** ツールでは、ステートメントを実行するために「**go**」を記述していましたが、mssql 拡張機能では、「**go**」は必須ではありません（後述の**バッチ**という区切りのためには、**go** を利用しますが、必須ではありません）。

14. mssql 拡張機能のエディターに記述した SQL ステートメント（クエリ）を実行するには、次のように右クリックして、［**Execute Query**］（クエリの実行）をクリックします。

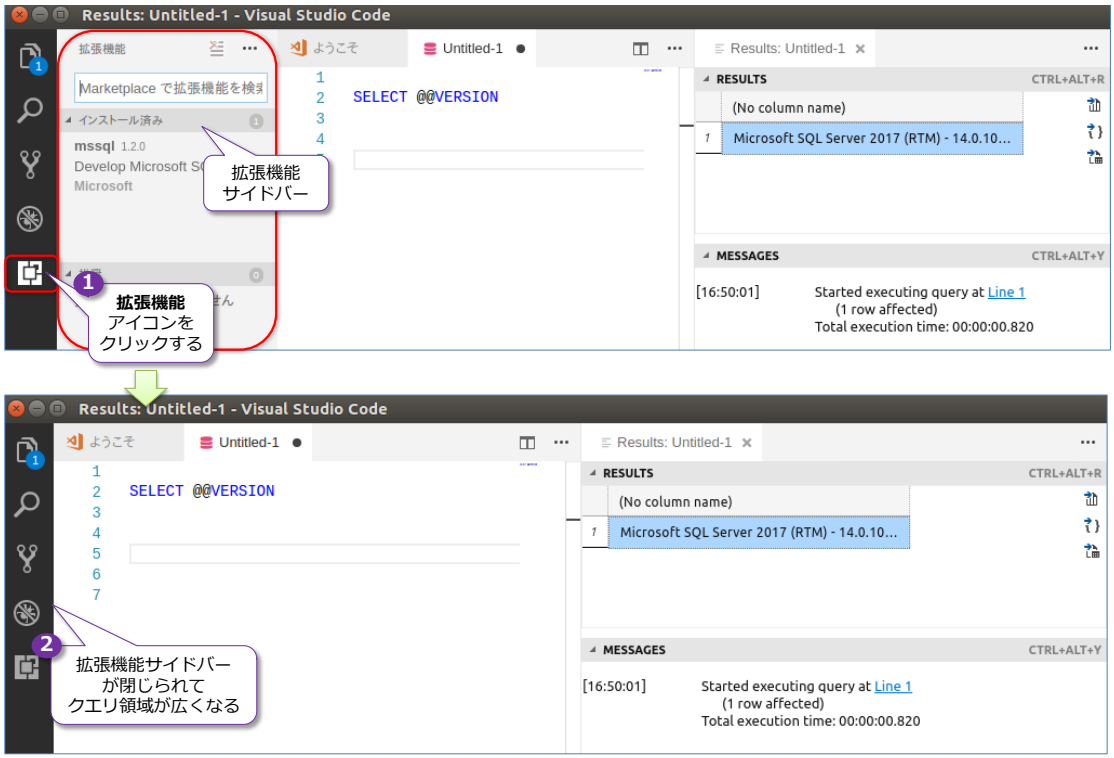


［**Execute Query**］をクリックする代わりに、**Ctrl + Shift + E** キーを押下しても、クエリを実行することができます（Execute の **E** と覚えると押しやすくなります）。

15. クエリを実行すると、画面右側に「**RESULTS**」と「**MESSAGES**」という 2 つのウィンドウが表示されて、結果を確認することができます。



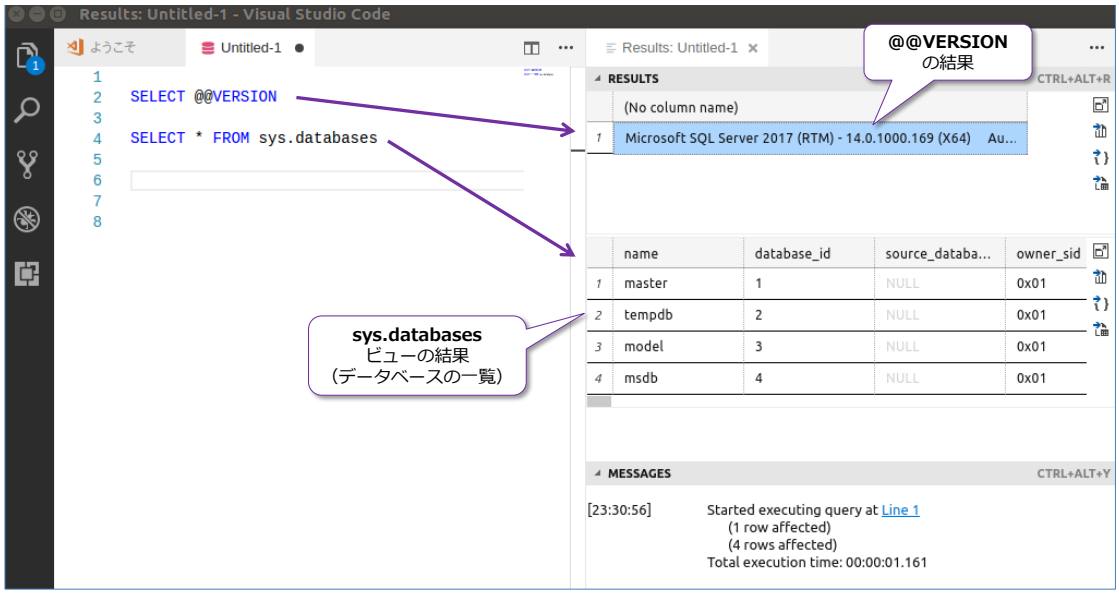
上の画面では、**拡張機能サイドバー**や、**出力ウィンドウ**が表示されているので、画面が狭くなっていますが、出力ウィンドウは、右上の［**X**］ボタンをクリックして閉じることができます。拡張機能サイドバーは、次のように画面左側のサイドバーを表示したときにクリックした**拡張機能アイコン**をクリックすれば、閉じることができます。



16. 次に、SQL Server のデータベースの一覧を取得できる「**sys.databases**」システム ビューを参照してみましょう。

```
SELECT * FROM sys.databases
```

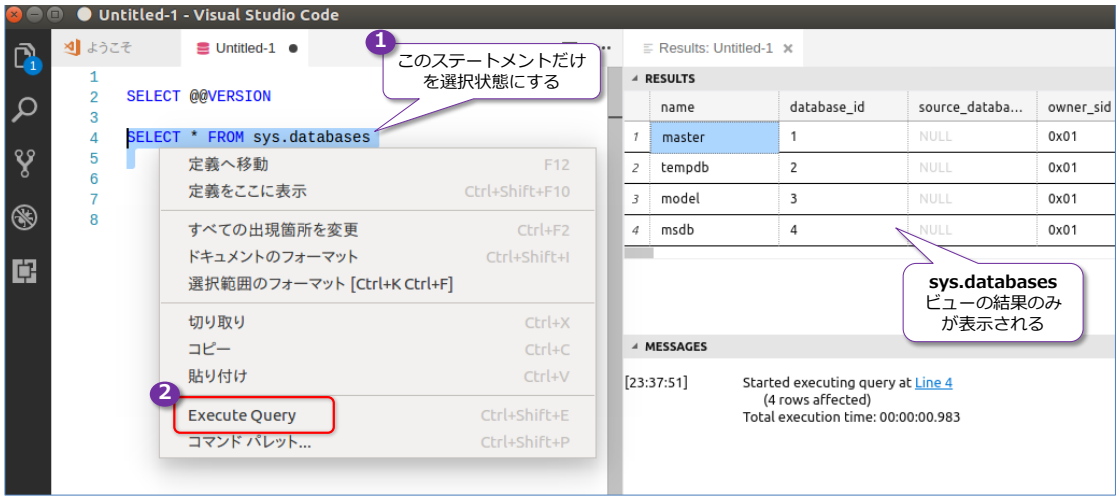
SELECT ステートメントを記述したら、右クリックして［**Execute Query**］または **Ctrl + Shift + E** キーを押下して、クエリを実行してみましょう。



［**Execute Query**］では、既定では、エディターに記述した全ての SQL ステートメントを実行するので、前の手順で記述した「**SELECT @@VERSION**」も、今回の「**sys.databases**」も両方とも実行されています。「**sys.databases**」の結果からは、SQL Server が内部的に利用しているシステム データベースである「**master**」や「**tempdb**」、「**model**」、「**msdb**」の

4 つのデータベースを確認することができます。

17. 次に、「**SELECT * FROM sys.databases**」ステートメントのみをドラッグして選択状態にしてから、［**Execute Query**］を実行してみてください。



このように、ステートメントを選択状態にすることで、該当ステートメントのみを実行できるようになります。

以上のように、mssql 拡張機能を利用すると、SQL ステートメントを非常に入力／実行しやすくなるので、ぜひ活用してみてください。

次の章以降では、この mssql 拡張機能を利用した手順で説明していきます。

STEP 3. SQL Server に接続するアプリケーション開発の概要

この STEP では、Linux 上の SQL Server に接続／アクセスするアプリケーション（**Linux** および **Mac OS** クライアントで動作可能なアプリケーション）を開発する際の概要を説明します。

この STEP では、次のことを学習します。

- ✓ **Java** を利用したアプリケーション開発の概要
- ✓ **PHP** を利用したアプリケーション開発の概要
- ✓ **Python** を利用したアプリケーション開発の概要あ
- ✓ **node.js** を利用したアプリケーション開発の概要
- ✓ **.NET Core** を利用したアプリケーション開発の概要

3.1 アプリケーション開発の概要（Linux、Mac OS クライアント）

従来ながらの **Windows** をクライアントとしたアプリケーションを開発する場合には、**Linux** 上の SQL Server 2017 にアクセスする場合でも、Windows クライアントに **.NET Framework** をインストールしておくだけで、**C#** や **VB**（ADO.NET や Entity Framework、ASP.NET など）を利用して、Windows 上の SQL Server にアクセスするのと全く同じように、SQL Server on Linux にアクセスするアプリケーションを開発することができます。

Linux や **Mac OS** をクライアントにする場合には、「**.NET Core**」を Linux や Mac OS にインストールしておけば、**C#** や **VB**、**ASP.NET**（ADO.NET や Entity Framework Core など）を利用して、アプリケーションを開発することができます。

もちろん、.NET ではない言語からでも、SQL Server on Linux にアクセスするアプリケーションは簡単に開発することができます。**Linux** や **Mac OS** をクライアントにする場合には、主に次の言語を利用できます。

- Java
- PHP
- Python
- node.js
- Ruby
- Go
- .NET Core（C#、VB、ASP.NET）

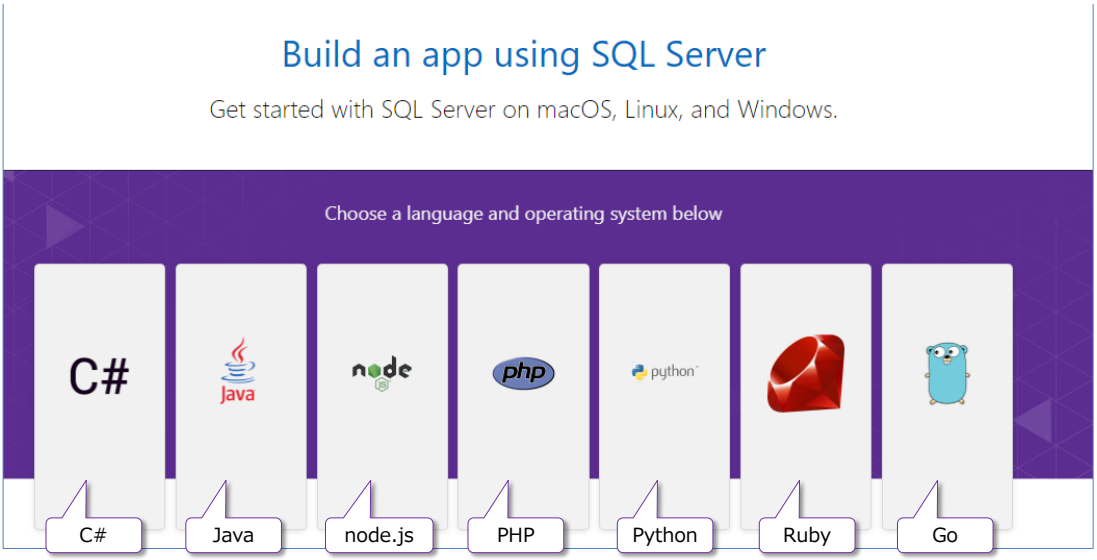
Java でも、**PHP** でも、**Python** でも、**node.js** でも、これらの言語から他のデータベース サーバーにアクセスするのと同じように、SQL Server on Linux にアクセスするアプリケーションを開発することができます。

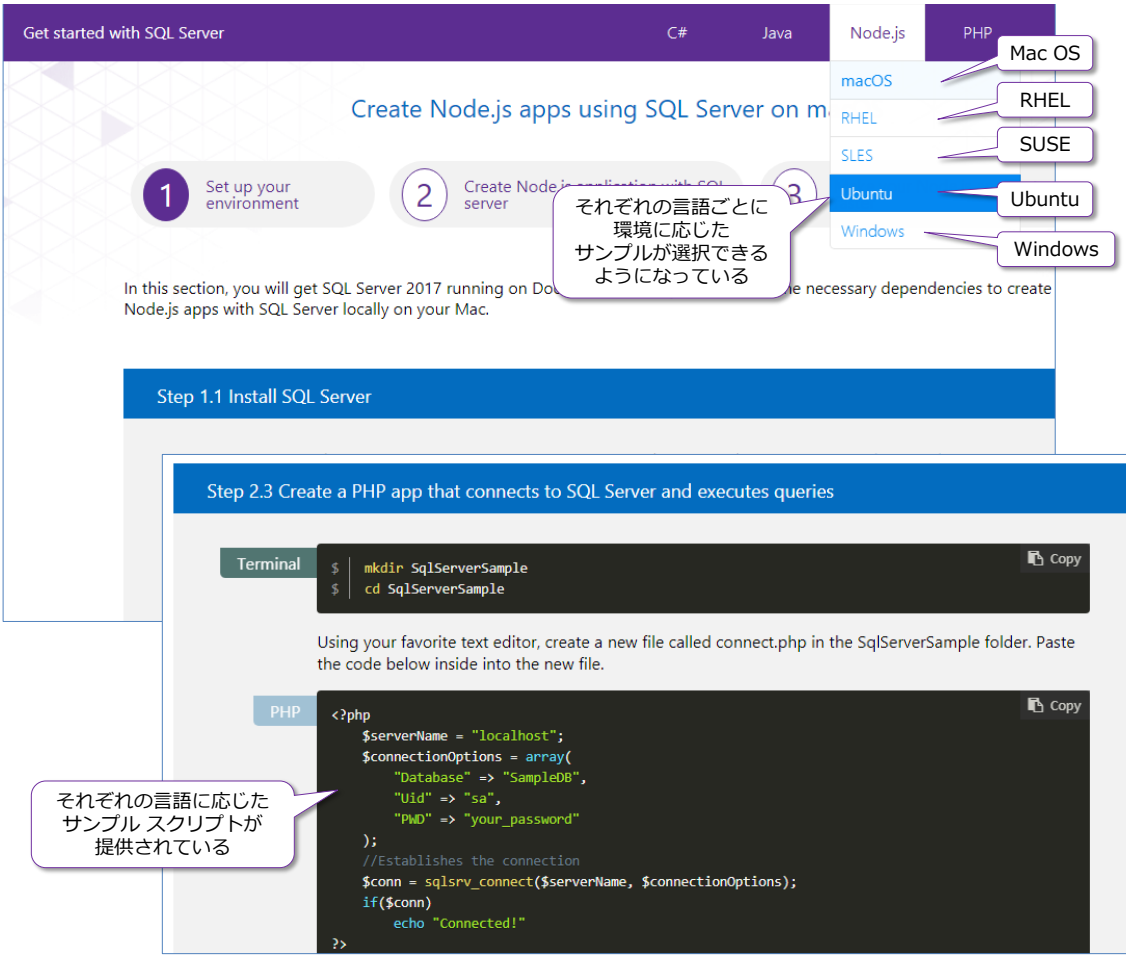
上記の言語と、SQL Server 2017 on Linux に接続するためのドライバー／接続モジュールの関係は、次のようになります。

- Java： **JDBC**（Microsoft **JDBC Driver 6.2** for SQL Server）
- PHP： **PHP Driver** for SQL Server
  (PHP Driver for SQL Server は Microsoft **ODBC Driver 13** for SQL Server が必要)
- Python： **pyodbc**（Microsoft **ODBC Driver 13** for SQL Server）
- node.js： **tedious**
- Ruby： **FreeTDS**
- Go： Microsoft **ODBC Driver 13** for SQL Server
- .NET Core（C#、VB、ASP.NET）： **ADO.NET**、**Entity Framework Core**

各言語から、どのように SQL Server on Linux にアクセスして、アプリケーションを開発するのかについては、以下の URL（公式ページ）でチュートリアルが公開されているので、こちらもぜひご覧いただければと思います。

Build an app using SQL Server
Get started with SQL Server on macOS, Linux, and Windows
http://aka.ms/sqldev





3.2 Java を利用したアプリケーション開発の概要

開発言語として **Java** を利用する場合には、**Java** から他のデータベース サーバーにアクセスするのと同じように、**JDBC** を利用して、SQL Server on Linux にアクセスするアプリケーションを開発することができます。

SQL Server 用の **JDBC** ドライバー（Type 4）は、以下の URL からダウンロードすることができます（執筆時点での最新版は **6.2** です）。

Microsoft JDBC Driver 6.2 for SQL Server
https://www.microsoft.com/ja-JP/download/details.aspx?id=55539

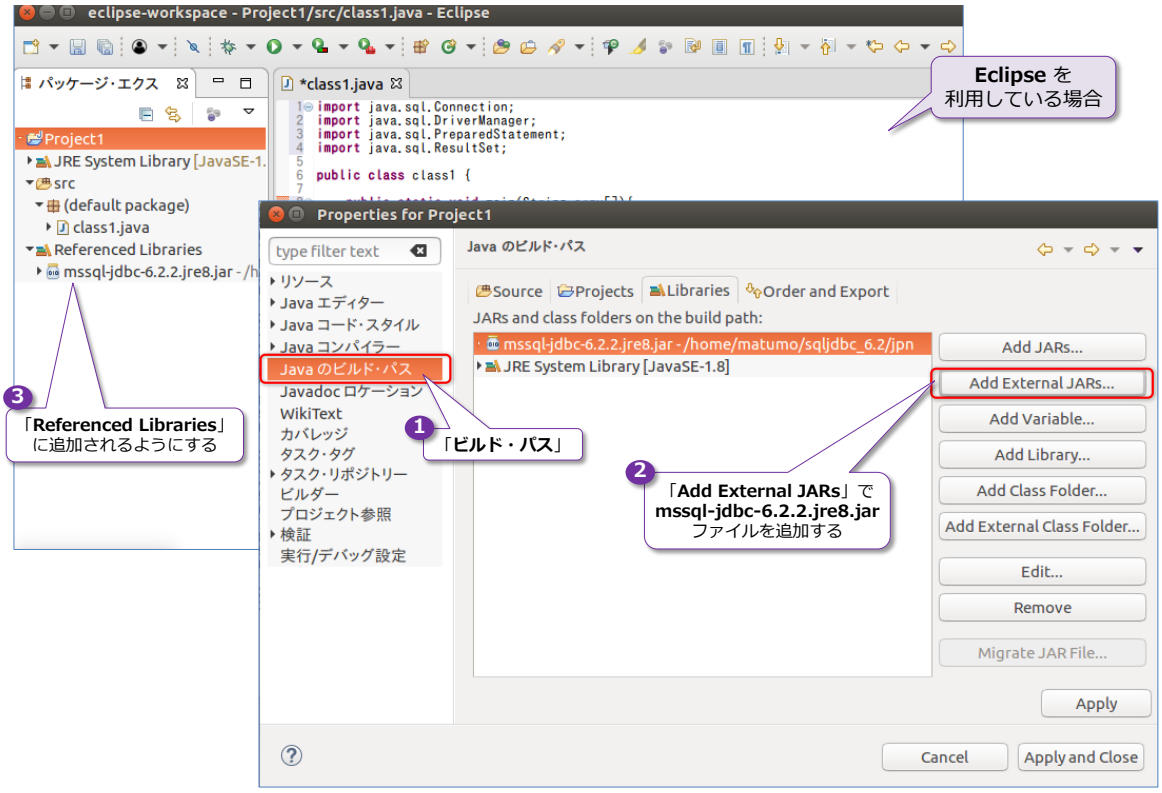
ダウンロードした .zip を展開

ダウンロードした **.zip** ファイル(**sqljdbc_6.2.2.0_jpn.tar.gz**)を展開すると、「**sqljdbc_6.2**」

ディレクトリが出来上がって、その下の「**jpn**」ディレクトリに「**mssql-jdbc-6.2.2.jre8.jar**」ファイルがあります。これが **JRE 8**（Java Runtime Environment 8）用の JDBC ドライバーになるので、この **.jar** ファイルに **CLASSPATH** を通しておけば、次のように JDBC を利用して SQL Server on Linux に接続することができます。

```
Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
String connectionUrl = "jdbc:sqlserver://localhost:ポート番号;" +
                       "databaseName=データベース名;user=sa;password=saのパスワード;";
Connection cn = DriverManager.getConnection(connectionUrl);
```

開発環境として **Eclipse** を利用している場合には、次のように**ビルドパス**で「**Add External JARs**」をクリックして、「**mssql-jdbc-6.2.2.jre8.jar**」ファイルを追加して、**Referenced Libraries** に追加されるようにします。

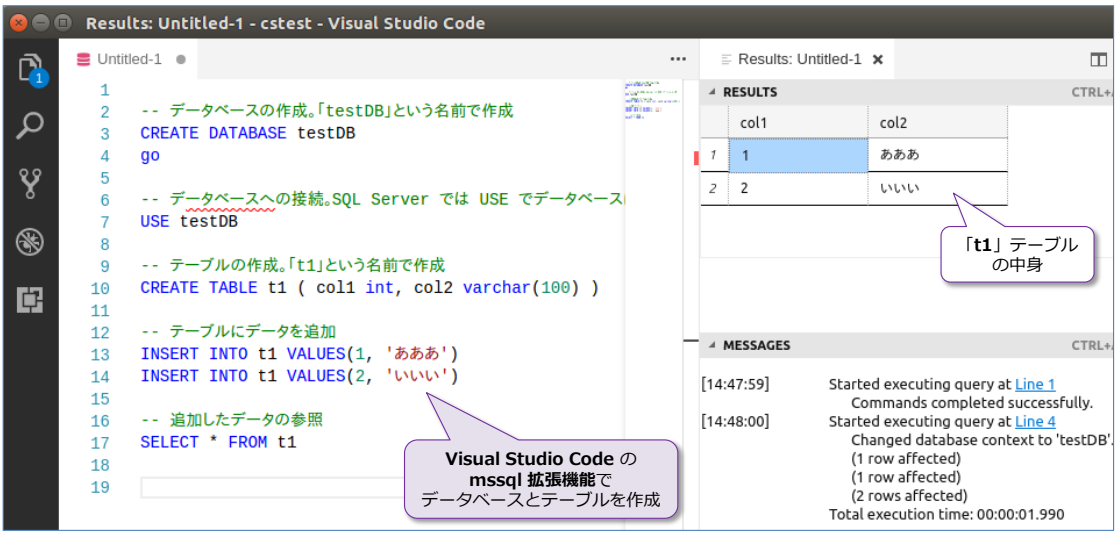


Java のコード例

次に、1 章で紹介したデータベースやテーブルの作成例の「**testDB**」データベースと「**t1**」テーブルを利用したコードで説明します。このデータベースとテーブルを作成していない場合には、以下の SQL ステートメントを、Visual Studio Code の mssql 拡張機能を利用して、実行しておいてください。

```
-- データベースの作成。「testDB」という名前で作成
CREATE DATABASE testDB
go
```

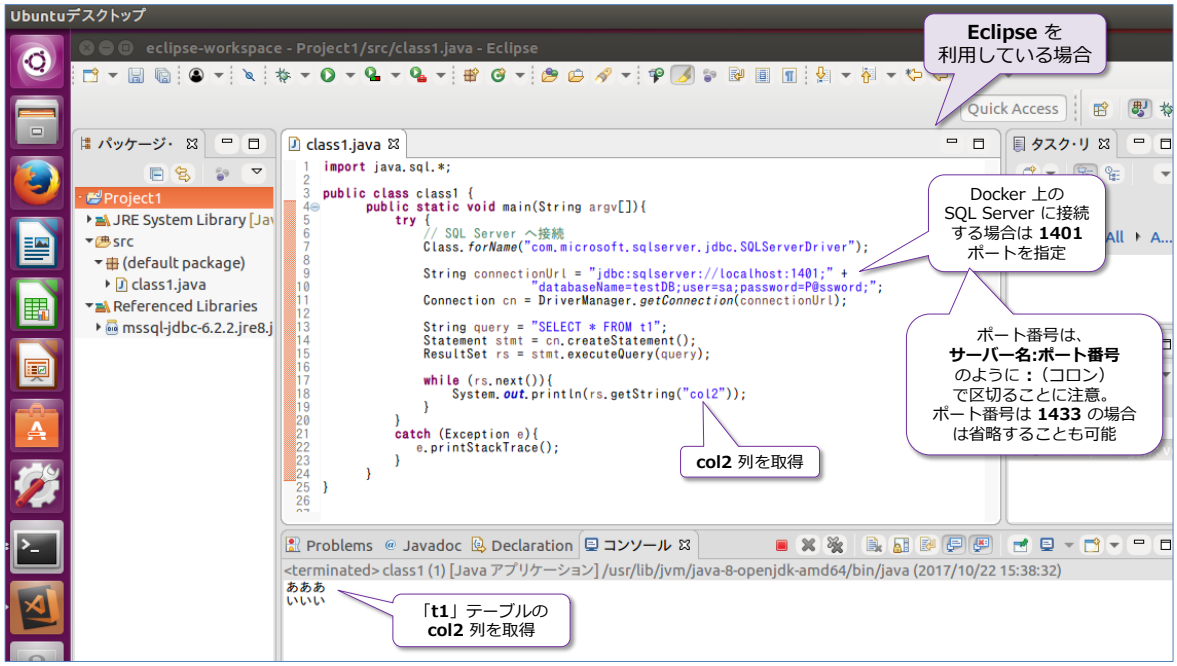
```
-- データベースへの接続。SQL Server では USE でデータベースに接続する
USE testDB

-- テーブルの作成。「t1」という名前で作成
CREATE TABLE t1 ( col1 int, col2 varchar(100) )

-- テーブルにデータを追加
INSERT INTO t1 VALUES(1, 'あああ')
INSERT INTO t1 VALUES(2, 'いいい')

-- 追加したデータの参照
SELECT * FROM t1
```



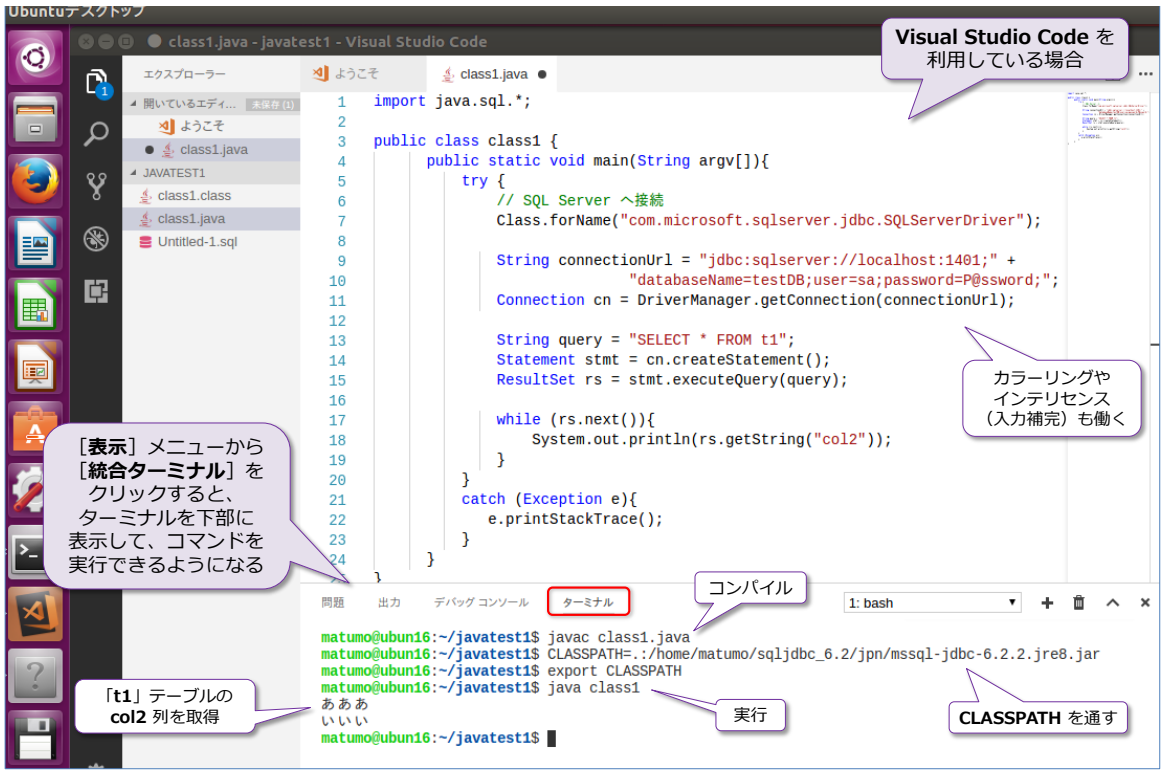
この「**testDB**」データベースの「**t1**」テーブルを参照するには、次のように **Java** コードを記述します。

```
import java.sql.*;

public class class1 {
  public static void main(String argv[]){
    try {
          // SQL Server へ接続、testDB データベースに接続
          Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
          String connectionUrl = "jdbc:sqlserver://サーバー名:ポート番号;" +
                        "databaseName=testDB;user=sa;password=sa のパスワード;";
          Connection cn = DriverManager.getConnection(connectionUrl);

          // t1 テーブルの参照
          String query = "SELECT * FROM t1";
          Statement stmt = cn.createStatement();
          ResultSet rs = stmt.executeQuery(query);

          while (rs.next()){
              // t1 テーブルの col2 列のデータを取得
              System.out.println(rs.getString("col2"));
```

```
            }
        }
        catch (Exception e){
            e.printStackTrace();
        }
    }
}
```



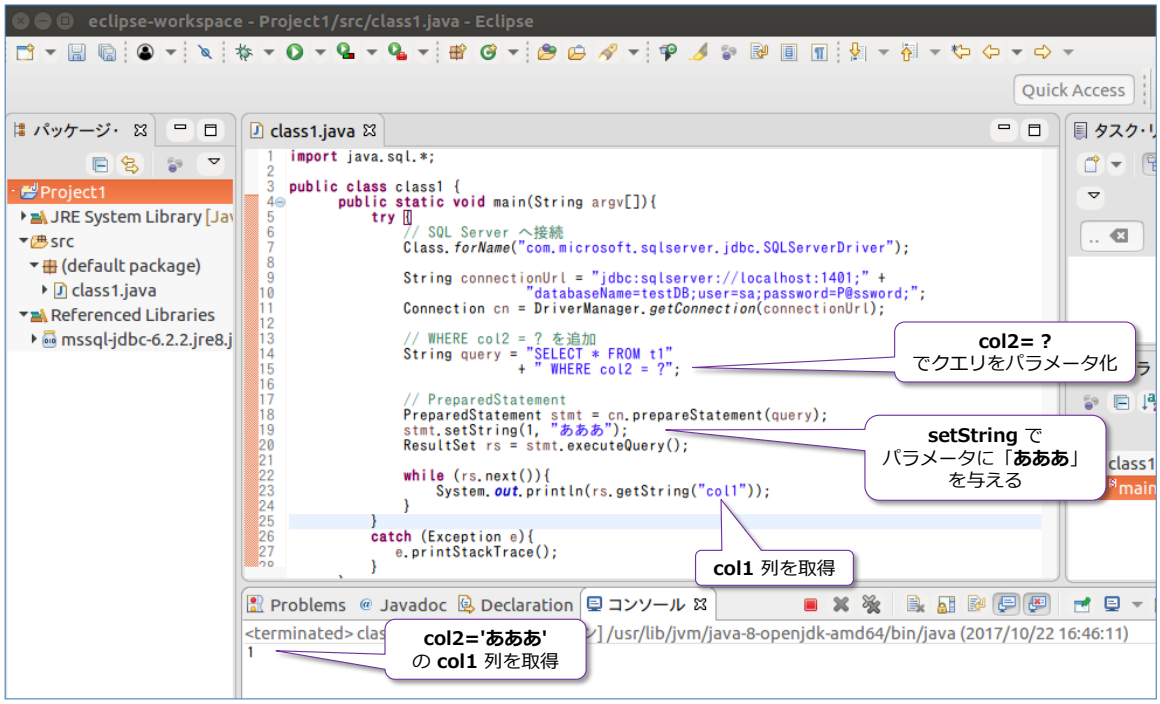
なお、**Visual Studio Code** を利用しても、**Java** のアプリケーションを開発できます。



Java のコード例２ ～PreparedStatement～

JDBC でのクエリのパラメーター化（**PreparedStatement**）に関しても、他のデータベースを操作するのと同様、次のように利用できます。

```
import java.sql.*;
              : （中略）
    try {
          // SQL Server へ接続、testDB データベースに接続
          Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
          String connectionUrl = "jdbc:sqlserver://サーバー名:ポート番号;" +
                        "databaseName=testDB;user=sa;password=sa のパスワード;";
          Connection cn = DriverManager.getConnection(connectionUrl);

          // WHERE col2 = ? を追加
          String query = "SELECT * FROM t1"
                       + " WHERE col2 = ?";

          // PreparedStatement。col2 = ? の ? に あああ をセット
          PreparedStatement stmt = cn.prepareStatement(query);
          stmt.setString(1, "あああ");
          ResultSet rs = stmt.executeQuery();

          while (rs.next()){
              // col1 列のデータを取得
              System.out.println(rs.getString("col1"));
          }
      }
              : （以下略）
```

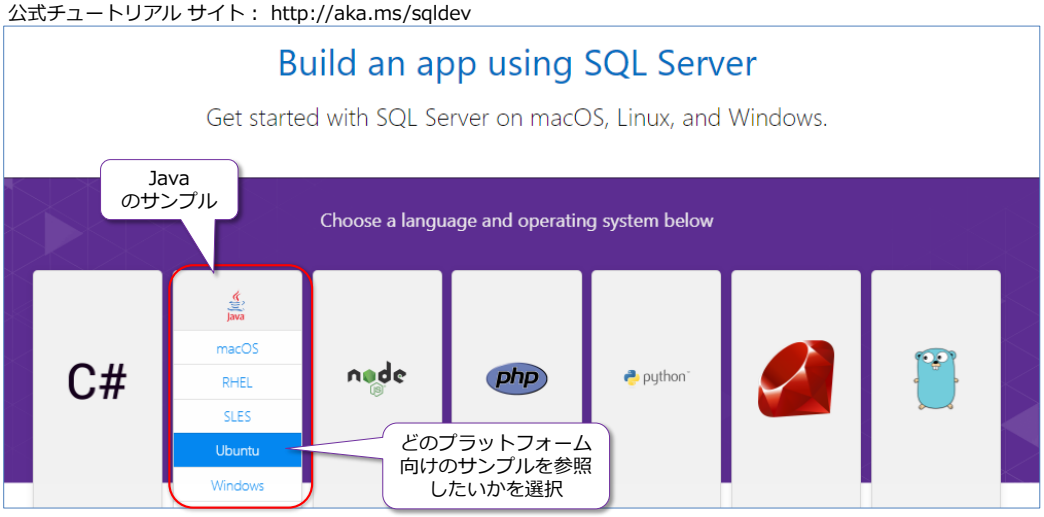


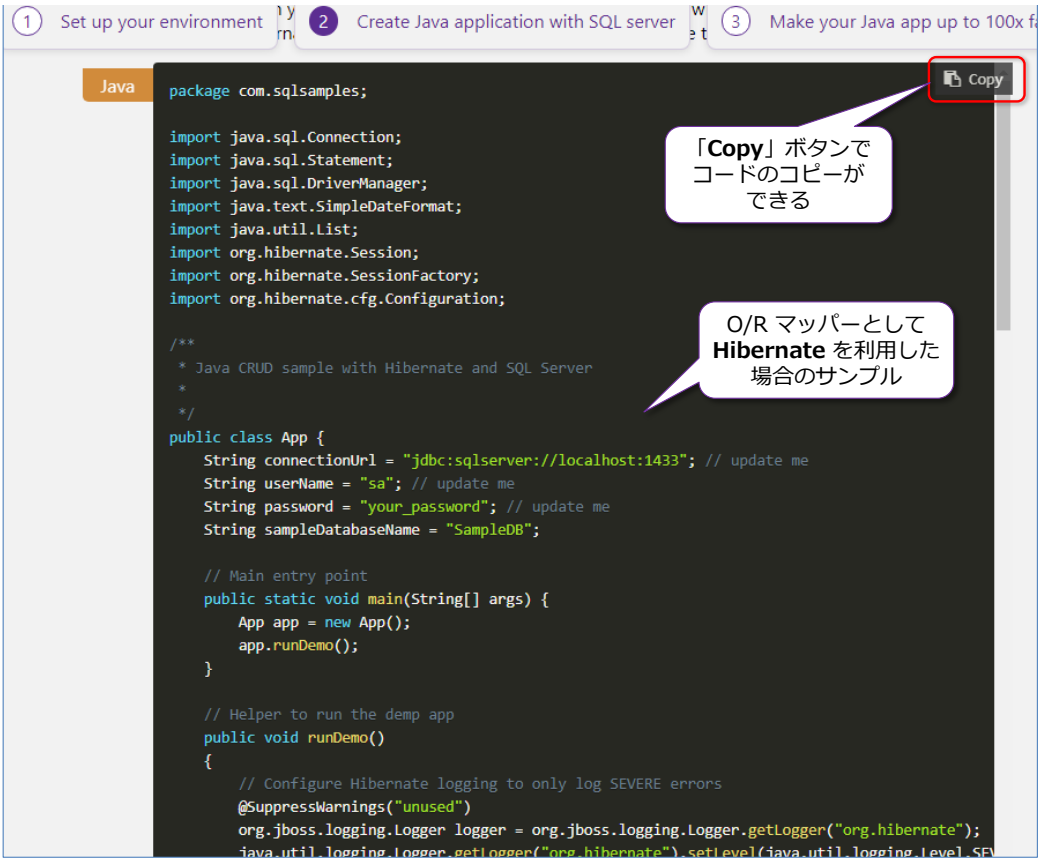
このように、**SQL Server on Linux** を利用していても、他のデータベース サーバーに対する Java プログラミングと同様、JDBC を利用して、全く同じようにプログラミングできます。

Maven や Hibernate を利用する場合

Java の開発環境として **Maven** を利用する場合や、O/R マッパーとして **Hibernate** を利用する場合については、前掲のチュートリアル サイト（**aka.ms/sqldev**）にサンプルがあるので、こちらが参考になると思います。

公式チュートリアル サイト： http://aka.ms/sqldev



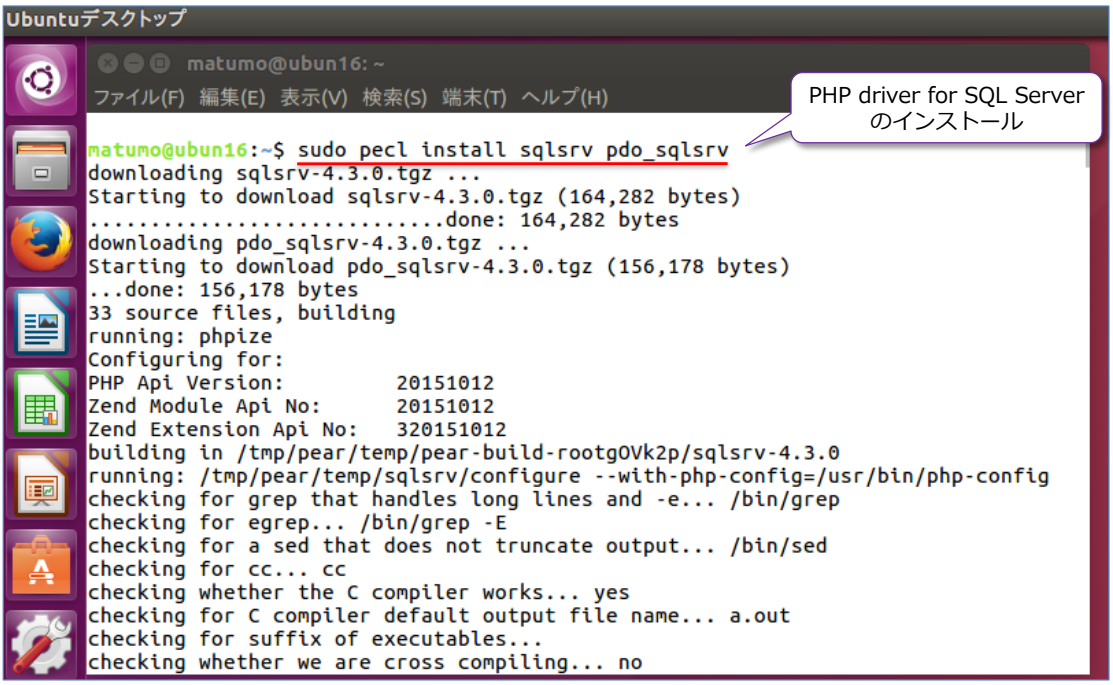


3.3 PHP を利用したアプリケーション開発の概要

PHP を利用して、SQL Server にアクセスするアプリケーションを開発するには、「**PHP Driver for SQL Server**」を利用します。

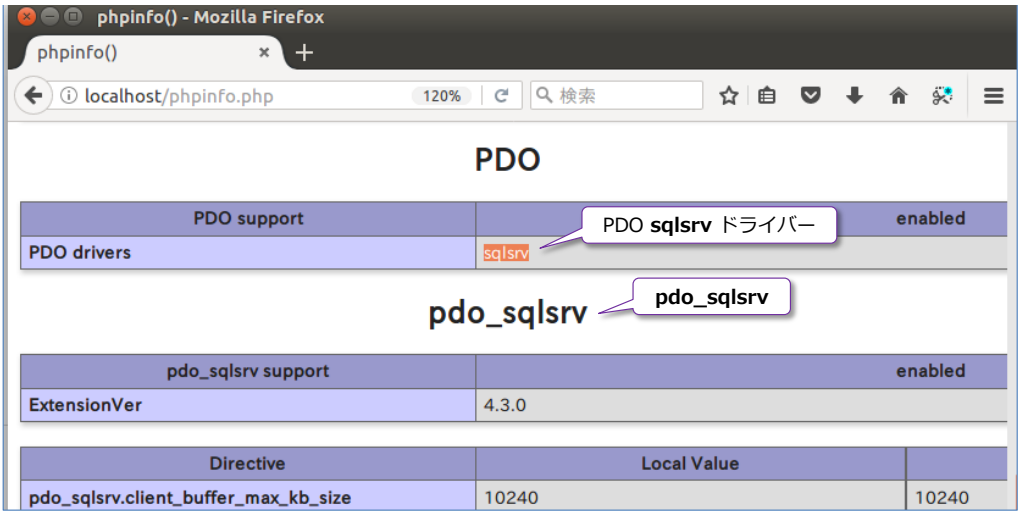
PHP Driver for SQL Server のインストール方法は、次のとおりです（**Ubuntu** や **RHEL**、**SUSE** を利用している場合）。

```
# PHP Driver for SQL Server のインストール (PHP の pecl を利用)
sudo pecl install sqlsrv pdo_sqlsrv

# php.ini の修正
sudo echo "extension=pdo_sqlsrv.so" >> `php --ini | grep "Loaded Configuration" | sed -e "s|.*:¥s*||"`
sudo echo "extension=sqlsrv.so" >> `php --ini | grep "Loaded Configuration" | sed -e "s|.*:¥s*||"`

# Ubuntu 上の Apache2 の場合は以下の php.ini も修正
sudo echo "extension=pdo_sqlsrv.so" >> /etc/php/7.0/apache2/php.ini
sudo echo "extension=sqlsrv.so" >> /etc/php/7.0/apache2/php.ini
# Ubuntu 上の Apache2 の再起動
sudo service apache2 restart
```



PHP Driver for SQL Server が正しくインストールされたかどうかは、次のように **phpinfo()** で確認することができます。

```
<?php
   phpinfo();
?>
```



phpinfo の結果を **Ctrl+F** で「**sqlsrv**」で検索して、上記の結果が表示されれば、**PHP Driver for SQL Server** が正しくインストールされています。

> **Note： ODBC ドライバーのインストール（PHP Driver for SQL Server の必須要件）**
>
> ODBC ドライバーは、本自習書の Step 2.2 の手順で sqlcmd ツールをインストールしている場合には、インストール済みになりますが、もし、PHP Driver for SQL Server がうまく動作しない場合には、ODBC ドライバーが正しくインストールされていない可能性があるので、前掲のチュートリアル サイト（**aka.ms/sqldev**）の PHP セクションの Step 1 を参考にインストールしてみてください。

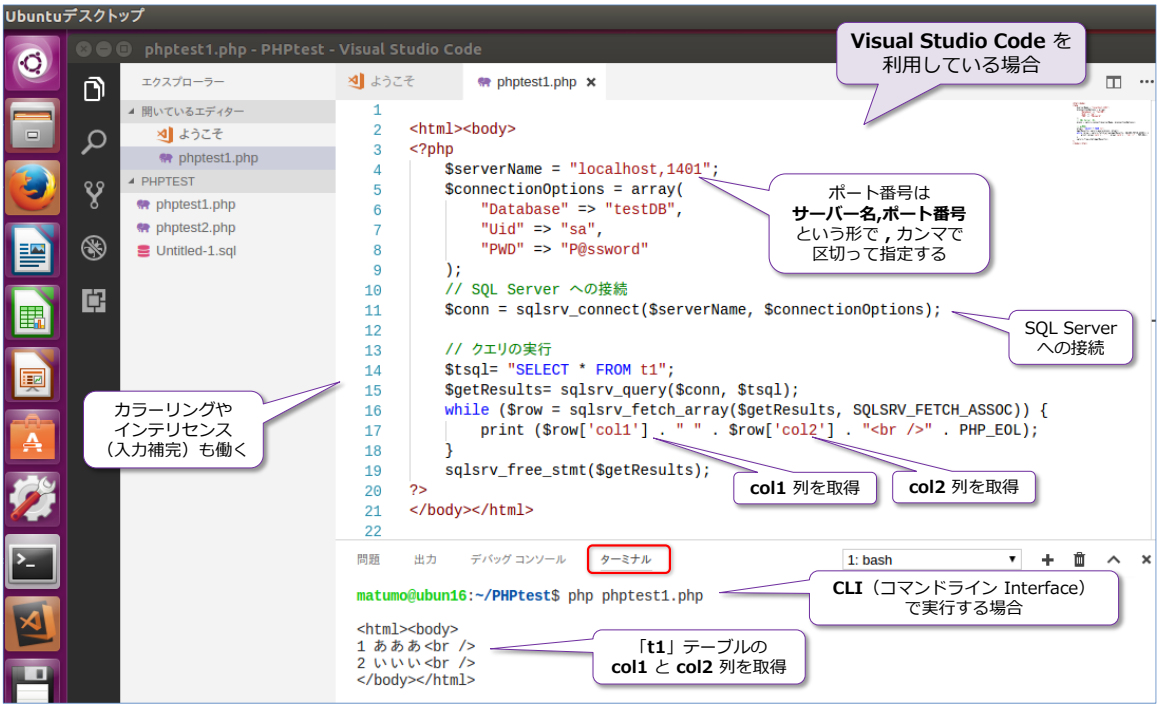
PHP のコード例

SQL Server on Linux にアクセスする PHP のコード例は、次のとおりです。

```
<html><body>
<?php
    $serverName = "サーバー名,ポート番号";
    $connectionOptions = array(
        "Database" => "データベース名",
        "Uid" => "sa",
        "PWD" => "sa のパスワード"
    );
    // SQL Server への接続
    $conn = sqlsrv_connect($serverName, $connectionOptions);

    // クエリの実行
    $tsql = "SELECT * FROM t1";
    $getResults = sqlsrv_query($conn, $tsql);
    while ($row = sqlsrv_fetch_array($getResults, SQLSRV_FETCH_ASSOC)) {
        print ($row['col1'] . " " . $row['col2'] . "<br />" . PHP_EOL);
    }
    sqlsrv_free_stmt($getResults);
?>
</body></html>
```

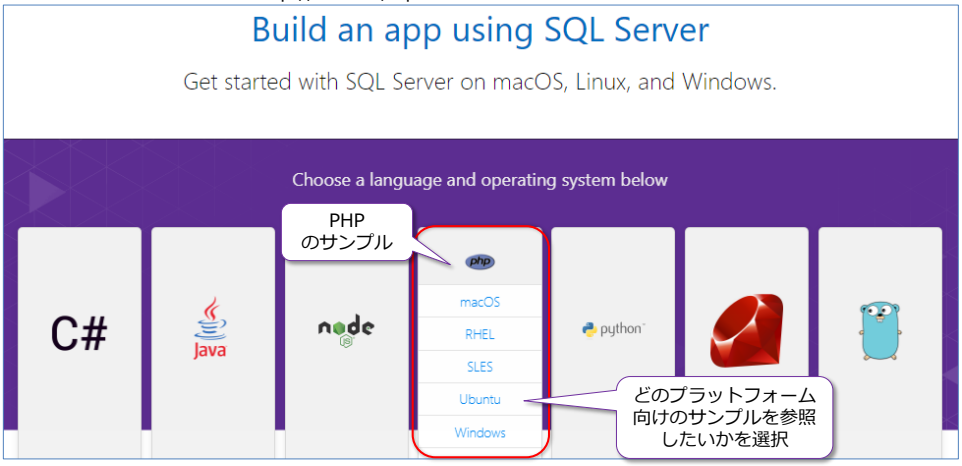
sqlsrv_connect で SQL Server へ接続して、**sqlsrv_query** でクエリを実行、**sqlsrv_fetch_array** で行のフェッチ（クエリ結果を 1 行読み取り）することができます。

Visual Studio Code でコードを記述している例



その他の PHP の利用方法については、前掲のチュートリアル サイト（**aka.ms/sqldev**）にサンプルがあるので、こちらが参考になると思います。

公式チュートリアル サイト： http://aka.ms/sqldev



3.4 Python を利用したアプリケーション開発の概要

開発言語として **Python** を利用して、SQL Server にアクセスするアプリケーションを開発するには、「**pyodbc**」などを利用します。

ODBC ドライバー（**ODBC Driver 13 for SQL Server**）は、Step 2.2 で sqlcmd ツールをインストールしている場合には、インストール済みになりますが、インストールしていない場合は、次のようにインストールできます（1 行目の curl ～ は、改行なしで記述してください）。

```
# Ubuntu の場合の ODBC ドライバーと sqlcmd ツールのインストール
curl https://packages.microsoft.com/config/ubuntu/16.04/prod.list | sudo tee /etc
/apt/sources.list.d/mssql-tools.list
sudo apt-get update
sudo ACCEPT_EULA=Y apt-get install mssql-tools
sudo apt-get install unixodbc-dev
echo 'export PATH="$PATH:/opt/mssql-tools/bin"' >> ~/.bash_profile
echo 'export PATH="$PATH:/opt/mssql-tools/bin"' >> ~/.bashrc
source ~/.bashrc
```

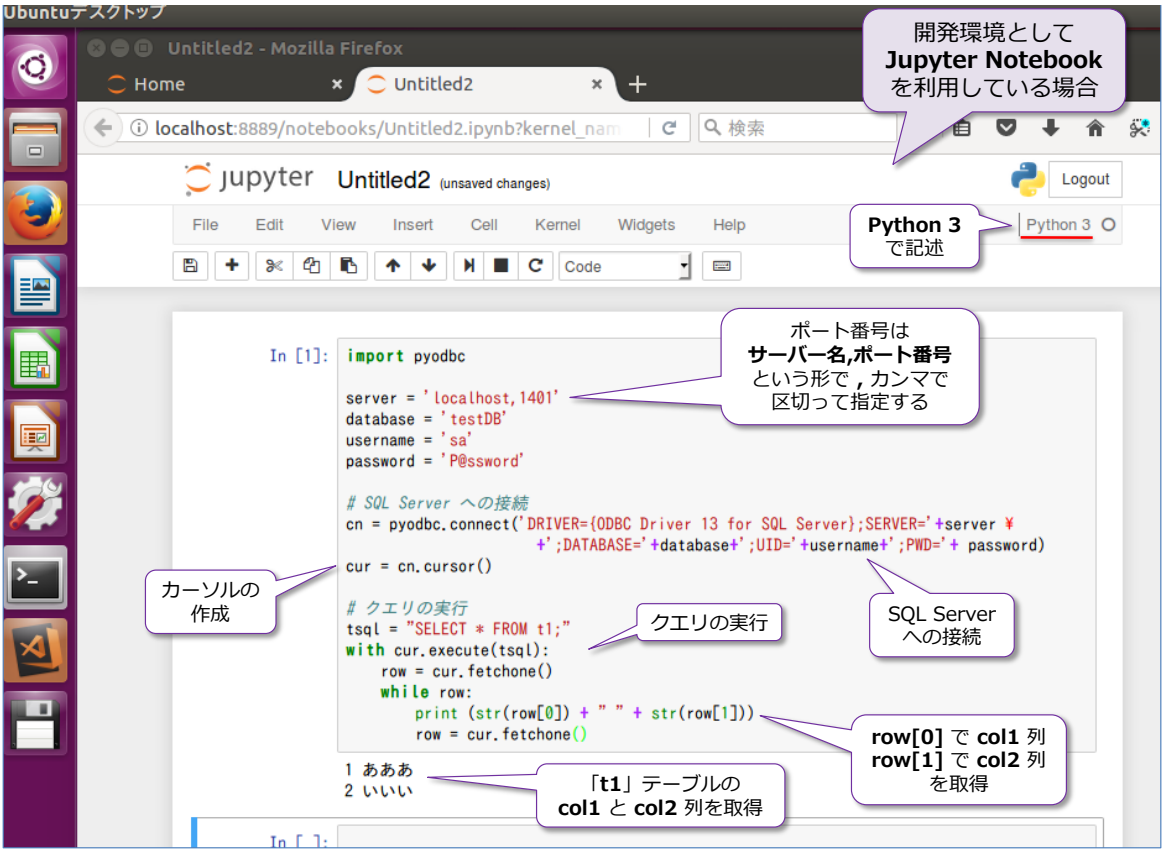
Python のコード例（pyodbc）

SQL Server にアクセスする Python のコード例は、次のとおりです（Python 3.6 の場合）。

```
import pyodbc

server = 'サーバー名,ポート番号'
database = 'データベース名'
username = 'sa'
password = 'sa のパスワード'

# SQL Server への接続
cn = pyodbc.connect('DRIVER={ODBC Driver 13 for SQL Server};SERVER='+server ¥
                    +';DATABASE='+database+';UID='+username+';PWD='+ password)
cur = cn.cursor()

# クエリの実行
tsql = "SELECT * FROM t1;"
with cur.execute(tsql):
    row = cur.fetchone()
    while row:
        print (str(row[0]) + " " + str(row[1]))
        row = cur.fetchone()
```

ODBC Driver 13 for SQL Server では、接続文字列に「**DRIVER=**」で「**{ODBC Driver 13 for SQL Server}**」、「**SERVER=**」にサーバー名とポート番号（カンマ区切り）、「**DATABASE=**」にデータベース名、「**UID=**」に接続ユーザー名、「**PWD=**」にユーザーのパスワードを指定することで、SQL Server に接続することができます（接続文字列は、**pyodbc.connect** の引数に与え

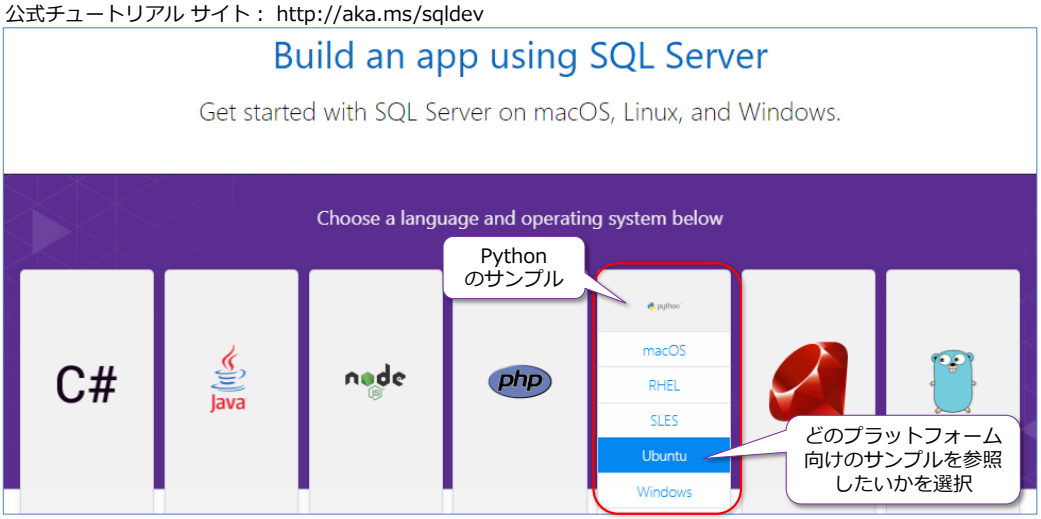
ます)。クエリの実行は、**cn.cursor()** でカーソルを作成して、**cur.execute(～)** に SQL ステートメントを与えることで、実行できます。



SELECT ステートメントの実行結果は、**cur.fetchone()** で 1 行フェッチ（読み取り）、**row**[**0**] で 1 列目（**col1**）、**row**[**1**] で 2 列目（**col1**）を取得できます。

その他の Python の利用方法については、前掲のチュートリアル サイト（**aka.ms/sqldev**）にサンプルがあるので、こちらが参考になると思います。

公式チュートリアル サイト： http://aka.ms/sqldev



3.5 node.js を利用したアプリケーション開発の概要

node.js を利用して、SQL Server にアクセスするアプリケーションを開発するには、「**tedious**」などを利用します。

tedious をインストールする方法は、次のとおりです。

```
# node.js をインストールしていない場合は、node.js のインストール (Ubuntu の場合)
sudo apt-get install nodejs
sudo apt-get install npm
sudo apt install nodejs-legacy

# アプリケーション用のフォルダーを作成して、そこへ移動。testapp1 という名前の場合
mkdir testapp1
cd testapp1

# npm を利用して tedious をインストール
npm install tedious
```

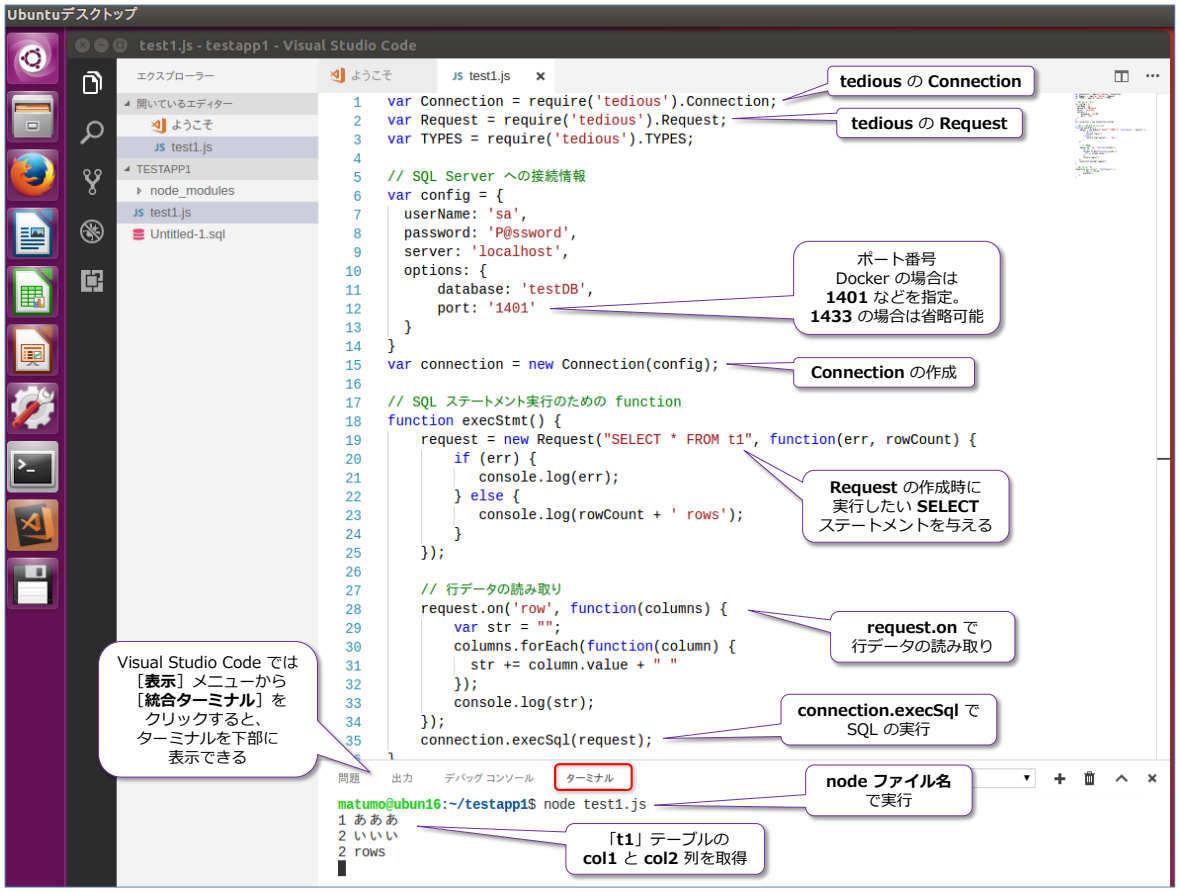
tedious のコード例

tedous を利用して、SQL Server にアクセスするには、次のようにコードを記述します。

```
var Connection = require('tedious').Connection;
var Request = require('tedious').Request;
var TYPES = require('tedious').TYPES;

// SQL Server への接続情報
var config = {
  userName: 'sa',
  password: 'sa のパスワード',
  server: 'サーバー名',
  options: {
      database: 'データベース名',
      port: 'ポート番号'
  }
}
var connection = new Connection(config);

// SQL ステートメント実行のための function
function execStmt() {
    request = new Request("SELECT * FROM t1", function(err, rowCount) {
        if (err) {
           console.log(err);
        } else {
           console.log(rowCount + ' rows');
        }
    });
```

```
    // 行データの読み取り
    request.on('row', function(columns) {
        var str = "";
        columns.forEach(function(column) {
          str += column.value + " "
        });
        console.log(str);
    });
    connection.execSql(request);
}

// SQL Server への接続
connection.on('connect', function(err) {
      // SQL ステートメントの実行
      execStmt();
    }
);
```

Visual Studio Code を利用してコードを記述している例

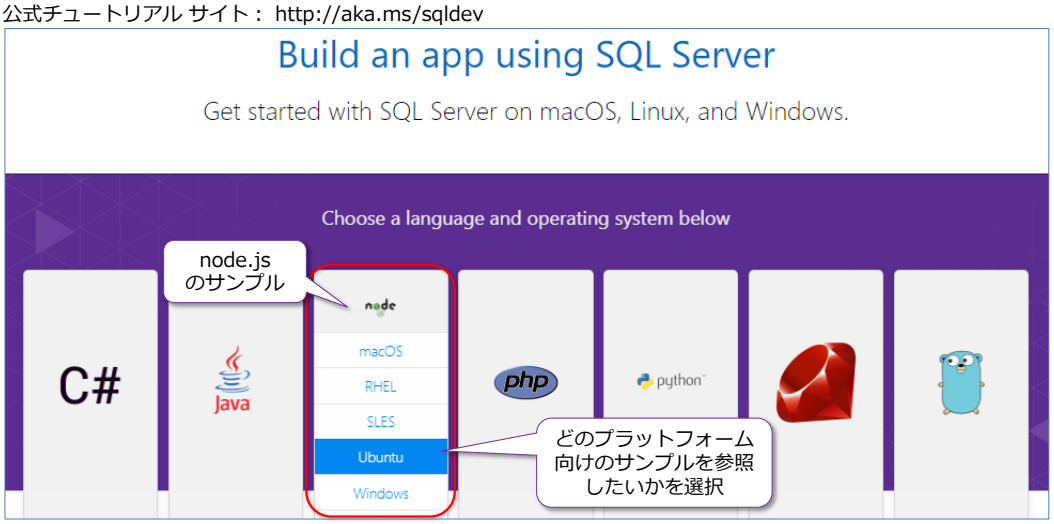


コードを記述した後は、次のようにファイルを実行できます（**test1.js** という名前で、tedious をインストールした **testapp1** ディレクトリにファイルを保存した場合）。

```
# 実行（testapp1 ディレクトリに移動した状態で以下を実行）
node test1.js
```

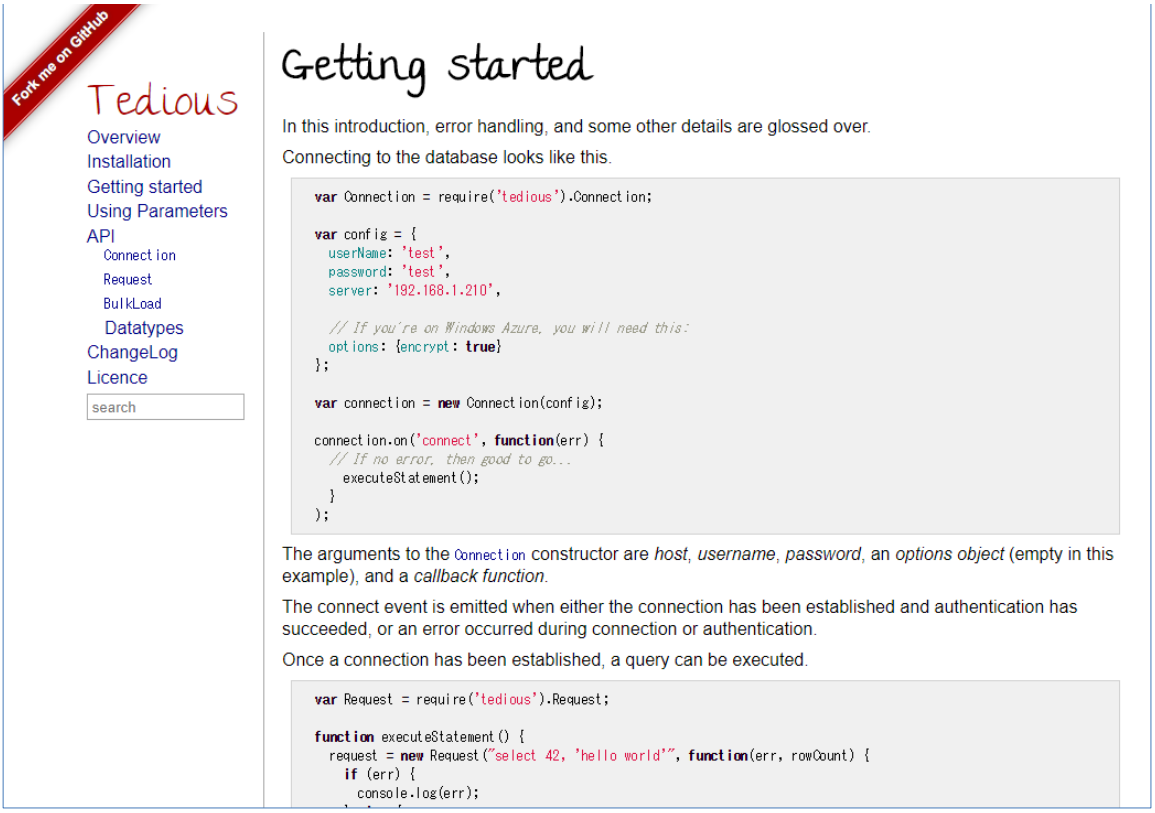
その他の node.js の利用方法（**Sequelize ORM** など）については、前掲のチュートリアル サイト（**aka.ms/sqldev**）にサンプルがあるので、こちらが参考になると思います。

公式チュートリアル サイト： http://aka.ms/sqldev



tedious については、以下の公式ページが参考になると思います。

tedious Getting started
http://tediousjs.github.io/tedious/getting-started.html



3.6 .NET Core を利用したアプリケーション開発の概要

.NET Core は、**Linux** や **Mac OS** などのクロス プラットフォームで利用することができる、**オープンソース**の .NET Framework バージョンです。**.NET Core** を利用すれば、Windows 環境でお馴染みの **C#** や **VB**、**ASP.NET** を利用して、Linux／Mac OS でもアプリケーションを開発することができます（.NET でのデータ アクセスで定番になっている **ADO.NET** や **Entity Framework** を利用して SQL Server にアクセスするアプリケーションを開発できます）。

.NET Core 2.0（執筆時点での最新バージョン）がサポートしている Linux バージョンや、前提条件、.NET Core 2.0 のインストール方法については、以下の URL（公式ページ）に詳しく記載されています。

Linux における .NET Core の前提条件
https://docs.microsoft.com/ja-jp/dotnet/core/linux-prerequisites?tabs=netcore2x


サポートされている Linux バージョン

.NET Core 2.x　.NET Core 1.x

.NET Core 2.0 は、1 つのオペレーティング システムとして Linux を扱います。 サポートされている Linux ディストリビューション用に、1 つの Linux ビルド (チップ アーキテクチャあたり) があります。

.NET Core 2.x は、次の Linux 64 ビット (x86_64 または amd64) ディストリビューション/バージョンでサポートされています。

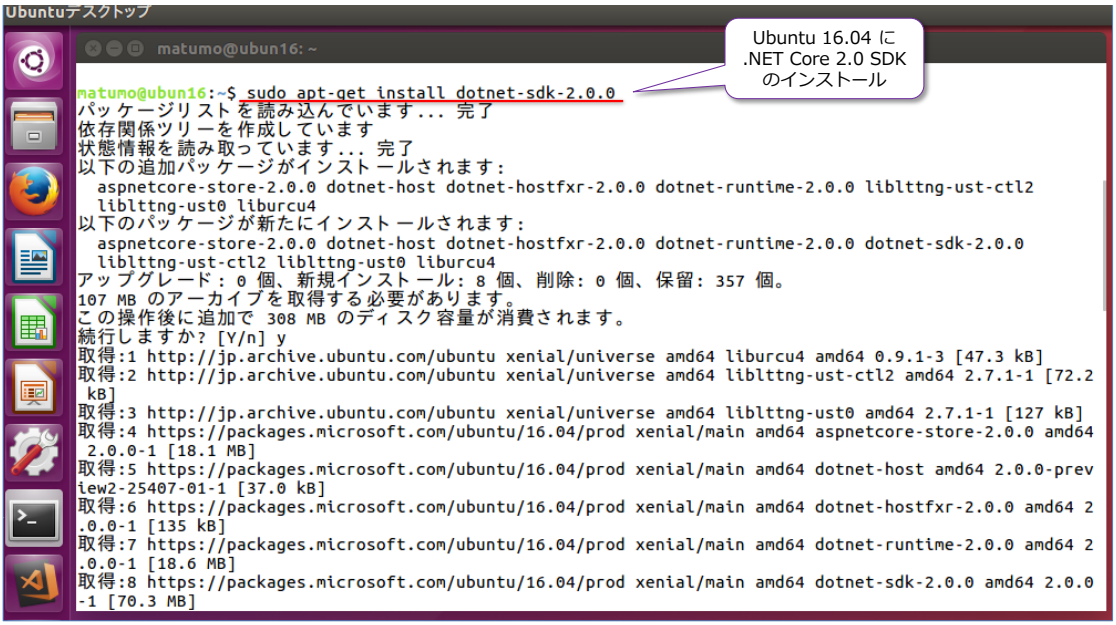
- Red Hat Enterprise Linux 7
- CentOS 7
- Oracle Linux 7
- Fedora 25、Fedora 26
- Debian 8.7 以降のバージョン
- Ubuntu 17.04、Ubuntu 16.04、Ubuntu 14.04
- Linux Mint 18、Linux Mint 17
- openSUSE 42.2 以降のバージョン
- SUSE Enterprise Linux (SLES) 12 SP2 以降のバージョン

.NET Core 2.x がサポートされているオペレーティング システム (サポートされている OS バー

.NET Core 2.0 のインストール

.NET Core 2.0 のインストール方法は、上記のページに詳しく記載されていますが、**Ubuntu 16.04** を利用している場合には、次のようにインストールすることができます。

```
# Ubuntu 16.04 の場合の .NET Core 2.0 SDK のインストール
curl https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor > microsoft.gpg
sudo mv microsoft.gpg /etc/apt/trusted.gpg.d/microsoft.gpg

# 以下は改行なしで 1行で記述します
sudo sh -c 'echo "deb [arch=amd64] https://packages.microsoft.com/repos
/microsoft-ubuntu-xenial-prod xenial main" > /etc/apt/sources.list.d/dotnetdev.list'

sudo apt-get update
sudo apt-get install dotnet-sdk-2.0.0
```



.NET Core 2.0 の利用方法（Visual Studio Code がお勧め）

.NET Core 2.0 を利用する場合には、次の流れになります。

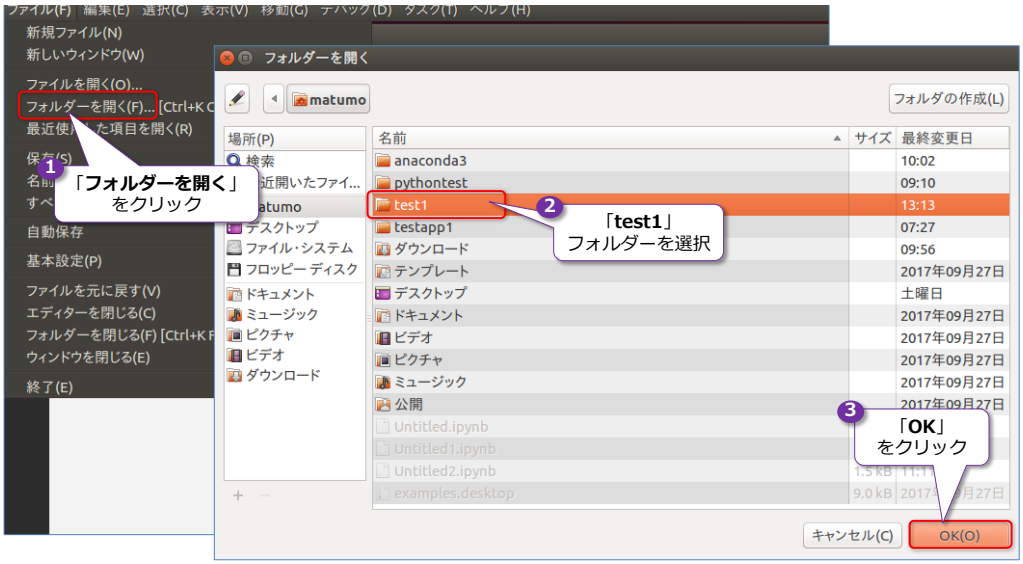
- **フォルダーを作成**して、そこへ移動する（cd でカレント ディレクトリを変更）
- **dotnet new ～** と入力して、プロジェクトのひな形を作成する。
  コンソール アプリケーションを作成する場合は、～ に **console** と指定、
  VB を利用する場合には **-lang vb** を追加する（-lang 省略時は C# になる）
- **Program.cs**（C# コード）や **～.csproj**（プロジェクト ファイル）を修正する
- **dotnet restore** を実行する（～.csproj を変更した場合）
- **dotnet run** でコードをコンパイル／実行する

.NET Core 2.0 では、まずフォルダーを作成します（フォルダー名は、アプリケーションに付けるプロジェクト名に相当するものになります）。

```
# フォルダーを作成して、そこに移動する。「test1」という名前のフォルダーを作成
mkdir test1
cd test1
```

次に、「**dotnet new console**」と入力して、コンソール アプリケーションを作成しますが、ここからの手順は、**Visual Studio Code** を利用するのがお勧めです。

Visual Studio Code を利用する場合は、次のように［**ファイル**］メニューから［**フォルダーを開く**］をクリックして、作成したフォルダー「**test1**」を選択します。

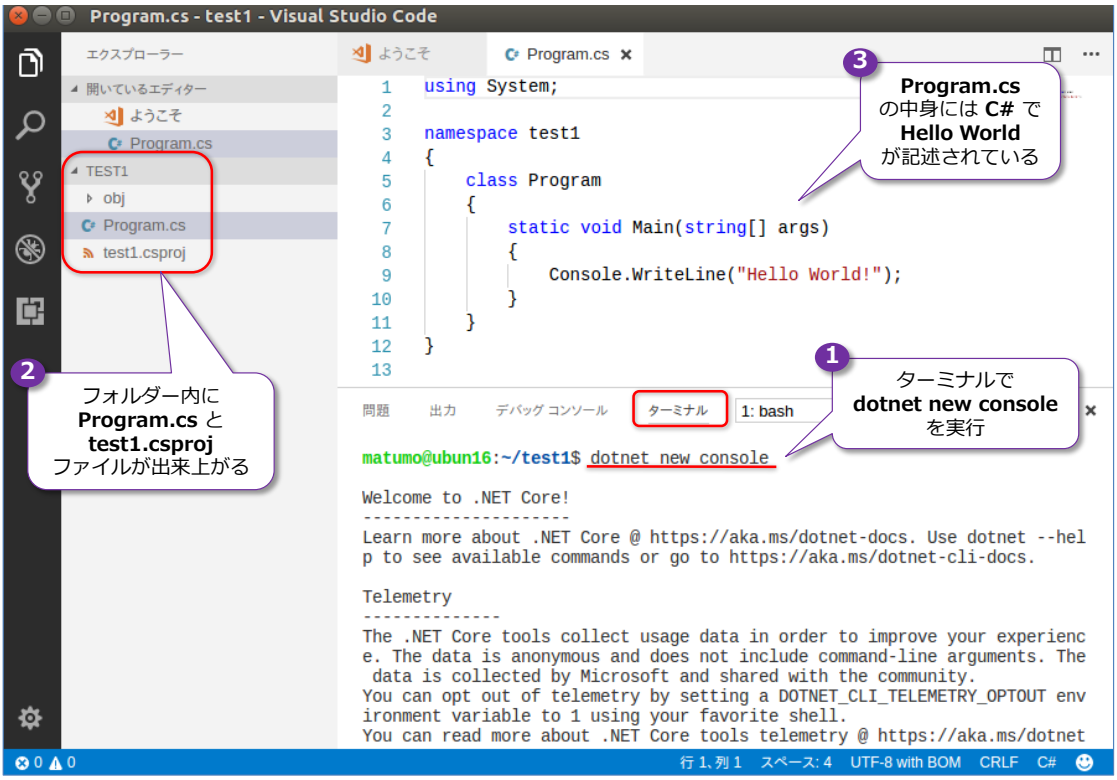


フォルダーを開いた後は、次のように [**表示**] メニューから [**統合ターミナル**] をクリックします。

これによって、Visual Studio Code 内にターミナルを表示できるようになり、ここでコマンドを実行することができます。また、フォルダーを開いておいたことで、そのフォルダーにカレント ディレクトリを移動してくれています（test1 フォルダーに cd した状態になっています）。

次に、ターミナルで「**dotnet new console**」と入力して、コンソール アプリケーションを作成します。

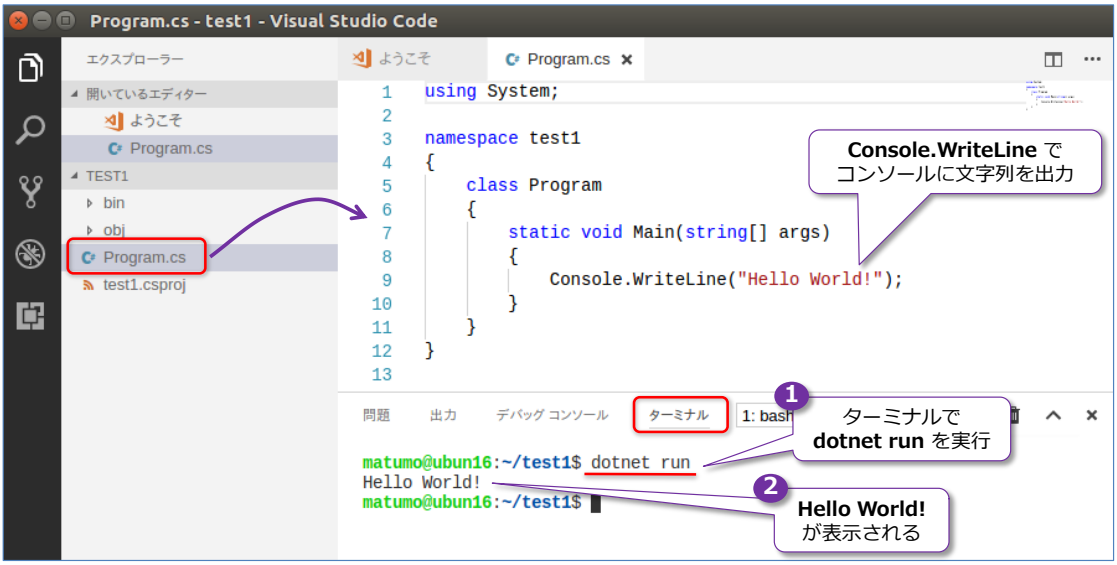
```
# コンソール アプリケーションのひな形を作成
dotnet new console
```



このコマンドによって、フォルダー内に「**Program.cs**」と「**test1.csproj**」ファイルが出来上がります。「**Program.cs**」ファイルを開くと、**C#** で **Hello World** を表示するコードが記述されています。

次に、この Hello World を実行するために、ターミナルで「**dotnet run**」を実行します。

```
# コンソール アプリケーションの実行
dotnet run
```



「**dotnet run**」を実行すると、「**Program.cs**」ファイルがコンパイルされて、それが実行されるので、**Hello World!** という文字列がターミナルに出力されたことを確認できます。このように、Visual Studio Code を利用すると、.NET Core でのコードを記述して、実行することを簡単に行

うことができます（コードのカラーリングはもちろん、インテリセンスも効きます）。

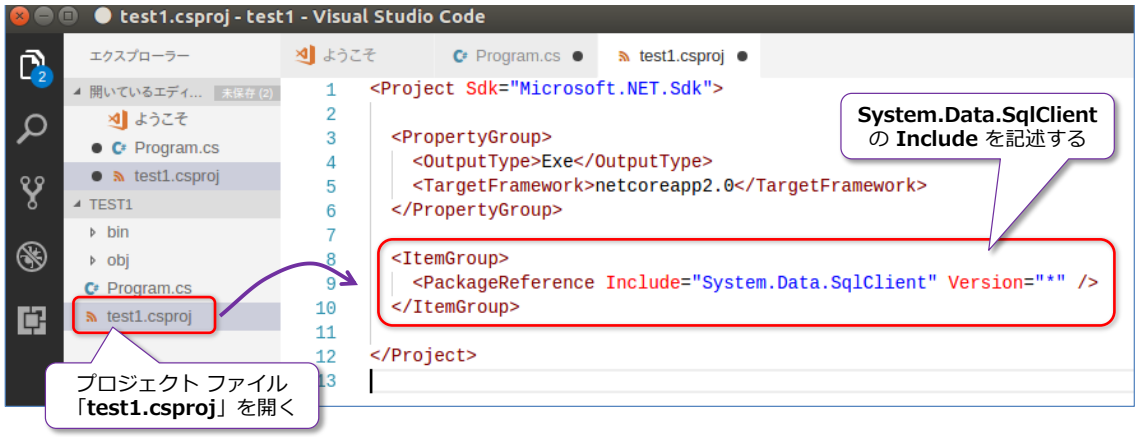
.NET Core 2.0 のコード例（C#、ADO.NET）

SQL Server on Linux にアクセスする .NET Core 2.0 のコード例は、次のようになります（Windows の場合のデータ アクセスと同様、ADO.NET を同じように利用できます）。

```
using System.Data;
using System.Data.SqlClient;
        : （略）
   static void Main( 略 )
   {
        string cnstr = "Server=ｻｰﾊﾞｰ名,ﾎﾟｰﾄ番号;Database=testDB;User ID=sa;Password=ﾊﾟｽﾜｰﾄﾞ;";
        using (SqlConnection cn = new SqlConnection(cnstr)) {
           using (SqlCommand cmd = new SqlCommand()) {
              try {
                    // SQL Server への接続
                    cn.Open();
                    cmd.Connection = cn;
                    cmd.CommandText = "SELECT * FROM t1";
                    // クエリの実行
                    using (SqlDataReader dr = cmd.ExecuteReader()) {
                        while (dr.Read()) {
                           Console.WriteLine(dr["col1"].ToString() + " "
                                        + dr["col2"].ToString());
                        }
                    }
                } catch (Exception ex) {
                    Console.WriteLine(ex.Message);
                } finally {
                    cn.Close();
                }
            }
        }
```

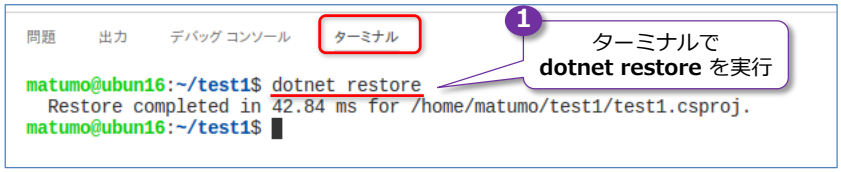
コードを記述後は、「**dotnet run**」を実行しますが、このままだと、**SqlConnection** などが見つからない（could not fount）となってコンパイル エラーになります。これを回避するには、プロジェクト ファイル（**test1.csproj**）に、**System.Data.SqlClient** を利用（**Include**）することを追記するようにします。具体的には、次のように記述します。

```
  <ItemGroup>
    <PackageReference Include="System.Data.SqlClient" Version="*" />
  </ItemGroup>
```

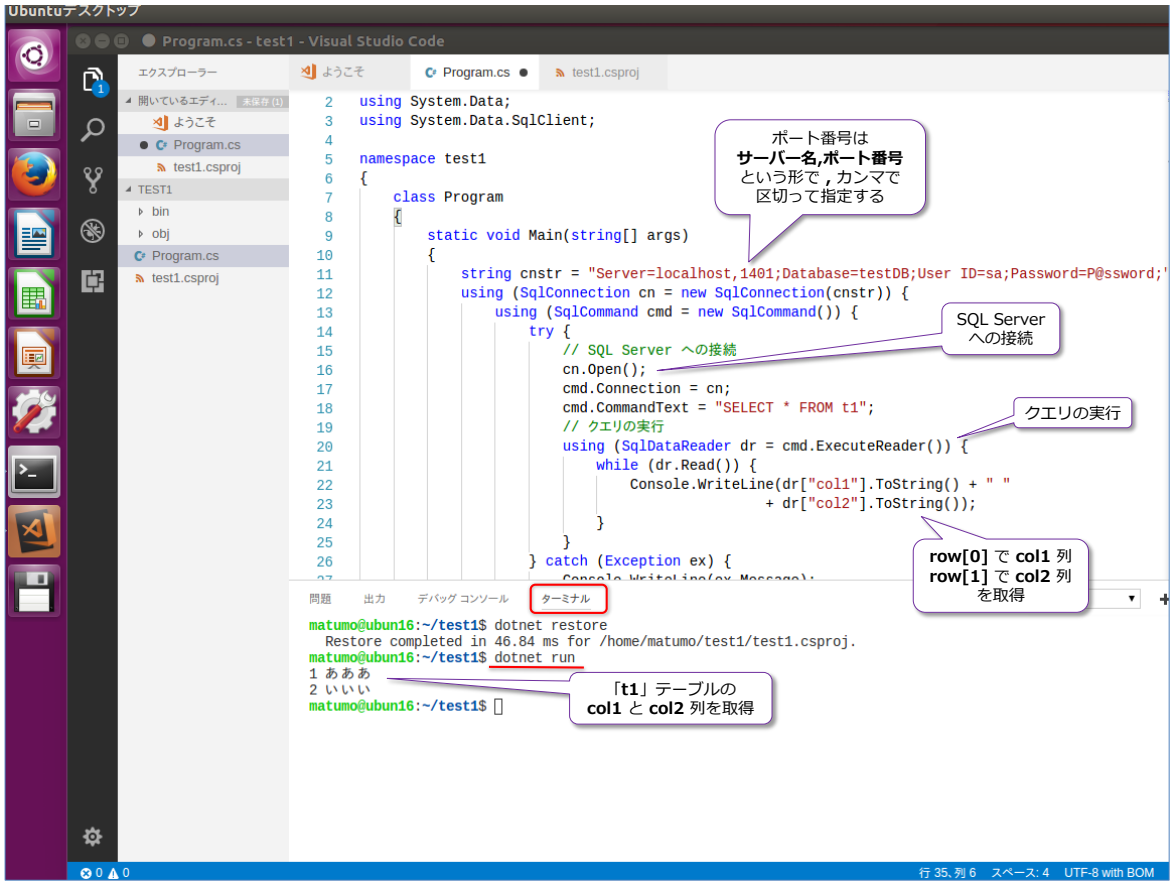


プロジェクト ファイルを修正した後は、ターミナルから「**dotnet restore**」コマンドを実行する必要があります。

```
dotnet restore
```

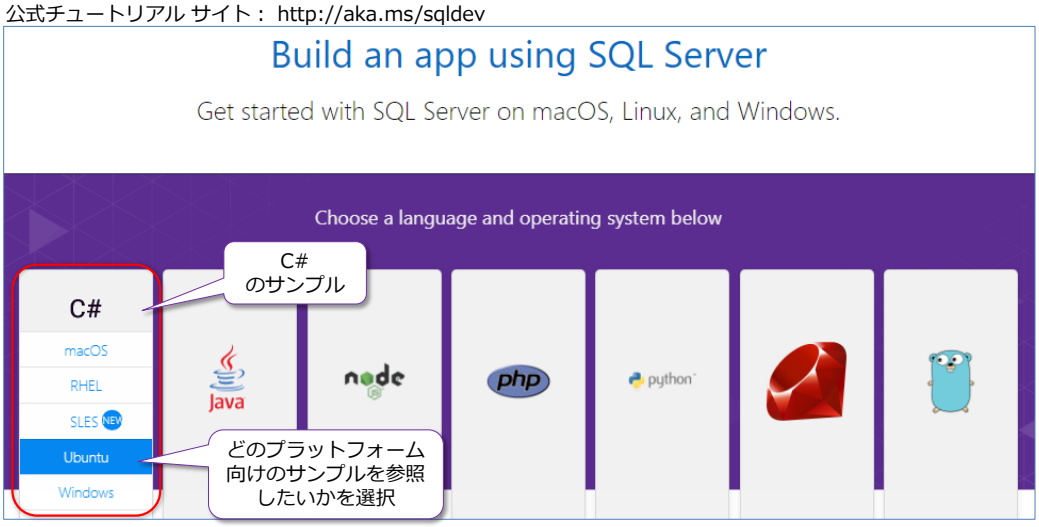


続いて、「**dotnet run**」を実行すれば、SQL Server にアクセスすることができます。



その他の .NET Core の利用方法（**Entity Frameowork Core**）については、前掲のチュートリアル サイト（**aka.ms/sqldev**）にサンプルがあるので、こちらが参考になると思います。

公式チュートリアル サイト： http://aka.ms/sqldev



また、.NET Core では、Windows 環境での .NET Framework との微妙な違いもあるので、それらについては、.NET Core のドキュメント（以下）が参考になります。

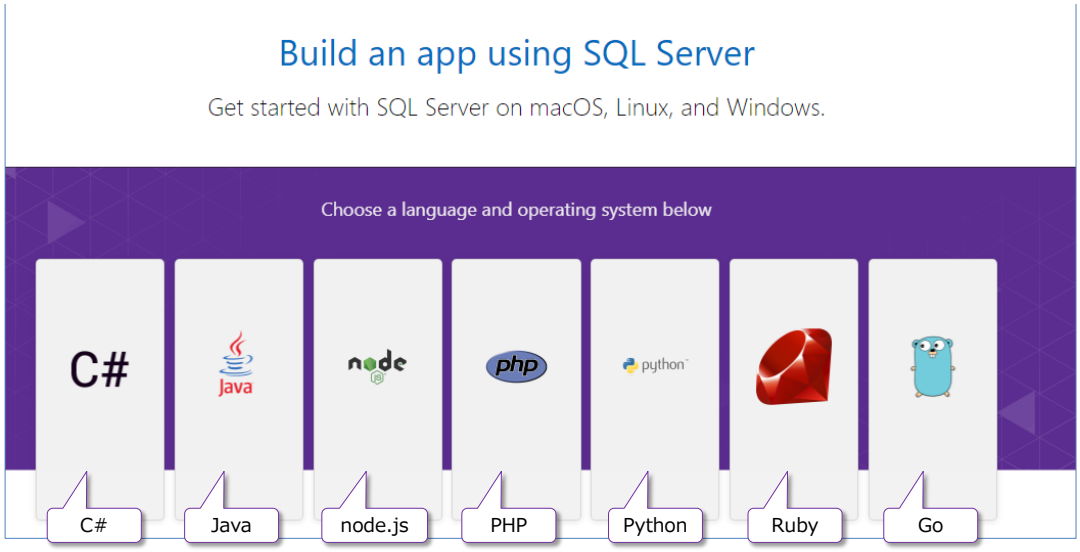
.NET Core の新機能
https://docs.microsoft.com/ja-jp/dotnet/core/whats-new/

ASP.NET Core の概要
https://docs.microsoft.com/ja-jp/aspnet/core/

dotnet new コマンド
https://docs.microsoft.com/ja-jp/dotnet/core/tools/dotnet-new?tabs=netcore2x

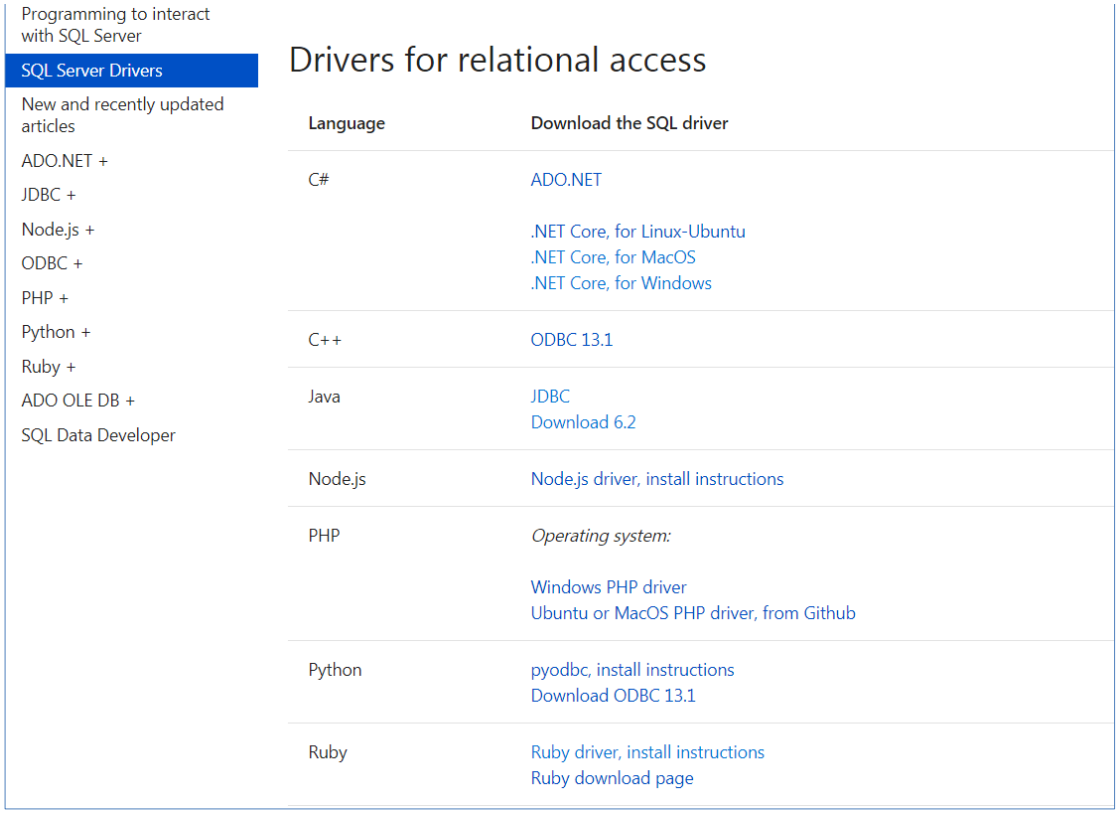
3.7 その他の言語を利用したアプリケーション開発の概要（Ruby、Go）

Ruby や **Go** など、その他の言語を利用したアプリケーション開発（SQL Server on Linux にアクセスするアプリケーションの開発）については、前掲のチュートリアル サイト（**aka.ms/sqldev**）にサンプルがあるので、こちらが参考になると思います。



また、SQL Server に関する接続モジュールに関しては、以下の URL にまとまっています。

Connection modules for Microsoft SQL databases
https://docs.microsoft.com/en-us/sql/connect/sql-connection-libraries



STEP 4. アプリケーション開発者のための SQL Server の基本操作

この STEP では、アプリケーション開発者が知っておくべき **SQL Server の基本操作**について説明します。「**ログイン／ユーザーの作成**」や「**オブジェクト権限の設定**」、「**列ストア インデックスによる性能向上**」、「**リンク サーバーによるリモート サーバーの操作**」、「**CSV ファイルのインポート／エクスポート**」、「**バックアップとリストア**」、「**SQL の定期実行**」などを説明します。

この STEP では、次のことを学習します。

- ✓ データベースの作成、テーブルの作成、データ型
- ✓ データの追加、更新、削除、照合順序（大文字と小文字の区別）
- ✓ ログイン／ユーザーの作成、オブジェクト権限の設定
- ✓ Transact-SQL の制御要素（変数、IF、WHILE など）
- ✓ 列ストア インデックス（カラム ストア）による性能向上
- ✓ リンク サーバーによるリモート サーバー データの取得
- ✓ CSV ファイルのインポート／エクスポート
- ✓ データベースのバックアップとリストア
- ✓ SQL Server Agent ジョブによる SQL の定期実行

4.1 アプリケーション開発者のための SQL Server の基本操作

この STEP では、**アプリケーション開発者**が知っておくべき **SQL Server の基本操作**を説明します。データベースの作成やテーブルの作成などは、既に紹介していますが、ここでは改めて詳しく説明します。

この STEP で説明する具体的な内容は、次のとおりです。

- データベースの作成、データベースへの接続
- テーブルの作成、データ型
- データの追加、更新、削除、照合順序（大文字と小文字の区別）
- 管理者アカウント（sa アカウント、sysadmin ロール）
- ログイン／ユーザーの作成、オブジェクト権限の設定、public、スキーマ、dbo
- Transact-SQL の制御要素（変数、IF、WHILE、Oracle PL/SQL との比較など）
- 列ストア インデックス（カラム ストア）による性能向上
- リンク サーバーによるリモート サーバー データの取得
- CSV ファイルのインポート／エクスポート
- データベースのバックアップとリストア
- SQL Server Agent ジョブ（SQL の定期実行）
- その他の考慮事項

SQL Server での SQL ステートメントを記述する上での基本

上記の内容を説明する前に、SQL Server における SQL ステートメントを記述する上での基本について説明します。

- **文末のセミコロン**はあってもなくても OK
  SQL ステートメントの文末には、; を付けても付けなくてもどちらでも大丈夫です。
- 1 行コメントは **//**、複数行コメントは **/* */** で囲む
- SQL Server では **FROM** を省略できる
  ex) **SELECT @@VERSION** のような実行ができる
- 特殊な名前は [ ] または " " で囲む（大カッコまたは二重引用符で囲む）
  ex) SELECT * FROM [**特殊な名前**] のように、オブジェクト名に特殊な名前を付けている場合は [ ] または " " で囲みます。このため、SQL Server のサンプル コードでは、特殊でない名前のものでも [ ] で囲んであるものが多数ありますが、これは念のために利用しているもので、特殊な名前でなければ外すことが可能です。

4.2 データベースの作成、テーブルの作成

まずは、データベースの作成や、作成したデータベースの確認、データベースへの接続、テーブルの作成、データの追加／更新／削除などを説明します。

データベースの作成（CREATE DATABASE）

SQL Server でデータベースを作成するには、**CREATE DATABASE** ステートメントを利用します。

```
-- データベースの作成（sampleDB という名前のデータベースの場合）
CREATE DATABASE sampleDB
```

CREATE DATBASE [**sampleDB**] のように、名前を [ ] で囲んで実行することもできます。

作成したデータベースを削除する場合は、**DROP DATABASE** ステートメントを実行します。

```
-- データベースの削除（sampleDB データベースを削除する場合）
DROP DATABASE sampleDB
```

データベースを作成するときに、作成場所（ディレクトリ）やファイル サイズを指定したい場合には、次のように **ON PRIMARY** や **LOG ON** を利用します。

```
-- データベースの作成場所やサイズを指定する場合。/db ディレクトリに 1GB で作成する例
CREATE DATABASE sampleDB
 ON PRIMARY (
     NAME = sampleDB,                    -- 論理名
     FILENAME = '/db/sampleDB.mdf',      -- 物理パス（データ ファイルは .mdf）
     SIZE = 1GB )                        -- ファイル サイズ
 LOG ON
  (  NAME = sampleDB_log,                -- 論理名
     FILENAME = '/db/sampleDB.ldf',      -- 物理パス（ログ ファイルは .ldf）
     SIZE = 1GB )                        -- ファイル サイズ
```

SQL Server では、データベースの実体は 2 種類のファイルで構成しますが、データやオブジェクトを格納するための**データ ファイル**と、データ変更の履歴を格納するための**トランザクション ログ ファイル**（Oracle でいう REDO ログ ファイルに相当）を利用します。前者は **.mdf**（Master Data File の略）、後者は **.ldf**（Log Data File の略）という拡張子を付けるのが慣習です。

それぞれのファイルは、**論理名**（Logical Name）と呼ばれる名前を設定（上のステートメントでの **NAME** で指定）して、データベースの実体を操作するための管理系の SQL ステートメント（バックアップやリストア、ファイル配置の変更、ファイル サイズの変更時など）で利用します。

上のステートメントの **FILENAME** では、ファイルの作成場所を物理パスで指定（**/db** ディレクトリを指定）、**SIZE** にはファイル サイズとして **1GB** を指定しています。SQL Server では、デ

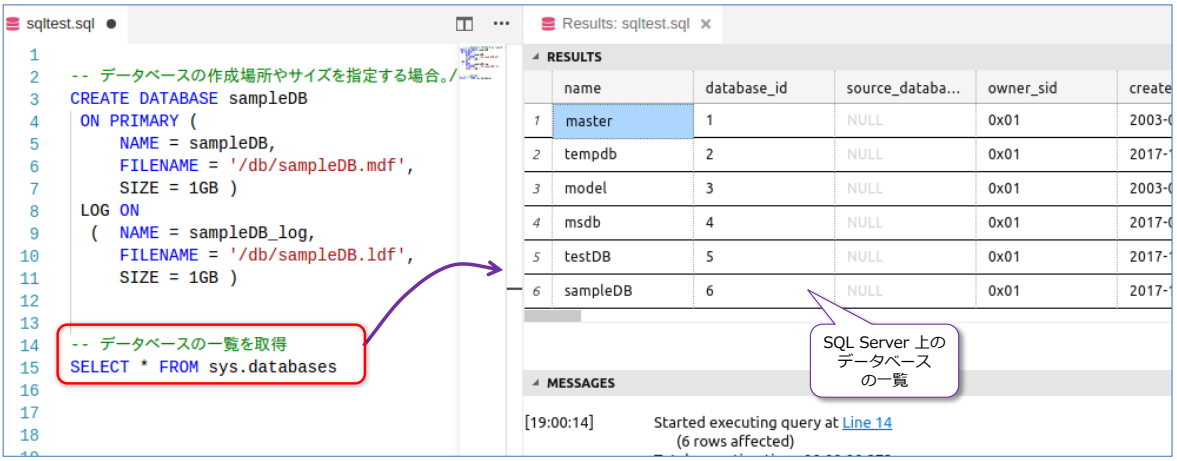
ータベース サイズ（データ ファイルやトランザクション ログ ファイル）が足りなくなったときに、自動的に拡張する機能がありますが、自動拡張が発生した場合には、その間、トランザクションを受け付けられなくなって性能低下を引き起こすことになるので、**SIZE** には、大きめのサイズを設定して、自動拡張が発生しないように運用していくのがベスト プラクティスになります。

上のステートメントの **ON PRIMARY** は、データ ファイル（.mdf）をプライマリのファイル グループに作成するという意味になりますが、SQL Server では、ファイル グループという単位でデータベースに対する管理操作が可能です（Oracle でいう表領域に近い考え方がファイル グループで、データベースに対して複数のファイル グループを作成して、テーブルやデータ パーティションが格納されるファイル グループを分散する、といった利用方法ができます）。

作成したデータベースの確認（sys.databases）

SQL Server では、作成したデータベースの一覧は、**sys.databases** システム ビューを参照することで確認できます。

```
-- データベースの一覧を取得
SELECT * FROM sys.databases
```



master や **tempdb**、**model**、**msdb** の 4 つは、SQL Server が内部的に利用しているシステム データベースで、**master** には構成オプションなど SQL Server が内部的に利用する環境設定やシステム オブジェクトなどが格納され、**tempdb** は一時的なデータを格納するための領域、**model** は新しくデータベースを作成するときのモデルになるデータベース、**msdb** は後述の SQL Server Agent ジョブなどを格納するためのデータベースです。なお、master と model、msdb の 3 つは、Oracle でいうところの システム表領域＋初期化パラメータ＋制御ファイル、tempdb は一時表領域に相当する場所です。

これらのシステム データベースには、**ユーザー オブジェクト**（ユーザーが作成するテーブルやビュー、ストアド プロシージャなど）は格納しないようにして、**CREATE DATABASE** でユーザーが作成したデータベース内に、ユーザー オブジェクトを格納していくようにします。

データベースへの接続（USE）

SQL Server では、データベースへの接続に **USE** を利用します。

```
-- データベースへの接続。SQL Server では USE でデータベースに接続する
USE sampleDB
```

もし、**USE** を利用せずに、操作を行った場合は、**sa** アカウントの場合は **master** データベースに接続された状態（**USE master** の状態）になります。**CREATE DATBASE** は、master に接続した状態で実行しますが、ユーザー オブジェクトの作成（**CREATE TABLE** でのテーブル作成など）を行う場合は、ユーザー データベースに **USE** した状態で実行するようにします。

テーブルの作成（CREATE TABLE）

テーブルの作成は、**CREATE TABLE** ステートメントを利用します。

```
-- テーブルの作成。「t1」という名前で作成する例
CREATE TABLE t1
( col1 int
 ,col2 varchar(100) )
```

この例では、「**t1**」という名前のテーブルに、「**col1**」列を「**int**」データ型、「**col2**」列を「**varchar(100)**」データ型で作成しています。**int** は整数データを格納できるデータ型、**varchar** は文字列データを格納できるデータ型です（前者は Oracle でいう NUMBER、後者は VARCHAR に相当します）。

SQL Server で利用できる主なデータ型には、次のものがあります。

	SQL Server	対応する Oracle のデータ型	説明
文字列	char(n) varchar(n)	CHAR(n) VARCHAR(n)	固定長／可変長文字列（8,000 バイトまで）
	nchar(n) nvarchar(n)	NCHAR(n) NVARCHAR(n)	Unicode 対応の文字列
	varchar(max) nvarchar(max)	CLOB, LONG NCLOB	8,000 バイト越えの可変長文字列を格納
バイナリ	binary(n) varbinary(n) varbinary(max)	RAW(n), LONG RAW BLOB	バイナリ データの格納
整数	tinyint, smallint, int, bigint	NUMBER(p)	整数を格納
小数点	decimal(p, s) numeric(p., s) など	NUMBER(p, s)	p: 全体桁数, s: 小数点以下の桁数
日付時刻	date smalldatetime datetime など	DATE	日付／時刻データの格納

SQL Server では、固定長の文字列に **char**、可変長の文字列に **varchar**、Unicode データを格納したい場合に **nchar** または **nvarchar**、整数は **int** など、小数点以下の桁数がある場合は **decimal**、日付データを格納する場合は **date**（日単位）、時刻を含めた日付を格納したい場合には **datetime**、バイナリを格納したい場合に **varbinary**、大きいデータを格納したい場合に **(max)**

を付ける、というのが基本的なデータ型の利用方法になります。

```
-- テーブルの作成例２
CREATE TABLE test1
( col1 char(200)          -- 固定長文字列の格納。200バイトまで格納可能
 ,col2 nvarchar(200)      -- Unicode 対応の可変長文字列。100文字まで格納可能
 ,col3 date               -- 日付データを格納. 2017/10/12 など日単位（時刻は切捨て）
 ,col4 varbinary(max)     -- バイナリの大きいデータ（8,000バイト以上のもの）を格納
 ,col5 numeric(10, 3)     -- 全体の桁数 10、小数点以下の桁数 3 のデータを格納
)
```

テーブルの削除（DROP TABLE）

テーブルの削除は、**DROP TABLE** ステートメントを利用します。

```
-- テーブルの削除
DROP TABLE test1
```

なお、存在しないテーブルに対して **DROP TABLE** を実行しようとすると実行エラーになりますが、次のように「**IF EXISTS**」を付けて実行することで、存在しなかったとしても、エラーにならないようになります（存在する場合にのみ削除を実行）。

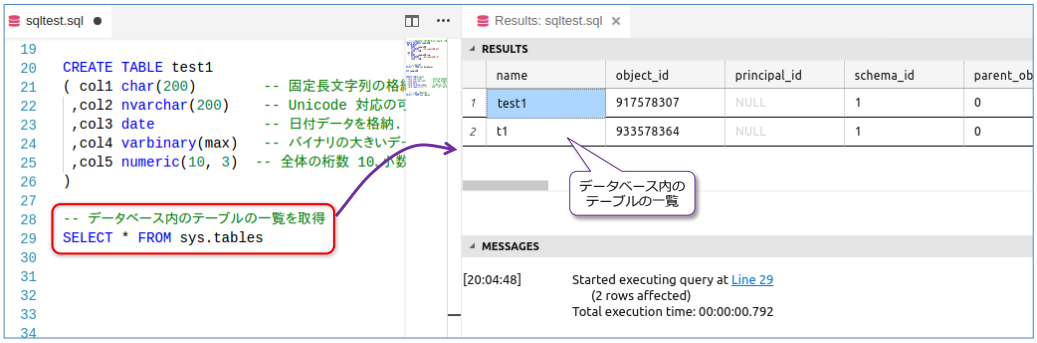
```
-- 該当テーブルが存在するなら削除
DROP TABLE IF EXISTS test1
```

この **IF EXISTS** は、他のオブジェクト（後述のビューやストアド プロシージャなど）を操作する場合にも共通で利用できるものになっています。

作成したテーブルの一覧を取得

作成したテーブルの一覧（データベース内のテーブルの一覧）を取得したい場合には、次のように **sys.tables** システム ビューを参照します。

```
-- データベース内のテーブルの一覧を取得
SELECT * FROM sys.tables
```

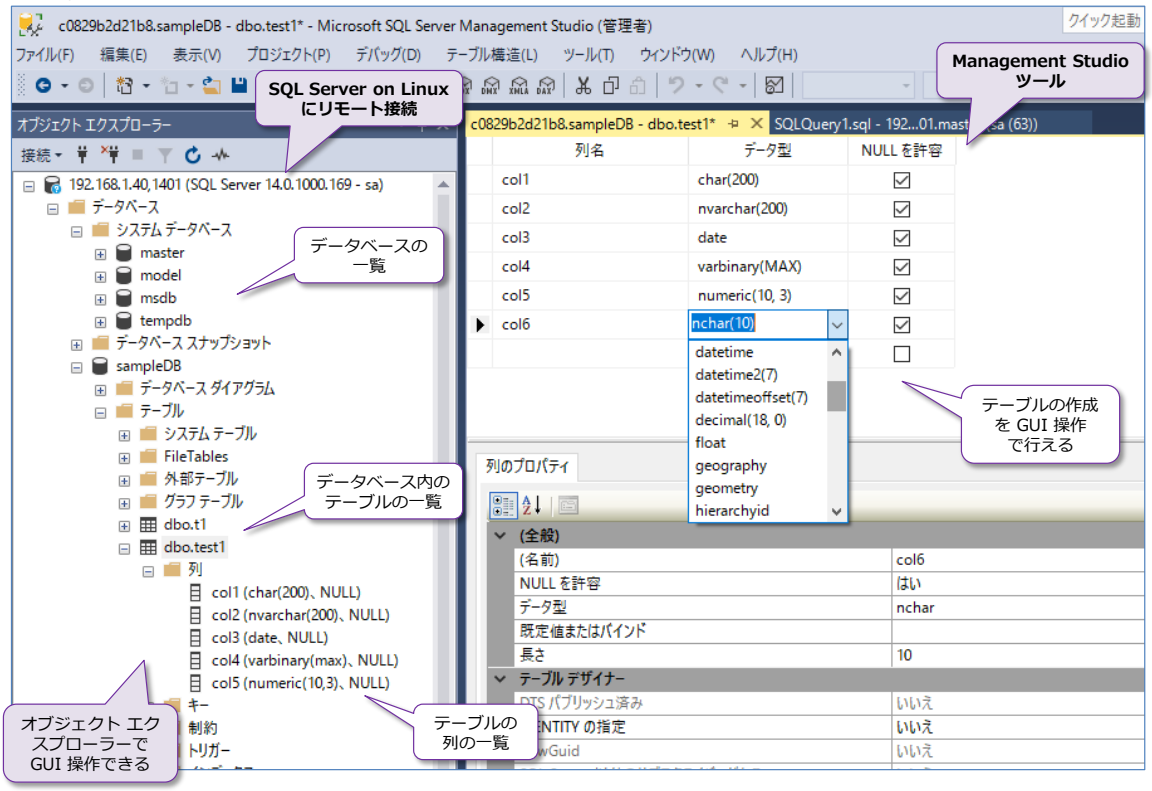


テーブル内の列定義（列名やデータ型）を取得したい場合は、**sp_help** を利用します。

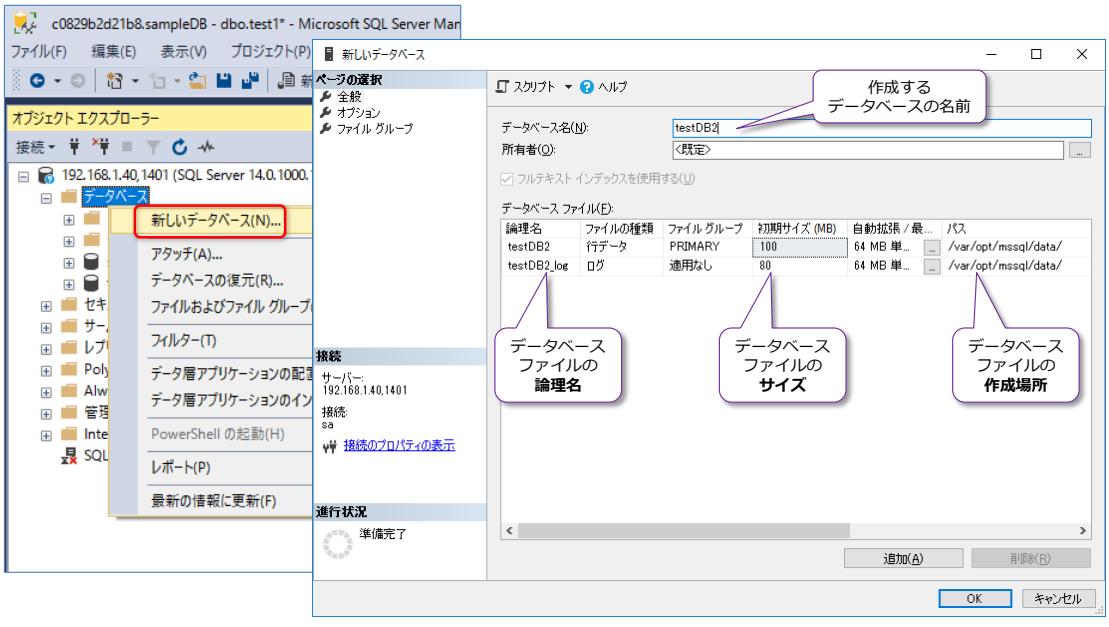
```
-- t1 テーブルの列定義を取得
EXEC sp_help 't1'
```

なお、Windows 環境に Management Studio ツールをインストールして、SQL Server on Linux をリモート操作できる場合には、Management Studio のオブジェクト エクスプローラーを利用して、GUI 操作でデータベースを作成したり、テーブルを作成したりすることができます。

Management Studio から SQL Server on Linux にリモート接続すれば GUI で操作できる



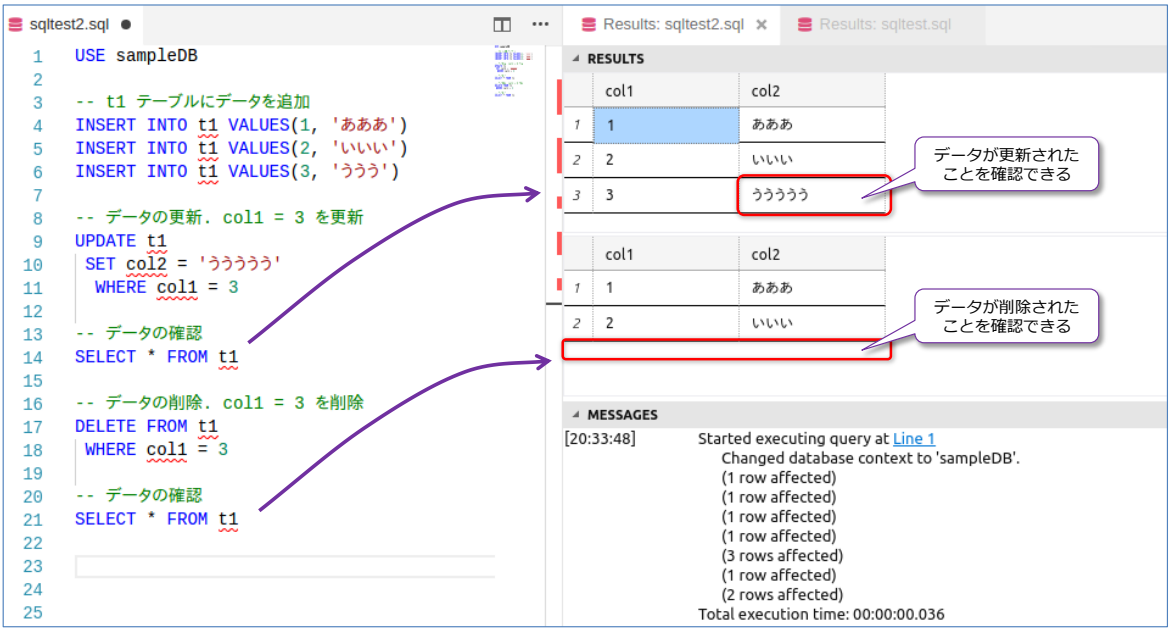
データベースの作成も GUI 操作で可能



4.3 データの追加／更新／削除

SQL Server におけるデータの追加（**INSERT**）や、更新（**UPDATE**）、削除（**DELETE**）は、他のデータベース サーバーと同様に操作できます。

```
-- データベースに接続
USE sampleDB

-- t1 テーブルにデータを追加
INSERT INTO t1 VALUES(1, 'ああああ')
INSERT INTO t1 VALUES(2, 'いいいい')
INSERT INTO t1 VALUES(3, 'ううううう')

-- データの更新. col1 = 3 を更新
UPDATE t1
 SET col2 = 'うううううう'
  WHERE col1 = 3

-- データの確認
SELECT * FROM t1

-- データの削除. col1 = 3 を削除
DELETE FROM t1
 WHERE col1 = 3

-- データの確認
SELECT * FROM t1
```



SQL Server では、データの追加／更新／削除があった場合に、自動的に履歴テーブルに変更履歴を格納することができる「**テンポラル テーブル**」という機能があります。これを利用すれば、オペレーション ミスなどがあった場合に、簡単にデータを復旧できるようになります。

なお、テーブルに対する不正使用（アクセスされたくない機密情報が格納されたテーブルにアクセ

スさたり、不正にデータが削除されたりすること）に対応するには、後述のセキュリティのところで説明する「**SQL Server Audit**」という監査機能があります。この Audit 機能を利用すれば、SQL Server に対するあらゆる操作を記録・監視することができる（監査証跡：Audit Trail を残せる）ので、昨今の**コンプライアンス対策**（法令遵守）として最適な機能です。

また、機密情報をマスクして返す（権限のないユーザーには本当の値を見せない）機能として、**動的データ マスク**（後述）があったり、行単位でユーザー制御（行レベルでユーザーが参照できるデータを資源できる機能）もあり、SQL Server は**セキュリティ向上**のための機能が非常に充実しています（次章で詳しく説明します）。

大文字と小文字の区別は？ ～照合順序：COLLATION～

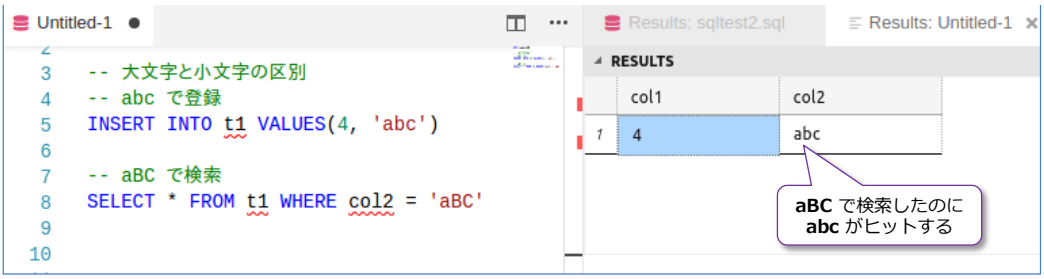
SQL Server における大文字と小文字の区別は、**照合順序**（**Collation**）によって動作が異なりますが、日本語版の SQL Server の既定の照合順序である「**Japanese_CI_AS**」では、次のような動作になります。

既定の照合順序「Japanese_CI_AS」の場合の大文字と小文字の区別

	SQL Server	Oracle
ステートメント	区別しない	区別しない
オブジェクト名	区別しない	区別しない
データ	区別しない	**区別する**

Japanese_CI_AS の **CI** は **Case Insensitive** の略で、**Case**（大文字／小文字）を **Insensitive**（区別しない）、**AS** は **Accent Sensitive** の略で、アクセントを区別するという意味になります。したがって、SQL Server の場合、既定ではデータの大文字と小文字を区別しません。

これは、次のような状況です。



abc という形で小文字で格納したデータに対して、**aBC** で検索すると、**abc** がヒットするという形です。**aBC** だけでなく、**ABC** や **AbC**、**Abc** で検索しても **abc** はヒットします。Oracle の場合は、データの大文字と小文字を区別するので、**abc** というデータの場合は、**aBC** や **ABC** ではヒットしません。

SQL Server を、Oracle と同じようにデータの大文字と小文字を区別するようにするには、照合順序に「**Japanese_CS_AS**」を利用するようにします。照合順序は、SQL Server 全体またはデータベース単位またはテーブルの列に対して設定することができ、照合順序を省略した場合には、

SQL Server の設定が継承されます。また、列単位で照合順序を設定すれば、それが一番強くなります。

なお、現在の照合順序を確認するには、次のように実行します。

```
-- SQL Server レベルの照合順序の確認
SELECT SERVERPROPERTY('collation')

-- データベース レベルの照合順序の確認
SELECT collation_name, * FROM sys.databases

-- 列レベルの照合順序の確認（t1 テーブルの場合）
EXEC sp_help 't1'
```

列単位の照合順序を設定する場合は、**CREATE TABLE** でのテーブルの作成時に、次のように **COLLATE** 句を利用します。

```
-- 列レベルで照合順序を変更する場合
CREATE TABLE test2
( col1 char(200)                                -- 照合順序を省略すると Japanese_CI_AS
 ,col2 varchar(200) COLLATE Japanese_CS_AS  -- 照合順序を変更
)
```

このように、Japanese_CS_AS を設定すれば、大文字と小文字を区別できるようになります。

データベース レベルで照合順序を変更したい場合は、データベースの作成時に、次のように **COLLATE** 句を利用します。

```
-- データベース レベルで照合順序を変更する場合
CREATE DATABASE sampleDB2 COLLATE Japanese_CS_AS
```

ただし、このようにデータベース レベルで照合順序を変更した場合は、作成するオブジェクトの名前にもこの照合順序が影響します。例えば、**Japanese_CS_AS** では、大文字と小文字を区別するので、**t1** という名前のテーブルを作成したとすると、「**SELECT * FROM T1**」のように大文字で利用しようとすると、該当オブジェクトが存在しない、と判断されてエラーになってしまいます。

照合順序には、**Japanese_CS_AS_KS** や **Japanese_BIN2** など、他にも種類があります。利用できる照合順序の一覧は、次のように取得できます。

```
-- 日本語の照合順序の一覧を取得
SELECT Name from sys.fn_helpcollations()
 WHERE name LIKE 'Japanese%'
```

Japanese_CS_AS_KS の **KS** の **K** は **Kana** の略で、**ひらがな**と**カタカナ**を区別するかどうか、**Japanese_BIN2** の **BIN** はバイナリの意味で、文字コード レベルですべての文字を区別する、などの違いがあったりします。SQL Server で利用できる主な照合順序は、次のような違いがあり

ます。

SQL Server の主な照合順序の動作の違い

	大文字と小文字（Case）	アクセント（Accent）	カナと平仮名（Kana）	半角と全角（Wide）
Japanese_CI_AS	**区別しない**	区別する	**区別しない**	**区別しない**
Japanese_**CS**_AS	**区別する**	区別する	**区別しない**	**区別しない**
Japanese_**CS**_AS_**KS**	**区別する**	区別する	**区別する**	**区別しない**
Japanese_**CS**_AS_**KS**_**WS**	**区別する**	区別する	**区別する**	**区別する**
Japanese_**BIN2**	**区別する**	区別する	**区別する**	**区別する**

Japanese_CS_AS_KS_WS の **WS** の **W** は **Wide** の略で、**全角**と**半角**を区別するかどうかです。したがって、既定値の **Japanese_CI_AS** では、**CI**（大文字・小文字を区別しない）、**KI**（カナと平仮名を区別しない）、**WI**（半角と全角を区別しない）という動作をします。これは、商品検索などを考えると、非常に便利な動作で、例えば「**Guru's ガイド**」という本があった場合に、**Japanese_CI_AS** であれば「**GURU'S ガイド**」と入力しても検索にヒットするし、「**GURU'S がいど**」（ガイドのところが平仮名）でも検索にヒットする、「**GURU'S　ガイド**」（ＲＵ が全角、スペースが全角）でも検索にヒットする、といった柔軟性を得られます（商品を検索したいユーザーは、どういった入力の仕方をするのかは人によってさまざまなため、ルーズな検索ができたほうが、ユーザーにとって使いやすいアプリケーションになります）。

もちろん、厳密にデータを識別したい場合もあると思います。その場合は、列レベルで照合順序を変更することができるので、厳密に識別したいデータ列にのみ、**Japanese_BIN2** を設定して、バイナリで区別するようにしたり、大文字と小文字だけを区別したいなら **Japanese_CS_AS** を利用する、といった利用方法ができます。

Unicode データの扱い

SQL Server では、**Unicode** データをテーブルに格納したい場合には、データ型に **nchar** または **nvarchar**（**n** 付きのデータ型）を利用するようにします。

また、Unicode データは、**N'～'** という形で単一引用符の前に **N** を付ける（**N** プレフィックスを付ける）必要があります。

```
-- Unicode データを格納する場合
CREATE TABLE test3
( col1 nvarchar(200)  -- Unicode を格納するには n 付きのデータ型
)

-- Unicode データは N で囲む
INSERT INTO test3 VALUES(N'Unicodeデータ')
```

4.4 ユーザーの作成、オブジェクト権限の設定

ここでは、SQL Server におけるユーザー作成やオブジェクト権限の設定方法などを説明します。これらはセキュリティ上、最も基本となる設定になります。

sa（管理者アカウント）、sysadmin ロール（管理者グループ）

これまでの手順では、**sa** というアカウントを利用してきましたが、これは SQL Server on Linux をインストールすると利用できる、SQL Server の管理者アカウントです。

もし、**sa** のパスワードを変更したい場合には、次のように **ALTER LOGIN** ステートメントを実行します（8 文字以上の複雑なパスワード＝大文字、小文字、数字、記号の 4 種類のうち、いずれかの 3 種類の文字を含めるものを指定する必要があります）。

```
-- sa のパスワードを変更したい場合
USE master
ALTER LOGIN sa
 WITH PASSWORD = '新しいパスワード'
```

SQL Server では、**sysadmin** サーバー ロールという管理者として振る舞える権利をもったロール（管理者グループのようなもの）があります。**sa** は、このロールを与えられていることで、管理者として SQL Server に対して振る舞えています。

後述のログイン アカウントを作成して、そのアカウントに sysadmin ロールを割り当てたい場合（sa と同様の権限を持った管理者を別途作成したい場合）は、次のように **ALTER SERVER ROLE** ステートメントを実行します。

```
-- sysadmin ロールの割当て（管理者グループへの追加）
ALTER SERVER ROLE sysadmin
 ADD MEMBER ログインアカウント名
```

現在のログイン アカウントの一覧や、サーバー ロールのメンバーを確認したい場合には、次のようにシステム ビューを参照します。

```
-- ログイン アカウントとサーバー ロールの一覧
SELECT * FROM sys.server_principals

-- サーバー ロール内のメンバーの一覧
SELECT * FROM sys.server_role_members
```

ログイン アカウント、データベース ユーザーの作成

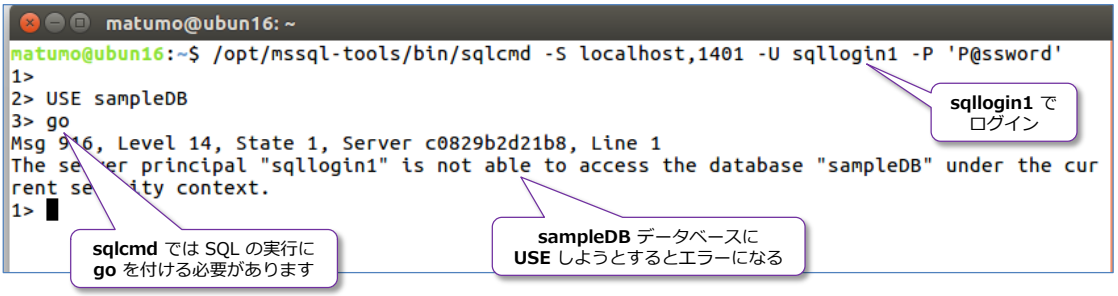
SQL Server では、**データベース ユーザー**（オブジェクト権限を設定するためのユーザー）を作成する前に、**ログイン アカウント**を作成するのが基本になります。ログイン アカウントは、SQL

Server の認証のために必要なアカウントで、パスワードを設定して、内部的には SQL Server の master データベース内に保存します。ログイン アカウントは、次のように **CREATE LOGIN** ステートメントで作成します。

```
-- ログイン アカウントの作成（sqllogin1 という名前でパスワードを P@ssword に設定）
USE master
CREATE LOGIN sqllogin1
 WITH PASSWORD = 'P@ssword'
```

このように、作成したログイン アカウントは、SQL Server にログインできるようになります。例えば、**sqlcmd** ツールを利用する場合は、次のようにログインできます。

```
-- sqlcmd ツールでログイン
/opt/mssql-tools/bin/sqlcmd -S localhost,1401 -U sqllogin1 -P 'P@ssword'
```



ログインを作成しているので、SQL Server にログインすることはできていますが、**USE** で **sampleDB** データベースに接続しようとすると、エラーが返ってきます。ログイン アカウントを作成しただけでは、データベースに接続することはできません。データベースに接続できるようにするためには、ログイン アカウントに対応したデータベース ユーザーを作成する必要があります。

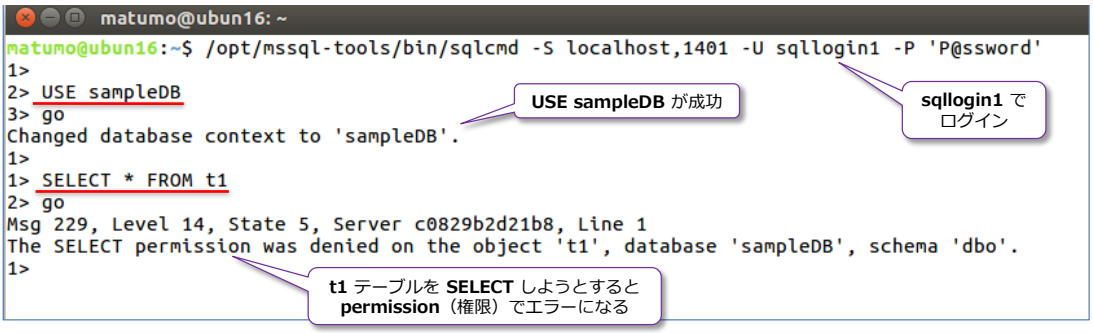
データベース ユーザーは、次のように **CREATE USER** ステートメントで作成できます（接続させたいデータベースごとにユーザーを作成する必要があります）。

```
-- sampleDB でデータベース ユーザーを作成（sqllogin1 という名前で作成）
USE sampleDB
CREATE USER sqllogin1 FOR LOGIN sqllogin1
```

CREATE USER に続けてデータベース ユーザーの名前、**FOR LOGIN** に続けてログイン アカウントの名前を指定します（データベース ユーザーとログイン アカウントを異なる名前に設定することもできますが、管理上は同じ名前のほうが管理しやすいので、同じ名前に設定しておくことをお勧めします）。

データベース ユーザーの作成が完了したら、次に、もう一度 sqlcmd ツールでログインを行って、**USE sampleDB** を実行してみます。

```
-- sqlcmd ツールでログイン
/opt/mssql-tools/bin/sqlcmd -S localhost,1401 -U sqllogin1 -P 'P@ssword'
```



データベース ユーザーを作成したことで、**USE sampleDB** は成功していますが、データベース内の **t1** テーブルを **SELECT** しようとすると "**SELECT 権限**" がない主旨のエラーが返って、中身を参照することができません。データベース ユーザーを作成しただけでは、テーブルを SELECT したり、更新（INSERT／UPDATE／DELETE）したりすることはできません。ユーザーがテーブルを操作できるようにするには、オブジェクト権限を設定する必要があります。

オブジェクト権限の設定（GRANT／REVOKE／DENY）

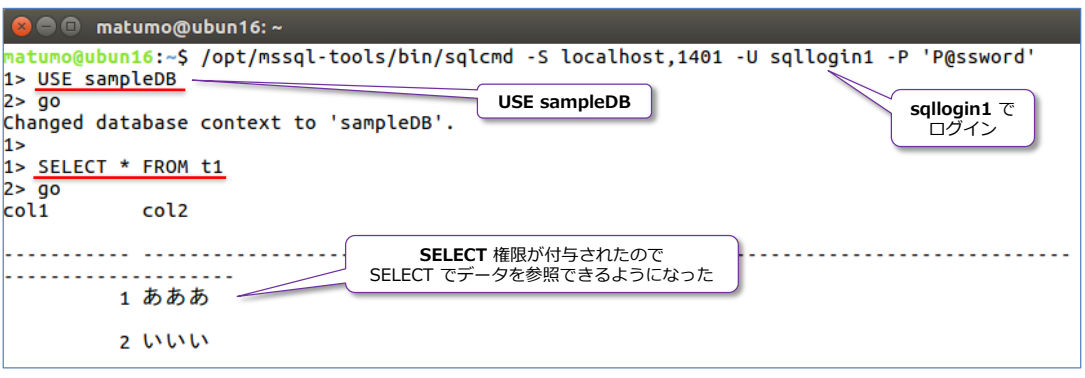
オブジェクト権限は、テーブルや後述のビュー、ストアド プロシージャなどのオブジェクトに対する権限のことで、テーブルの場合は、SELECT するための SELECT 権限、INSERT するための INSERT 権限、ストアド プロシージャであれば、実行するための EXECUTE 権限などがあります。オブジェクト権限を付与（GRANT）するには、次のように **GRANT** ステートメントを実行します。

```
-- t1 テーブルの SELECT 権限を sqllogin1 ユーザーに付与
GRANT SELECT ON t1 TO sqllogin1
```

GRANT は許可という意味ですが、これに続けて権限の名前（**SELECT** 権限なら **SELECT** と指定）、**ON** の後にオブジェクト名（**t1** など）、**TO** の後にデータベース ユーザー名（**sqllogin1** など）を指定します。

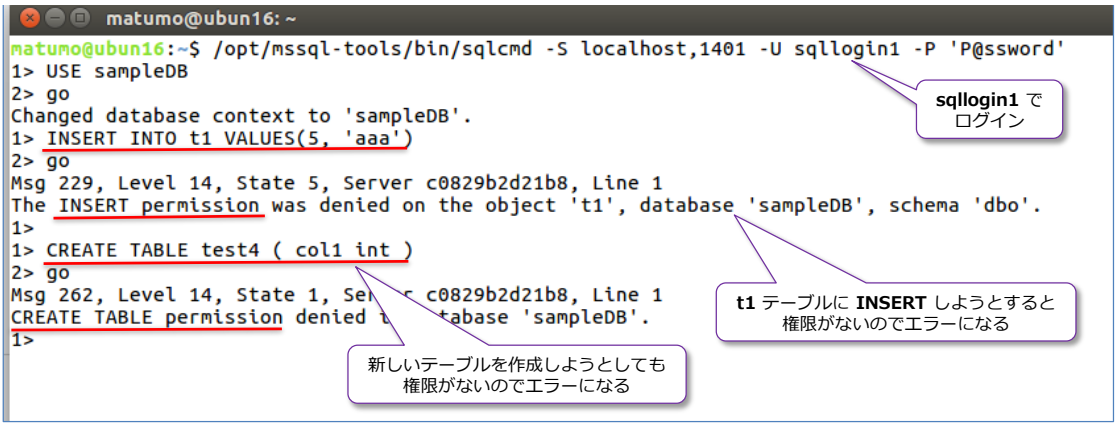
このように、**t1** テーブルに対する **SELECT** 権限が付与されていれば、**sqllogin1** データベース ユーザーは、SELECT ステートメントでデータを参照できるようになります。これを確認するために、もう一度 sqlcmd ツールでログインを行ってみましょう。

```
-- sqlcmd ツールでログイン
/opt/mssql-tools/bin/sqlcmd -S localhost,1401 -U sqllogin1 -P 'P@ssword'
```



以上のように、SQL Server では、**管理者アカウント**（**sa** アカウントや **sysadmin** ロールが割り当てられたログイン アカウント）の場合は、SQL Server に対してどんな操作でも可能（データベースの作成からテーブルの作成、データの参照などあらゆる操作が可能）ですが、**CREATE LOGIN** ステートメントで作成したログイン アカウントは、SQL Server に接続できるだけです。

また、**CREATE USER** ステートメントで、ログイン アカウントに対応した**データベース ユーザー**を作成したとしても、テーブルに対してオブジェクト権限が付与されていなければ、参照することさえもできません。前の手順で、**sqllogin1** データベース ユーザーは **t1** テーブルの **SELECT** はできるようになりましたが、次のように **INSERT** を実行したり、**CREATE TABLE** で他のテーブルを作成したりすることはできません。



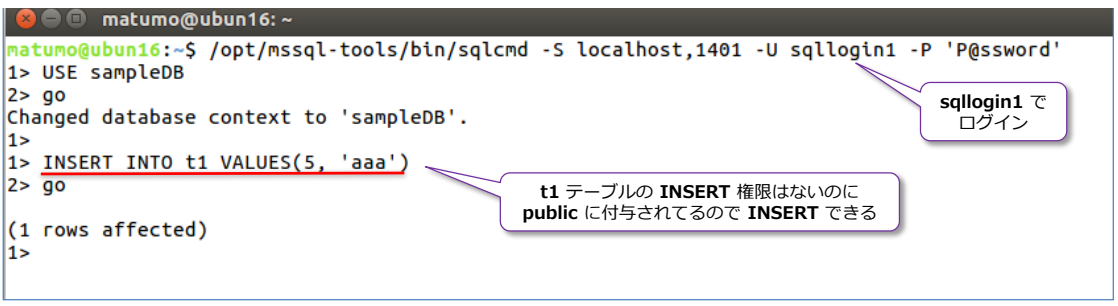
public（すべてのユーザー）

SQL Server では、**public** という特別な名前のデータベース ユーザー（正確にはロール）があります。これは、すべてのユーザーが当てはまり（Windows でいう everyone に相当）、**public** に対して権限が付与されているものに対しては、すべてのデータベース ユーザーがその権限を付与されているのと同じにすることができます。

例えば、次のように **public** に対して **INSERT** 権限を付与したとします。

```
-- t1 テーブルの INSERT 権限を public に付与
GRANT INSERT ON t1 TO public
```

このように、設定すると、**sqllogin1** データベース ユーザーでも **INSERT** を実行できるようになります。



このように、**public** を利用する場合は、意図しない抜け穴（管理者の設定ミスによるセキュリティ ホール）ができる可能性があるので、できる限り **public** は利用しないようにしましょう。

REVOKE（取り消し）

オブジェクト権限では、**GRANT**（許可）のほかに、GRANT したものを取り消す **REVOKE**（取り消し）がありますが、これは次のように利用します。

```
-- REVOKE で取り消し（t1 テーブルの SELECT 権限を取り消す）
REVOKE SELECT ON t1 FROM sqllogin1
```

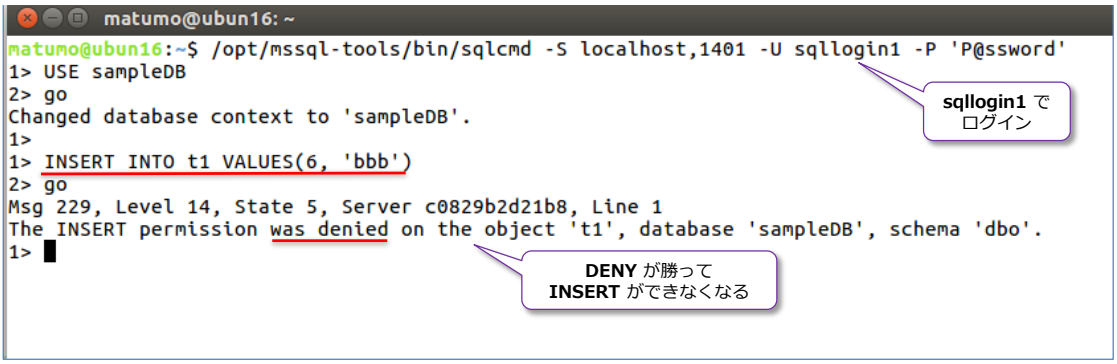
REVOKE では、**GRANT** の **TO** のところが **FROM** に変わっただけで、他は同じ構文です。

DENY（拒否）

オブジェクト権限には、**DENY**（拒否）というものもあります。例えば、次のように利用したとします。

```
-- t1 テーブルの INSERT 権限を sqllogin1 に対して拒否
DENY INSERT ON t1 TO sqllogin1
```

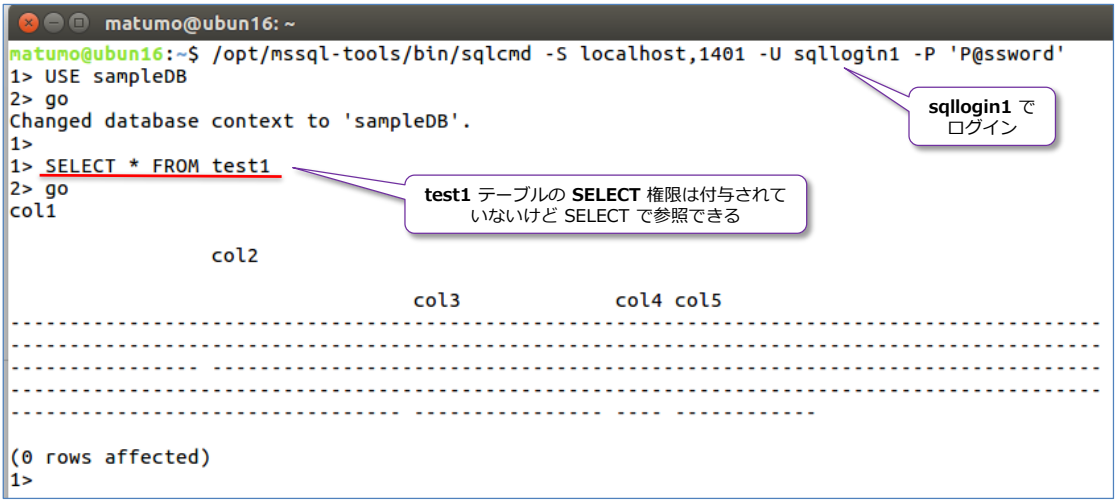
DENY は、**GRANT** を同じ構文です。これは、**sqllogin1** データベース ユーザーから **INSERT** 権限を **DENY**（拒否）していますが、先ほど **public**（すべてのユーザー）に対して、**INSERT** 権限を **GRANT**（許可）していたので、INSERT が実行できることを確認しました。しかし、**DENY** が与えられている場合は、**DENY** が勝ちます。これは次のような状況です。



db_datareader データベース ロール

SQL Server では、データベース内のすべてのオブジェクトに対して、読み取り権限（**SELECT** 権限）のみを付与できる **db_datareader** というデータベース ロール（権利）があります。このロールを付与するには、次のように **ALTER ROLE** ステートメントを利用します。

```
-- db_datareader ロールを sqllogin1 に付与
ALTER ROLE db_datareader
 ADD MEMBER sqllogin1
```



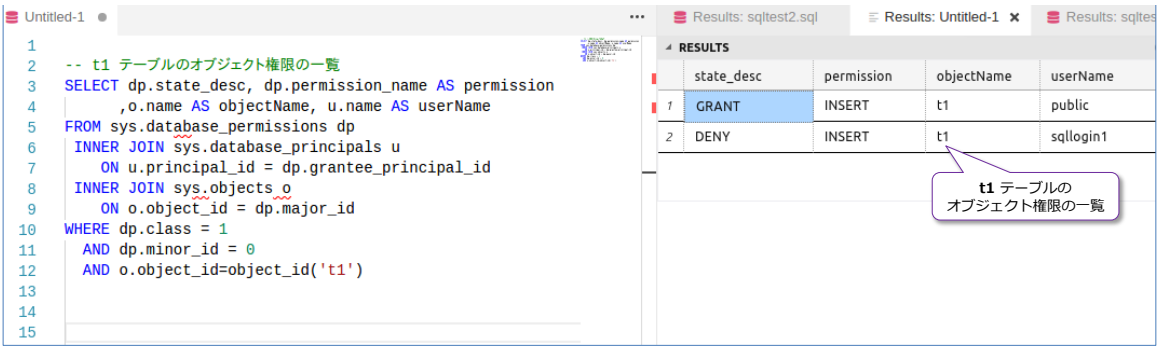
このように、**db_datareader** ロールを付与すれば、SELECT 権限のないテーブルでも参照できるようになるので、開発環境などで、とりいそぎテーブルの参照をさせたい場合などで便利です（本番環境での利用はお勧めしません）。なお、**db_datareader** ロールが付与されていたとしても、「**DENY SELECT ON test1 TO sqllogin1**」のように **DENY** で **SELECT** が拒否されている場合は **DENY** が勝ちます（該当テーブルの参照はできません）。

以上が SQL Server におけるデータベース ユーザーとオブジェクト権限の基本的な利用方法になります。セキュリティを高めるためには、次の章で説明する**動的データ マスク**を利用して、データをマスクしたり（一般ユーザーには、機密情報を別の値に変換して返す）、**行レベル セキュリティ**を設定して、行レベル単位で参照できるデータを制限したりすることができます。また、**バックアップ ファイル**や**データベース全体を暗号化**して、物理的な盗難対策をしたり、**Always Encrypted** を利用して列データを暗号化する、といったこともできます。

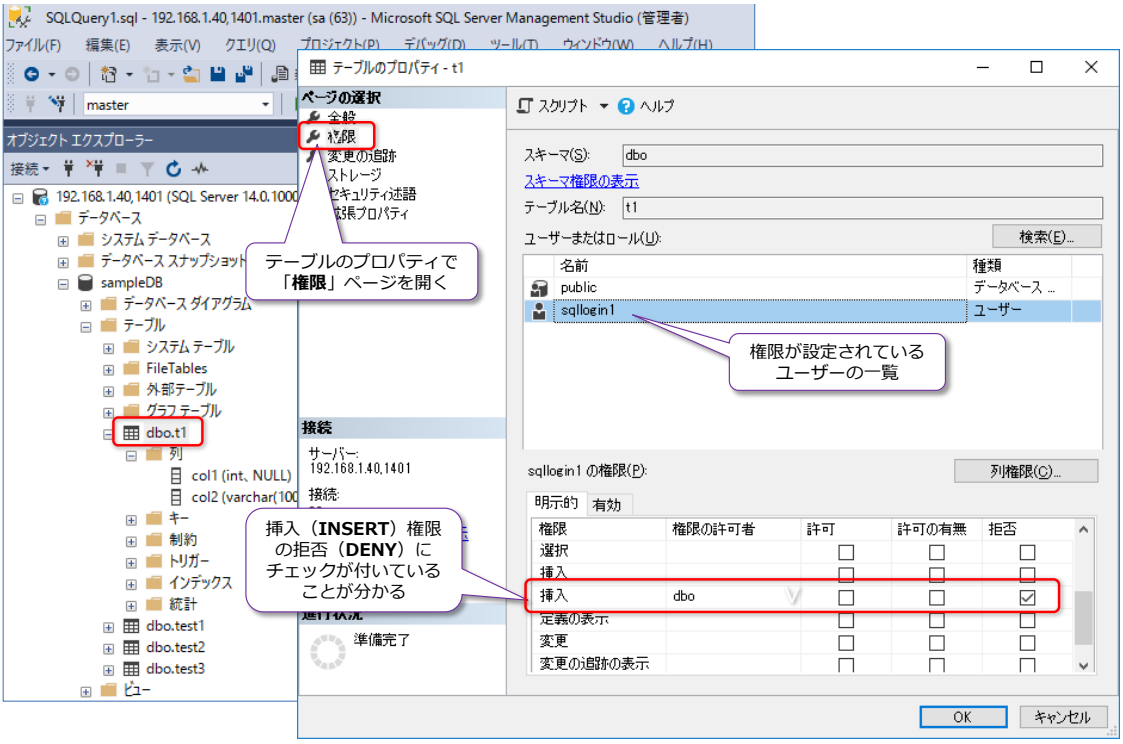
オブジェクト権限の一覧を取得

GRANT や DENY などで設定したオブジェクト権限は、次のように一覧できます。

```
-- t1 テーブルのオブジェクト権限の一覧
SELECT dp.state_desc, dp.permission_name AS permission
      ,o.name AS objectName, u.name AS userName
FROM sys.database_permissions dp
 INNER JOIN sys.database_principals u
    ON u.principal_id = dp.grantee_principal_id
 INNER JOIN sys.objects o
    ON o.object_id = dp.major_id
WHERE dp.class = 1
  AND dp.minor_id = 0 AND o.object_id=object_id('t1')
```



なお、Management Studio から SQL Server on Linux にリモート接続できる場合は、次のように GUI 操作でオブジェクト権限を確認および設定することができます。



包含データベース（Contained Database）

SQL Server では、ログイン アカウントを作成して（認証用のアカウントの作成）、それに対応したデータベース ユーザーを作成するのが基本だと説明しましたが、SQL Server 2012 以降では、**包含データベース**（Contained Database）と呼ばれる機能が提供されて、ログイン アカウントを作成しなくても、データベース ユーザーを利用できる機能が追加されました。

包含データベース機能を利用するには、次のように SQL Server の**環境設定オプション**を変更する必要があります。

```
-- 包含データベースを利用するための環境設定オプションの変更
EXEC sp_configure 'contained database authentication', 1
RECONFIGURE
```

sp_configure は、SQL Server の環境設定オプションを設定するためのシステム ストアド プロシージャで、包含データベースを利用するには「**contained database authentication**」を **1** に設定する必要があります。

包含データベースは、次のように **CREATE DATABASE** ステートメントでデータベースを作成するときに、「**CONTAINMENT=PARTIAL**」を指定します。

```
-- 包含データベースの作成. cdb1 という名前で作成
CREATE DATABASE cdb1
 CONTAINMENT = PARTIAL
```

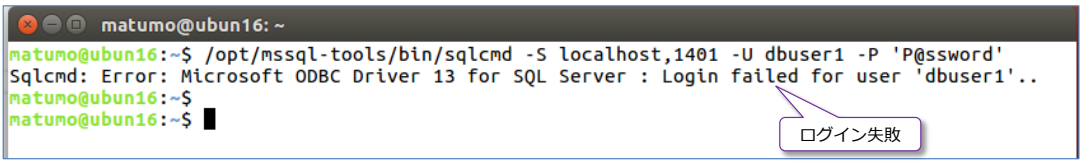
包含データベースを作成した後は、次のようにデータベース ユーザーを作成できます。

```
-- 包含データベース内にデータベース ユーザーの作成. dbuser1 という名前で作成
USE cdb1
go
CREATE USER dbuser1
 WITH PASSWORD = 'P@ssword'
```

CREATE USER ステートメントでデータベース ユーザーを作成するときに、**WITH PASSWORD** を付けて、任意のパスワードを設定することで、ログイン アカウントに紐付かない、データベース ユーザーを作成することができます。

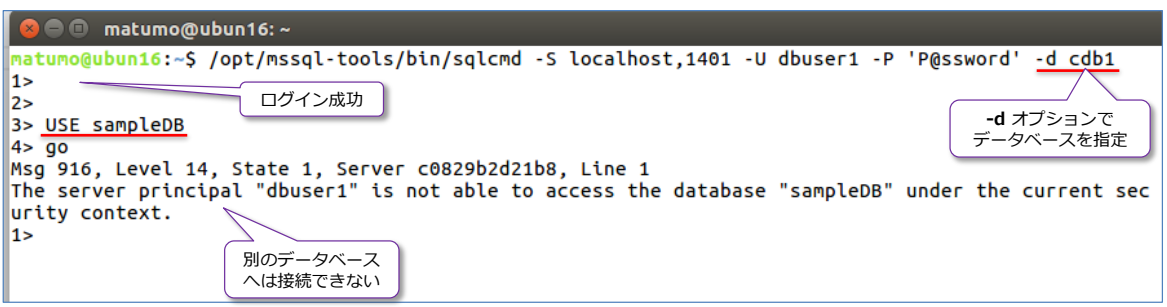
このように作成したデータベース ユーザー（**包含ユーザー**）は、次のようにログインすることはできません。

```
-- sqlcmd ツールでログイン
/opt/mssql-tools/bin/sqlcmd -S localhost,1401 -U dbuser1 -P 'P@ssword'
```



包含ユーザーの場合は、**-d** オプションを付けて、データベース名を明示指定する必要があります。

```
-- sqlcmd ツールでログイン
/opt/mssql-tools/bin/sqlcmd -S localhost,1401 -U dbuser1 -P 'P@ssword' -d cdb1
```



このように、包含データベースを利用すれば、ログイン アカウントを作成しなくても、データベース ユーザーを作成できるようになります。ただし、sqlcmd ツールや、データベース アプリケーションから接続する際には、データベース名の指定が必須になることに注意してください。また、該当データベースのみしかアクセスできない（他のデータベースに USE することはできない）ことにも注意する必要があります。

スキーマの作成（既定は dbo スキーマ）

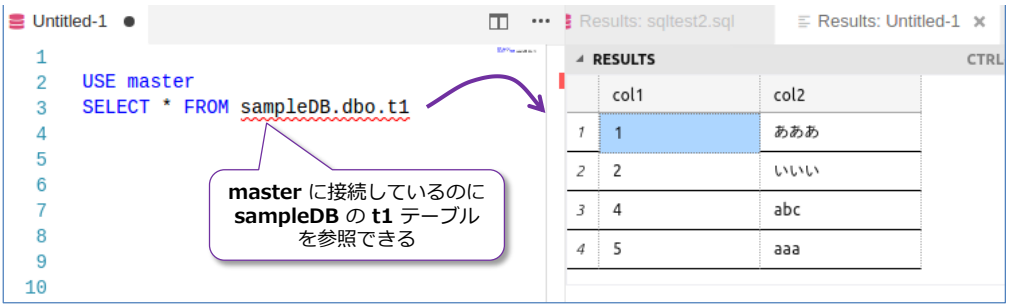
SQL Server では、スキーマを作成して、オブジェクトを分類することもできます。これまでの操作では、**sa** アカウントでテーブルを作成してきましたが、この場合は、**dbo** という名前のスキーマ（**dbo** は Database Owner の略）が利用されています。これまでの手順では、テーブルを操作するときにテーブル名のみを指定してきましたが、SQL Server では、次のように指定できます。

```
-- SELECT ステートメントの場合のテーブルの指定例
SELECT * FROM サーバー名.データベース名.スキーマ名.テーブル名
```

サーバー名は、後述の**リンク サーバー**機能を設定したときに利用できるものになりますが、**データベース名**の後に、**.**（ドット）を入れて**スキーマ名**、さらに **.**（ドット）を入れて**テーブル名**を指定する記述ができます。したがって、これまで利用してきた **sampleDB** データベースの **t1** テーブルは、**sa** アカウントが作成したものなので **dbo** スキーマが利用されていて、次のように指定できます。

```
-- t1 テーブルをデータベース名から指定
SELECT * FROM sampleDB.dbo.t1

-- dbo を省略することも可能（sa で操作している場合）
SELECT * FROM sampleDB..t1
```

このように、データベース名からテーブルを指定した場合には、該当データベースに接続（**USE**）していなくても、次のようにテーブルを操作できるようになるので便利です。



sa アカウントで操作している場合は、既定のスキーマが「**dbo**」に設定されているので、スキーマ名を省略した場合に、「**dbo**」が自動的に補われます。これによって、これまで作成したテーブルが dbo スキーマになっているので、参照時も「**sampleDB..t1**」といった形でスキーマ名を省略することができます。

ユーザー定義のスキーマの作成

スキーマを作成するには、**CREATE SCHEMA** ステートメントを利用します。

```
USE sampleDB
go
-- abc という名前のスキーマを作成
CREATE SCHEMA abc
```

スキーマを作成した後は、テーブルを作成するときなど、オブジェクトの作成時にそのスキーマを利用できるようになります。

```
-- abc スキーマの table1 テーブルを作成
CREATE TABLE abc.table1 ( a int )
-- データの追加
INSERT INTO abc.table1 VALUES(99)
-- データの参照
SELECT * FROM abc.table1
```

データベース ユーザーに対しては、次のように **ALTER USER** ステートメントを利用して、既定のスキーマ（**DEFAULT_SCHEMA**）を設定することもできます。

```
-- 既定のスキーマを設定. sqllogin1 ユーザーの既定のスキーマを abc に設定
ALTER USER sqllogin1
 WITH DEFAULT_SCHEMA = abc
```

このように、既定のスキーマを設定すると、**sqllogin1** ユーザーが「**SELECT * FROM table1**」を実行したときに「**abc.table1**」（abc が補われて）実行されます。なお、「**abc.table1**」というテーブルが存在しない場合は、「**dbo.table1**」のように dbo が補われます。

スキーマは、オブジェクト数が増えた場合に、オブジェクトをグループ化するために便利な機能です。なお、前掲のデータベースの一覧を取得するために利用した「**sys.databases**」ビューや、テーブルの一覧を取得するための「**sys.tables**」ビューの「**sys**」は、スキーマ名で、システム オブジェクトを分類するためのスキーマになっています。

4.5 Transact-SQL の制御構造（変数、IF、WHILE など）

SQL Server では、SQL ステートメントのことを「**Transact-SQL**」と呼びます。SQL ステートメントは、基本的なデータ操作に関しては ANSI SQL 標準のものがほとんどなので、どんなデータベースでも、ほぼ同じように操作することができます（Oracle でも、MySQL でも、PostgreSQL でも、DB2 でも基本操作であれば、同じように操作できます）。

一方、どのデータベースにも、標準外の拡張した SQL が存在しますが（これは **SQL の方言**とも呼ばれています）、SQL Server では、**Transact-SQL** という形で拡張しています。なお、Oracle では、拡張した SQL の部分のみを **PL/SQL** と呼んでいますが、SQL Server の場合は、標準 SQL も、拡張 SQL もすべて Transact-SQL ステートメントと呼んでいます。

Transact-SQL ならではの記述（変数や IF、WHILE）

SQL のプログラミング的な要素（変数や流れ制御）に関しては、データベース間で互換性がほとんどありませんが、Transact-SQL では、次のように変数などを利用します。

変数（ローカル変数）

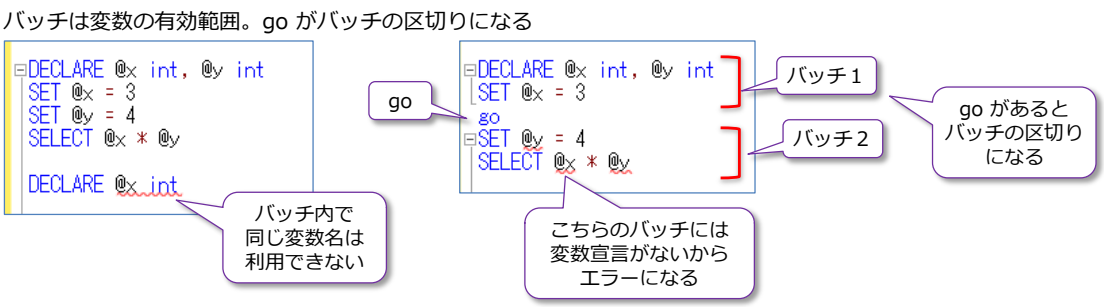
SQL Server での変数は、**DECLARE** で宣言して、変数名には **@** を付けるようにします。

```
-- 変数宣言。変数名には @ を付ける
DECLARE @val1 int = 999

DECLARE @x int, @y int
SET @x = 3        -- SET で値を代入
SELECT @y = 4     -- SELECT で値を代入
SELECT @x * @y    -- 結果は 12
```

変数への値の代入は、**SET** でも **SELECT** でもどちらでもできます。

バッチ（変数の有効範囲、区切りは go）

SQL Server では、**バッチ**という単位があります。これは変数の有効範囲で、**go** で区切られた範囲がバッチです。同じバッチ内では、同じ変数名は定義することができず、バッチをまたがった変数を用することもできません。



IF（条件分岐）

Transact-SQL での **IF** による条件分岐は、次のように利用します。

```
IF（条件式）
   BEGIN
       条件が真（true）の場合に実行したいステートメント
   END
ELSE
   BEGIN
       条件が偽（false）の場合に実行したいステートメント
   END
```

条件式のカッコ **()** は、省略することもできます。また、BEGIN と END は、実行したいステートメントが 1 つの場合にだけ省略することができます

WHILE（繰り返し処理）

Transact-SQL での **WHILE**（繰り返し処理）は、次のように利用します。

```
WHILE（条件式）
  BEGIN
     条件を満たしている間、実行したいステートメント
  END
```

IF と同様、条件式のカッコは省略することもできます。また、BEGIN と END は、実行したいステートメントが 1 つの場合にだけ省略することができます。

WHILE でデータを 10 件追加する場合は、次のように記述できます。

```
USE sampleDB

-- WHILE でデータを 10件追加
DECLARE @i int = 1
WHILE (@i <= 10)
  BEGIN
     INSERT INTO t1 VALUES(@i, 'sss')
     SET @i += 1
  END
```

Oracle PL/SQL と Transact-SQL の比較

Oracle の PL/SQL と Transact-SQL を比較すると、次のようになります。

	Oracle（PL/SQL）	**SQL Server（Transact-SQL）**

基本形	DECLARE
	　変数宣言部
	BEGIN
	　実行部（必須）
	EXCEPTION
	　例外処理部
	END;
		・フリー フォーマット
		・；は任意
		・どこでも変数宣言可能
		・ 例外処理は TRY ～ CATCH（応用編で解説）

変数宣言と値の代入	DECLARE
	変数名1 データ型 :=初期値;
	変数名2 データ型 :=初期値;
	BEGIN
	変数名1 := 値;
	変数名2 := 値;
	　　:
	END;
		DECLARE @変数名 データ型 = 初期値
		
		SELECT @変数名 = 値
		または
		SET @変数名 = 値

コメント	1行コメント -- 複数行コメント /*　*/	同じ
文字列出力	DBMS_OUTPUT.PUT_LINE('文字列')	PRINT '文字列'

IF による条件分岐	IF 条件式 THEN
	ステートメント;
	ELSIF 条件式 THEN
	ステートメント;
	ELSE
	ステートメント;
	END IF;
		IF 条件式
		[BEGIN]
		　ステートメント
		[END]
		ELSE
		[BEGIN]
		　ステートメント
		[END]

WHILE ループ	WHILE 条件式 LOOP ステートメント; END LOOP;	WHILE 条件式 [BEGIN] 　ステートメント [END]
LOOP	LOOP ステートメント; EXIT WHEN 条件; ステートメント; END LOOP;	なし
GOTO	GOTO LABEL; 　　: <<LABEL>>	GOTO LABEL 　　: LABEL:

組み込み関数の比較（Oracle、SQL Server）

SQL Server や Oracle では、便利な組み込み関数が多数用意されていますが、それぞれを比較すると、次のようになります。

	SQL Server	Oracle	役割
日付と時刻	GETDATE／CURRENT_TIMESTAMP	CURRENT_TIMESTAMP	現在の日付と時刻
	DATEADD	日付±n ADD_MONTHS	日付の加減算
	DATEPART、YEAR、MONTH	TO_CHAR(日付データ, 書式) EXTRACT(書式 FROM 日付データ)	文字列変換、部分抽出
	EOMONTH	LAST_DAY	月の最終日
	DATEDIFF	日付－日付 MONTHS_BETWEEN	日付の差分
変換	CONVERT(char(n),日付データ, x) FORMAT(日付データ, 書式)	TO_CHAR(日付データ, 書式)	日付を文字列変換
	CONVERT(char(n), 数値データ) FORMAT(数値データ, 書式)	TO_CHAR(数値データ, 書式)	数値を文字列変換
	CONVERT(decimal(p,s), 文字データ)	TO_NUMBER(文字データ, 書式)	文字列を数値変換
	CONVERT(datetime, 文字データ) DATEFROMPARTS	TO_DATE(文字データ, 書式)	文字列を日付変換

文字列に関する関数

SQL Server	Oracle	役割
CONCAT ／ + CONCAT_WS	CONCAT ／ ||	文字列連結
UPPER／LOWER	同じ	大文字、小文字変換
RTRIM／LTRIM	同じ	右／左から半角スペースの削除 Oracle ではスペース以外の文字も指定可能
TRIM	同じ	左右の指定文字削除
REPLACE／STUFF TRANSLATE	REPLACE	文字列の置換
RIGHT／LEFT	なし	右／左から部分抽出
SUBSTRING	SUBSTR	部分抽出
LEN／DATALENGTH	LENGTH／LENGTHB	文字列の長さ、バイト数
CHARINDEX	INSTR	文字列内の検索
なし	INITCAP	文字列の先頭を大文字変換
なし	RPAD／LPAD	右／左に指定文字を埋める
ASCII	同じ	文字の ASCII コード取得
CHAR	CHR	ASCII コードを文字変換

数値に関する関数

SQL Server	Oracle	役割
%	MOD	剰余
ROUND	ROUND	四捨五入
なし	TRUNC	切り捨て SQL Server では ROUND の第3引数に 1 を指定
POWER	POWER	べき乗
RAND	なし	乱数。Oracle では DBMS_RANDOM パッケージを利用
CEILING(n)	CEIL(n)	n に対してそれ以上の最小の整数
FLOOR(n)	FLOOR(n)	n に対してそれ以下の最大の整数
SQRT	SQRT	平方根
SIN／COS／TAN	同じ	三角関数

ビューやストアド プロシージャ

SQL Server でのビューおよびストアド プロシージャの利用方法は、次のとおりです。

ビュー

SQL Server でビューを作成する場合は、他のデータベースと同様、**CREATE VIEW** ステートメントを利用します。

```
-- 何かしらのステートメントがある場合は、CREATE VIEW の上に go を追加する
go
-- ビューの作成
CREATE VIEW ビュー名
AS
 SELECT ステートメント
go
```

SQL Server の場合は、このステートメント（CREATE VIEW）は、**バッチの先頭**でないと実行できないという決まりがあるので、CREATE VIEW の上に、**go** を入れて実行する必要があります。

ビューに対する変更は「**ALTER VIEW**」、削除は「**DROP VIEW**」で行います（Oracle では CREATE OR REPLACE という便利なステートメントがありますが、SQL Server では利用できないので、変更時は ALTER を利用する必要があります）。

ストアド プロシージャ

SQL Server でストアド プロシージャを作成する場合は、**CREATE PROCEDURE** ステートメントを利用します（中身は Transact-SQL で記述します）。

```
-- 何かしらのステートメントがある場合は、CREATE PROCEDURE の上に go を追加する
go
-- ストアド プロシージャの作成
CREATE PROCEDURE ストアド プロシージャ名
 入力パラメーターの定義
AS
  任意の Transact-SQL ステートメント
go
```

PROCEDURE は、PROC と省略することもできるので、**CREATE PROC** と記述できます。CREATE PROC は、CREATE VIEW と同様、バッチの先頭で実行する必要があります（上に go が必要です）。AS の中に記述する Transact-SQL ステートメントは、**フリー フォーマット**で記述することができます（SELECT ステートメントをそのまま記述することができ、複数の SELECT ステートメントを実行することもできます。これは筆者が SQL Server でとても気にいっている部分で、フリー フォーマットでいろいろな処理をストアド プロシージャ化していくことができます）。

ストアド プロシージャに対する変更は **ALTER PROC**、削除は **DROP PROC** で行います。

4.6 列ストア インデックスによる性能向上

SQL Server を利用する上で、必ず覚えておいてほしい機能が「**列ストア インデックス**」（カラムストア インデックス）です。これは、昨今のビッグデータ／DWH（データ ウェアハウス）環境では、メジャーとなっている**カラム指向データベース**の SQL Server の実装です。

SQL Server の列ストア インデックスは、まだカラム指向データベースが今のようにメジャーになる前（5 年以上前）の SQL Server 2012 のときから実装されていて、その前身となるエンジン（列指向エンジン）は SQL Server 2008 R2 のときに PowerPivot として実装・提供されています。

Let's Try

それでは、**列ストア インデックス**を試してみましょう。ここでは、列ストア インデックスを作成していない通常のテーブルの場合と、列ストア インデックスを作成したテーブルの場合の性能を比較を試してみます（**1 千万件**のデータを INSERT して性能比較します）。

なお、列ストア インデックスは、複数コア環境でより威力を発揮するので、検証する際には、複数コアを搭載した物理マシン、または複数コアを割り当てた仮想マシンを利用するのがお勧めです。

1. まずは、データベースを作成します。データベースの名前は「**csTestDB**」として、次のように **CREATE DATABASE** ステートメントを実行します。

```
-- データベース「csTestDB」の作成。/db ディレクトリへ作成
CREATE DATABASE csTestDB
 ON PRIMARY ( NAME = 'csTestDB'
             ,FILENAME = '/db/csTestDB.mdf'
             ,SIZE = 10GB )
 LOG ON ( NAME = 'csTestDB_log'
             ,FILENAME = '/db/csTestDB.ldf'
             ,SIZE = 5GB )
go
```

FILENAME で指定しているファイル パス（データ ファイルとトランザクション ログ ファイルの作成先となるディレクトリ）には、「**/db**」を指定していますが、皆さんの環境に合わせて適宜変更してください。また、ファイル サイズは **10GB** と **5GB** に設定しているので、**15GB** 分のディスク領域が必要になります。

テーブル「t1」の作成（通常のテーブル）と 1 千万のデータ追加

2. 次に、**CREATE TABLE** ステートメントを利用して、通常どおりに「**t1**」という名前のテーブルを作成します。

```
USE csTestDB
```

```
-- テーブル「t1」の作成
CREATE TABLE t1
 ( a int IDENTITY(1, 1)
  ,b int
  ,c char(200) DEFAULT 'dummy1'
  ,d char(200) DEFAULT 'dummy2'
 )
```

a、**b**、**c**、**d** の 4 つの列を作成して、**a** 列には **IDENTITY(1, 1)** と記述していますが、SQL Server では **IDENTITY** という機能で**連番**を作成することができます。**(1, 1)** と指定することで、1 から 1 つずつ増える連番を作成できます。Oracle でお馴染みの **SEQUENCE** は SQL Server にもあり、同じように利用できますが、IDENTITY のほうが簡単で利用しやすいので、ここでは IDENTITY を利用します。

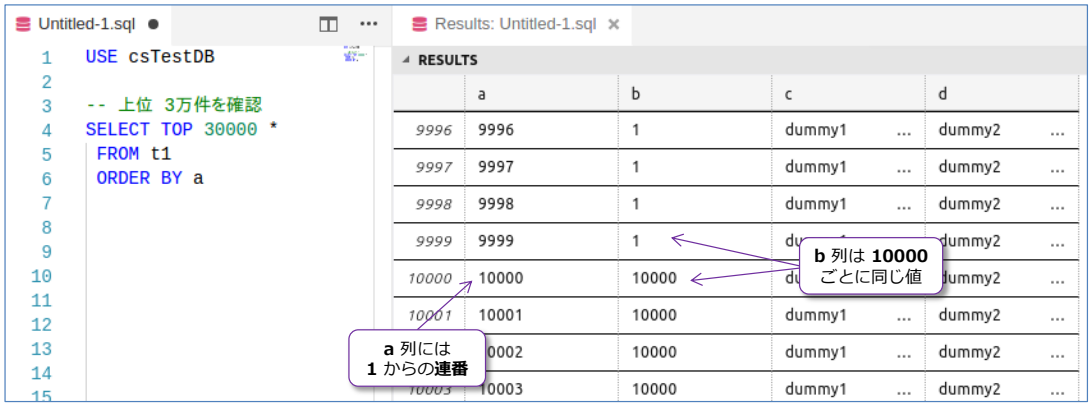
3. 次に、**WHILE** ループを利用して、**1,000 万件のデータ**を **INSERT** します。このとき、実行を速く終わらせるために、**SET NOCOUNT ON** を先頭に付けて、1 件 1 件（ループごと）の処理メッセージを受け取らないように設定しておきます。

```
SET NOCOUNT ON
DECLARE @i int = 1, @b int = 1
WHILE @i <= 10000000
  BEGIN
      IF @i % 10000 = 0 SET @b = @i
      INSERT INTO t1(b) VALUES(@b)
      SET @i += 1
  END
SET NOCOUNT OFF
```

1,000 万件のデータの追加になるので、環境にもよりますが、実行には 10 分～数時間くらいの時間がかかります（ディスクが低速な場合には、より実行時間が長くなります）。

4. データの追加が完了したら、次のように **SELECT** ステートメントを実行して、追加されたデータを確認しておきましょう。

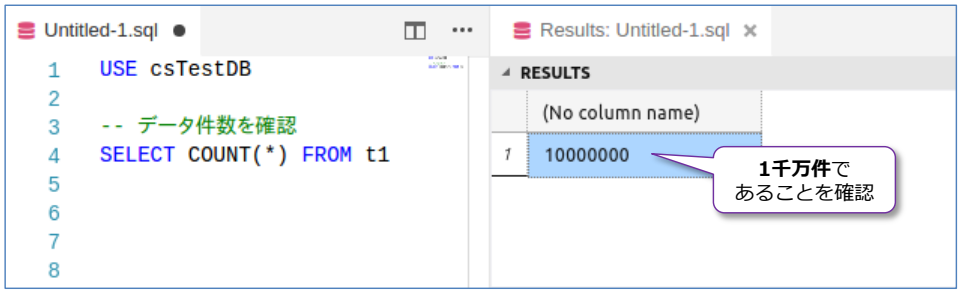
```
-- 上位 3万件を確認（空きメモリが 5GB 未満の場合は数分かかります）
SELECT TOP 30000 * FROM t1 ORDER BY a
```



a 列には、**IDENTITY** による連番、**b 列**には、**10,000 件**ごとに、**1**、**10,000**、**20,000**、**30,000** という値が入るようにしています。

5. 次に、**COUNT** 関数を利用して、データ件数が **1,000 万件**であることを確認しておきます。

```
SELECT COUNT(*) FROM t1
```

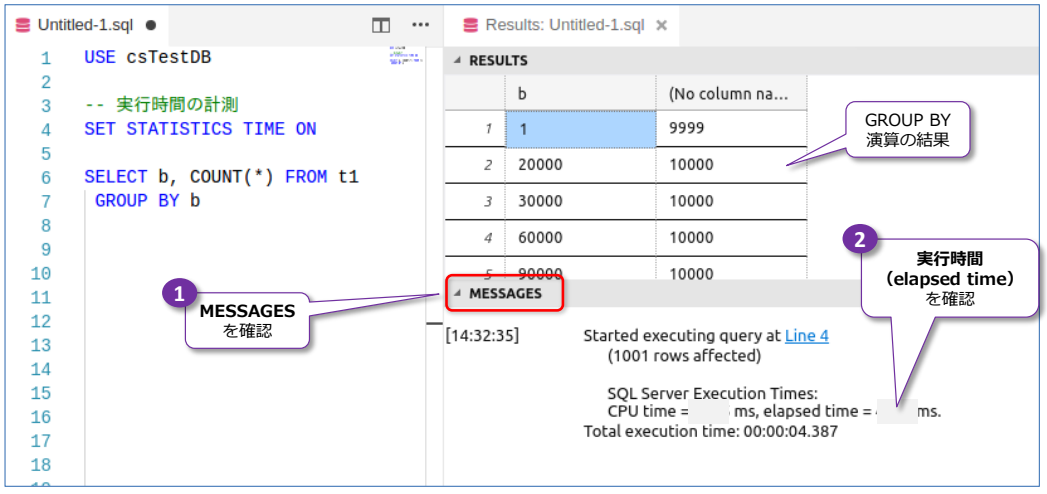


通常テーブルでの速度チェック（GROUP BY 演算）

次に、通常テーブルでの実行速度をチェックします（この後に作成する列ストア インデックスとの比較を行うために、通常テーブルでの速度をチェックしておきます）。SQL Server では、クエリの実行速度をチェックするために、**SET STATISTICS TIME ON** というコマンドがあり、これをセットすれば、クエリの実行時間や CPU 利用時間を測定することができます（ミリ秒精度での実行時間を計測できます）。

1. まずは、**SET STATISTICS TIME ON** を次のように設定して、**GROUP BY 演算**を利用した **SELECT** ステートメントを実行します。

```
-- 実行時間の計測
SET STATISTICS TIME ON

SELECT b, COUNT(*) FROM t1
 GROUP BY b
```

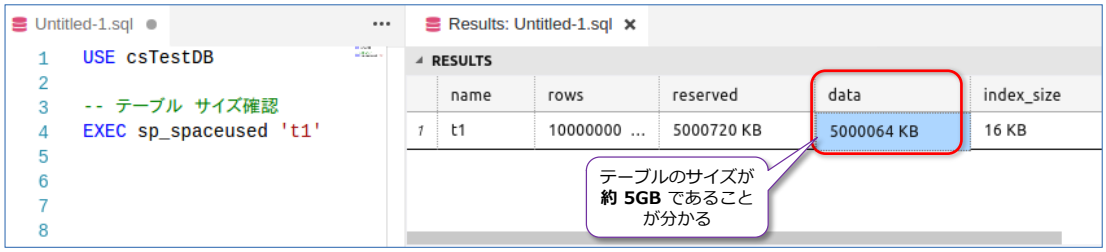


実行が完了すると、**MESSAGES** の中に **elapsed time** があり、これで実行時間を確認でき

ます。これをメモしておいてください。

2. 次に、**sp_spaceused** というシステム ストアド プロシージャを利用して、テーブルのサイズを確認しておきます。

```
-- テーブル サイズの確認
EXEC sp_spaceused 't1'
```



結果の「**data**」列でテーブル サイズを確認することができ、**約 5GB** のサイズであることを確認できます。

列ストア インデックスの作成／性能チェック

次に、**列ストア インデックス**（カラム ストア インデックス）を作成した場合の性能を調べてみましょう。列ストア インデックスには、テーブル データそのものをカラム ストアに変換する「**クラスター化列ストア インデックス**」（通称 **CCSI**: Clustered Column-Store Inex）と、既存のテーブルに対してインデックスとして列ストアを追加する「**非クラスター化列ストア インデックス**」（通称 **NCCSI**）の 2 種類があります。どちらを利用しても、大きな性能向上を実現できますが、ここでは、より結果が分かりやすくなる**クラスター化列ストア インデックス**（CCSI）を利用した手順で説明します。

1. まずは、クラスター化列ストア インデックスを試すためのテーブルを「**t1_ccsi**」という名前で作成します（**t1** テーブルと全く同じ構成にしておきます）。

```
-- まずは同じ構造のテーブルを作成
CREATE TABLE t1_ccsi
 ( a int IDENTITY(1, 1)
  ,b int
  ,c char(200) DEFAULT 'dummy1'
  ,d char(200) DEFAULT 'dummy2'
 )
```

2. 次に、**1 千万件**分のデータを **INSERT** しますが、**t1** テーブルと同じデータにするために（公平なテストにするために）、次のように **INSERT .. SELECT** ステートメントを実行するようにします。

```
INSERT INTO t1_ccsi (b, c, d)
 SELECT b, c, d FROM t1
```

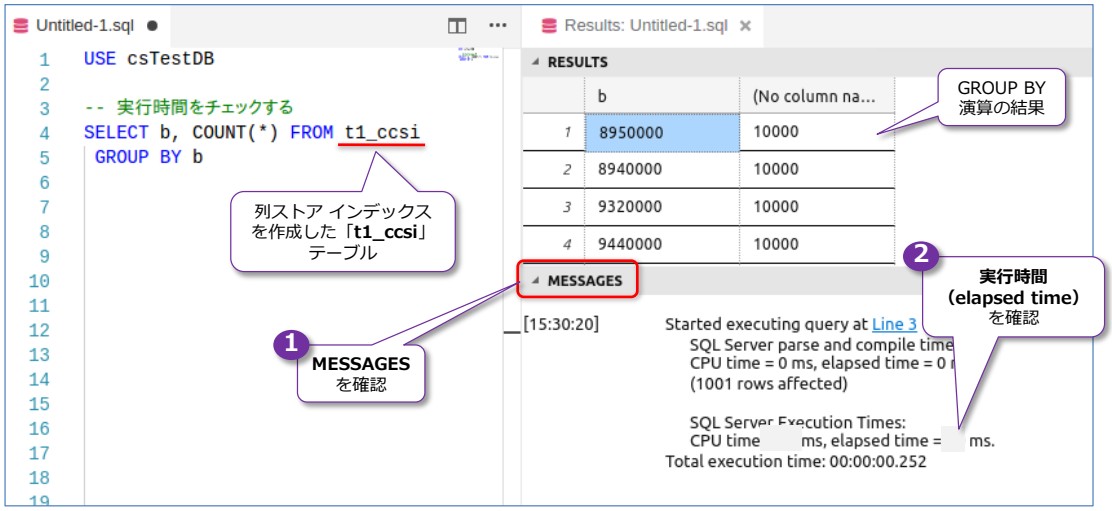
3. 次に、**クラスター化列ストア インデックス**を作成しますが、これには **CREATE CLUSTERED COLUMNSTORE INDEX** というステートメントを利用します。

```
CREATE CLUSTERED COLUMNSTORE INDEX idx1
 ON t1_ccsi
```

idx1 はインデックス名になります。クラスター化列ストア インデックスでは、実際のテーブル データそのものを、丸ごとカラム ストア構造に変換するので、すべての列データがカラム ストアになるため、インデックスの作成時に列を指定する必要はありません（すべての列が対象になります）。

4. クラスター化列ストア インデックスの作成が完了したら、次は、通常テーブルに対して実行したのと同じ **SELECT** ステートメント（**GROUP BY** 演算）を実行して、**実行時間**を確認してみましょう。

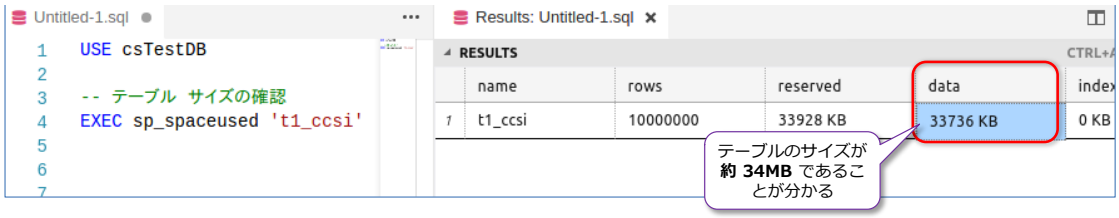
```
-- 実行時間をチェックする
SELECT b, COUNT(*) FROM t1_ccsi
 GROUP BY b
```



1 回目の実行はコンパイル時間が加算されるので、2 回、3 回、4 回と実行して、そのときの実行時間（**elapsed time**）を確認してみてください。**実行時間**が桁違いに小さくなっていることを確認できると思います（弊社環境では、**100 倍**以上速く実行できています）。

5. 次に、**sp_spaceused** システム ストアド プロシージャを実行して、テーブルのサイズを確認しておきます。

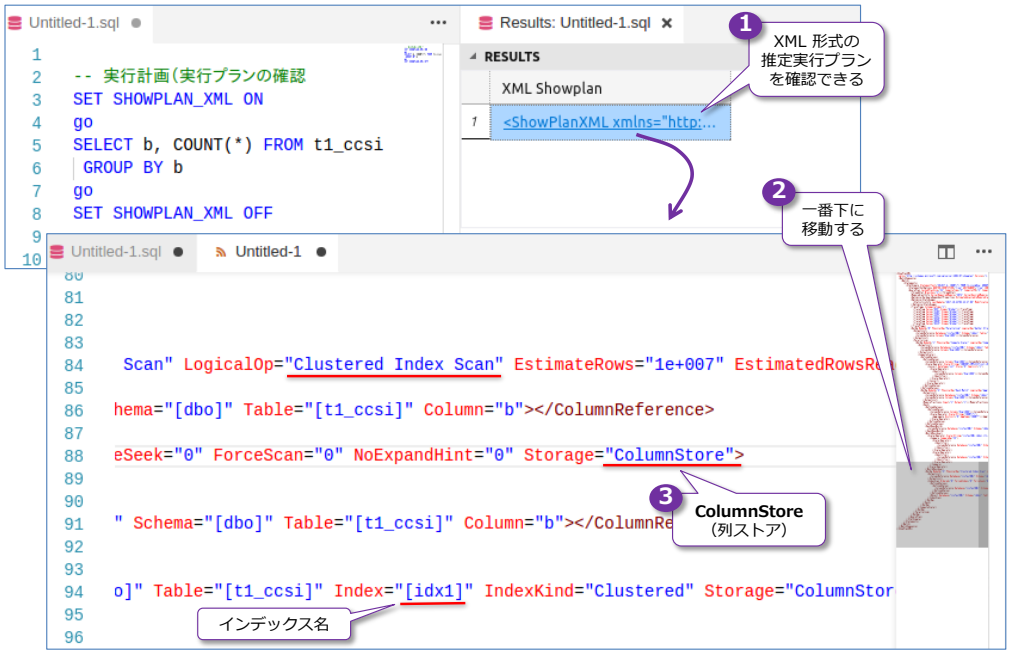
```
-- テーブル サイズの確認
EXEC sp_spaceused 't1'
```



サイズは、わずか**約 34MB** になっていることを確認できます。通常テーブルは**約 5GB** のサイズでした。今回のテストで格納したデータは、**c** および **d 列**に、まったく同じデータ（**dummy1**）を格納しているので、極端に圧縮率が高くなっていることを差し引く必要がありますが、実際、弊社のお客様環境でも、圧縮率が非常に高いことを確認しており、かつクエリ性能も向上している案件が多数あります（特に大規模 DWH 系の集計クエリでは大きな成果を得られています）。ぜひ、皆さんの環境でも、列ストア インデックスを作成して、サイズや性能を確認してみてください。

6. 次に、GROUP BY 演算を実行したときの**実行計画**（実行プラン）を確認してみましょう。Management Studio を利用している場合は、ツールバーの［**推定実行プラン**］ボタンをクリックするだけで、グラフィカルな実行プランを確認できるのですが、Visual Studio Code（mssql 拡張機能）の場合は、次のように **SET SHOWPLAN_XML ON** を実行して、XML 形式の実行プランを確認するようにします。

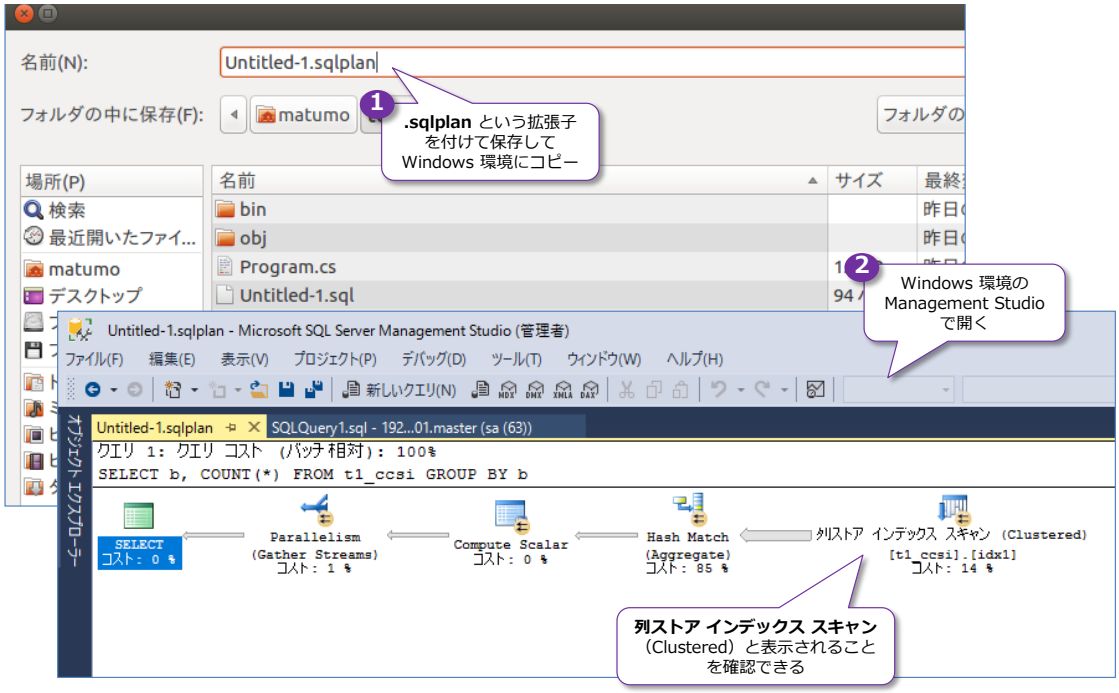
```
-- 実行計画（実行プランの確認
SET SHOWPLAN_XML ON
go
SELECT b, COUNT(*) FROM t1_ccsi
 GROUP BY b
go
SET SHOWPLAN_XML OFF
```



実行プランを一番下に移動すると、**ColumnStore** や **idx1**（インデックス名）などのキーワ

ードを確認できることから、列ストア インデックスを利用した内部処理であったことを確認できます。

なお、この XML 形式の実行プランは、**.sqlplan** という拡張子を付けて、Windows 環境の Management Studio で表示すれば、グラフィカルな実行プランとして、中身を確認することもできます。



以上のように、列ストア インデックスは、SQL Server の性能向上に大きく貢献する機能なので、ぜひ活用してみてください。

4.7 リンク サーバー、CSV のインポート／エクスポート

SQL Server では、リモートのサーバーとのデータのやりとりに便利な**リンク サーバー**、CSV ファイルのインポート／エクスポートで便利な **bcp** コマンド／**BULK INSERT** ステートメントなどがあります。これらは Linux 上でも利用できるので、ここではその利用方法を説明します。

リンク サーバーによるリモート データの取得

リンク サーバーは、リモートのサーバーとのやりとりで欠かせない非常に便利な機能です。Linux 上でこれを利用するには、次のように **sp_addlinkedserver** と **sp_addlinkedsrvlogin** システム ストアド プロシージャを実行します。

```
-- リモート SQL Server (192.168.1.50) に対してリンク サーバーを作成
EXEC master.dbo.sp_addlinkedserver
   @server = N'192.168.1.50,1433'
  ,@srvproduct=N'SQL Server'

-- ログイン設定 (リモート SQL Server 上の sa でログイン)
EXEC master.dbo.sp_addlinkedsrvlogin
   @rmtsrvname = N'192.168.1.50,1433'
  ,@useself=N'False'
  ,@locallogin=NULL
  ,@rmtuser=N'sa'
  ,@rmtpassword='P@ssword'
```

sp_addlinkedserver では、**@server** にリモート サーバー名を指定しますが、「**IP アドレス,ポート番号**」形式で、リモート サーバーが **1433** ポートを利用している場合は、ポート番号は省略可能です。また、DNS や hosts を利用している場合は、IP アドレスの代わりにホスト名を指定することもできます。**@srvproduct** では、リモート サーバーが SQL Server の場合は **N'SQL Server'** と指定します。

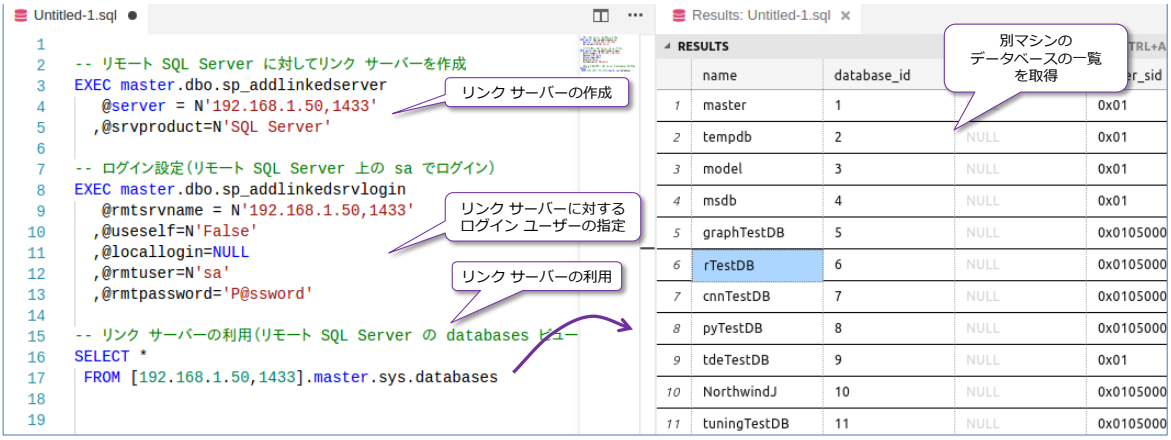
sp_addlinkedsrvlogin では、リモート サーバーにログインする設定を行いますが、**@rmtsrvname**（リモート サーバー名）には、**sp_addlinkedserver** の **@server** に指定したのと同じもの、リモート サーバー上の sa アカウントでログインするには、**@useself** を **N'False'**、**@locallogin** を **NULL** を指定して、**@rmtuser** に **N'sa'**、**@rmtpassword** に **'sa のパスワード'** を指定します。

以上でリンク サーバーの設定が完了です。後は、登録した名前を利用して、「**リンクサーバー名.データベース名.スキーマ名.オブジェクト名**」形式で、リモート サーバーのオブジェクトを操作することができます。例えば、次のようにステートメントを実行できます。

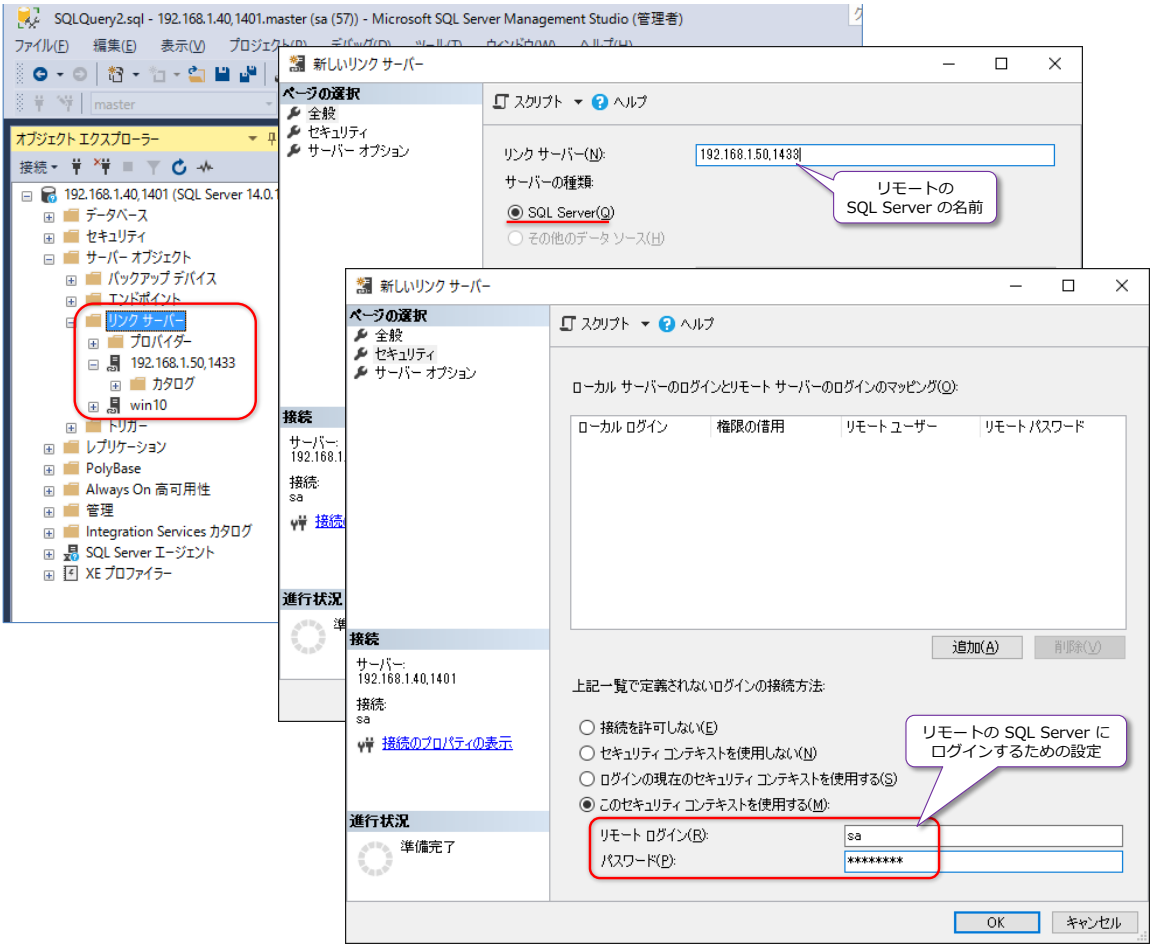
```
-- リンク サーバーの利用 (リモート SQL Server の databases ビューの参照)
SELECT * FROM [192.168.1.50,1433].master.sys.databases
```

リンク サーバー名を **[ ]** で囲んでいるのは、数字で始まるオブジェクト名が**特殊な名前**であるた

めです（SQL Server では特殊な名前を利用するには **[ ]** で囲む必要があります）。リンク サーバー名の後には、**データベース名**に **master**、**スキーマ名**に **sys**、**オブジェクト名**に **databasese** システム ビューを指定して、実行しています。



なお、Windows 環境の Management Studio から Linux 上の SQL Server をリモート操作できる場合には、次のように GUI を利用して、リンク サーバーを簡単に作成することができます。

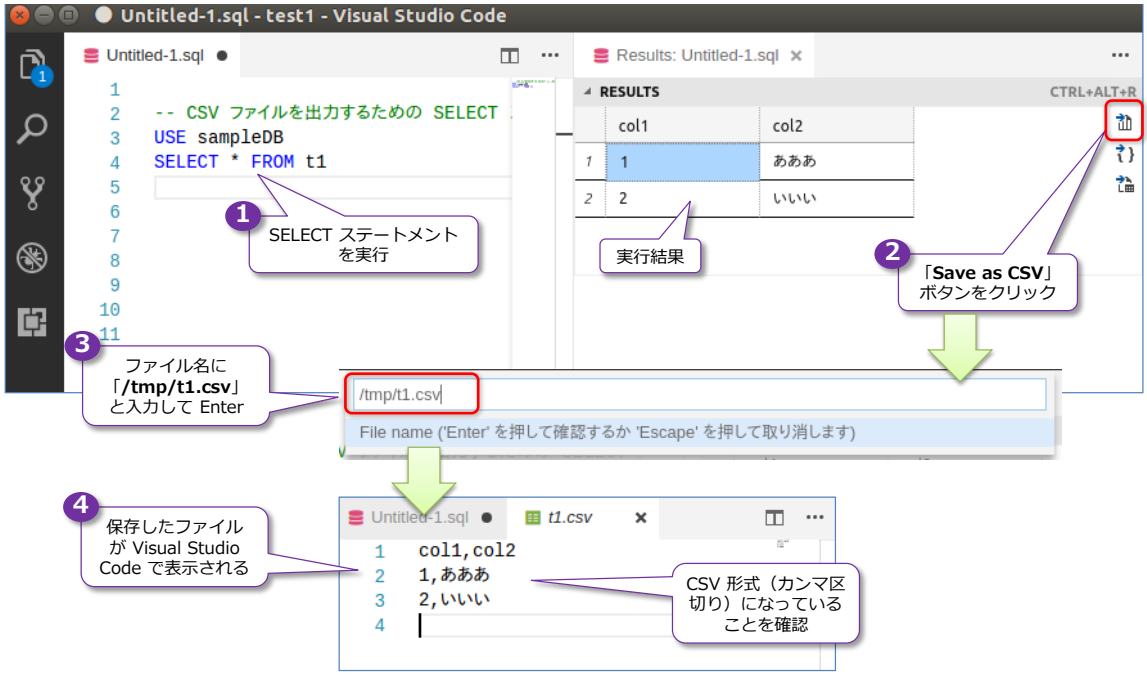


CSV ファイルのインポート（bcp）

SQL Server では、**CSV ファイル**（カンマ区切りのテキスト ファイル）をインポートすることも簡単に行えます。これを行うには、**bcp** コマンドや **BULK INSERT** ステートメントを利用しますが、まずは、CSV 形式のファイルを作成してみましょう。これは、**Visual Studio Code** の **mssql 拡張機能**を利用すると簡単に行えます。次のように SELECT ステートメントを実行します。

```
-- CSV ファイルを作成するための SELECT ステートメント
USE sampleDB
SELECT * FROM t1
```

ステートメントの実行後、実行結果（**RESULTS**）の右側にボタンが 3 つあり、実行結果を CSV や JSON、Excel 形式で保存できます。CSV 形式で保存するには、次のように一番上のボタン（**Save as CSV**）をクリックします。



Save as CSV ボタンをクリックすると、ファイル名を聞かれるので「**/tmp/t1.csv**」のように入力して、保存します（このファイル パスは、この後のインポート手順で利用します）。

次に、**t1.csv** ファイルをインポートするためのテーブル（**t1** と同じ構造の **t1_copy** テーブル）を次のように作成します。

```
-- 同じ構造のテーブルを作成
USE sampleDB
SELECT * INTO t1_copy FROM t1 WHERE 1 = 2
```

SQL Server では、このように **SELECT .. INTO** を利用すると、SELECT ステートメントの結果をもとに新しいテーブルを作成することができます（Oracle での CREATE TABLE AS SELECT に相当するものです）。**INTO** では新しいテーブルの名前を **t1_copy** と指定して、t1 テーブルと同じ構造のテーブルでデータは空（データの複製はしない）にするために、**WHERE 1=2** を付けて、

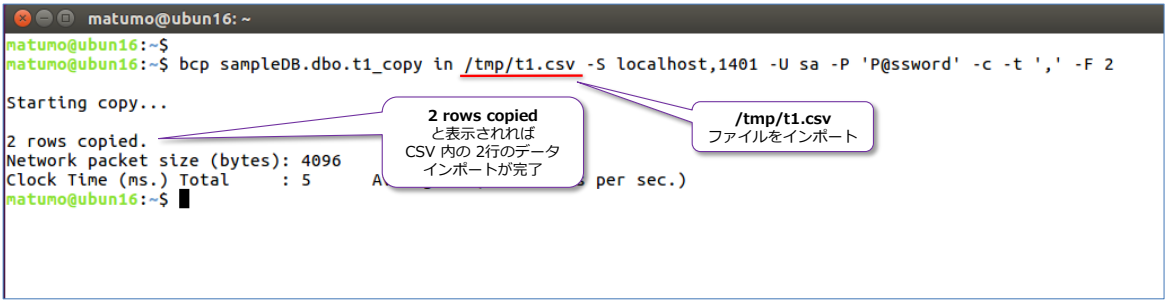
t1 テーブルの検索結果が **0 件**になるようにしています。

次に、**bcp** コマンドを利用して、**t1_copy** テーブルに、**t1.csv** ファイルをインポートします。**bcp** コマンドは、**sqlcmd** ツールをインストールしていれば、それと同じディレクトリ「**/opt/mssql-tools/bin/**」にインストールされています。もし、sqlcmd ツールをインストールしていない場合は、Step 2 を参考にインストールしてみてください。

bcp コマンドの利用方法

bcp コマンドは、ターミナルを利用して次のように実行します（コマンドは 1 行で記述するようにします）。

```
# bcp コマンドで CSV ファイルをインポート
bcp sampleDB.dbo.t1_copy in /tmp/t1.csv -S サーバー名,ポート番号 -U sa
-P 'saのパスワード' -c -t ',' -F 2
```



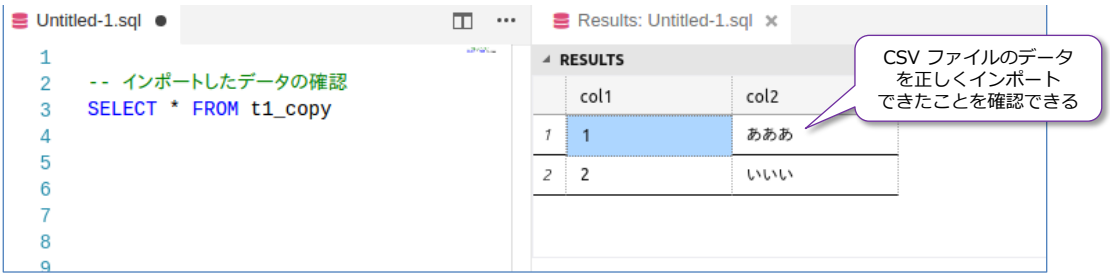
bcp コマンドでは、「**テーブル名 in ファイルパス**」と記述することで、ファイルをテーブルにインポートすることができます。テーブルを格納している SQL Server は、**-S** オプションで指定して（ポート番号はカンマ区切りで指定。ポート番号に 1433 を利用している場合はポート番号を省略可能）、**-U** で sa、**-P** で sa のパスワードを入力してログインします。

-c オプションの **c** は character（文字列）の略で、ファイルがテキスト データであることを指定し、**-t** オプションの **t** は teminator（終端）の略で、列の区切り文字を指定します。**','** と記述することで、カンマ区切りを指定できます。

最後の **-F** オプションの **F** は First row（最初の行）の略で、2 を指定することで、CSV ファイルの 2 行目からデータを取得するという指示になります。Visual Studio Code で出力した CSV ファイルは、1 行目に列名が出力されているので、これをスキップするために **-F 2** と指定しています。

bcp の実行が完了したら、取り込んだデータを確認しておきます。

```
-- インポートしたデータの確認
SELECT * FROM t1_copy
```



このように、CSV ファイルの取り込みも、bcp コマンドを利用することで簡単に行うことができます。

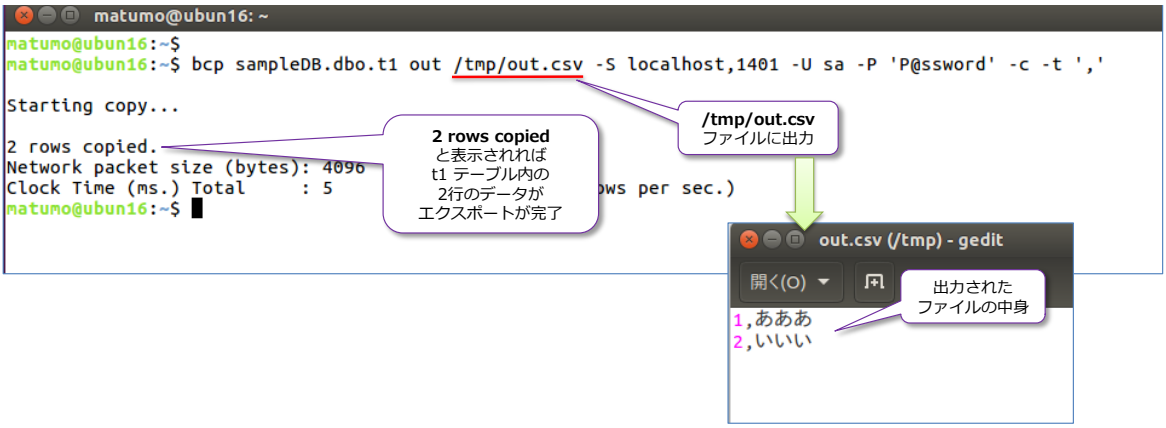
CSV ファイルのエクスポート（bcp out）

bcp コマンドは、インポートするだけでなく、エクスポート（CSV ファイルとして出力）する機能もあります。インポート時は、「**テーブル名 in ファイルパス**」のように **in** を指定しましたが、エクスポートをする場合は「**テーブル名 out ファイルパス**」のように **out** を指定します。

これは、次のように記述できます。

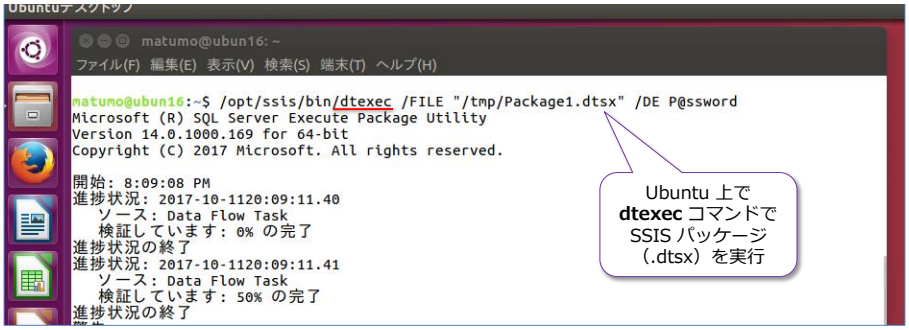
```
# t1 テーブルのデータを /tmp/out.csv ファイルにエクスポート
bcp sampleDB.dbo.t1 out /tmp/out.csv -S localhost,1401 -U sa -P 'P@ssword' -c -t ','
```

in のときと同じオプションで、**-S** で SQL Server を指定、**-U** と **-P** で sa でログイン、**-c** と **-t ','** で CSV 形式を指定する形になっています。

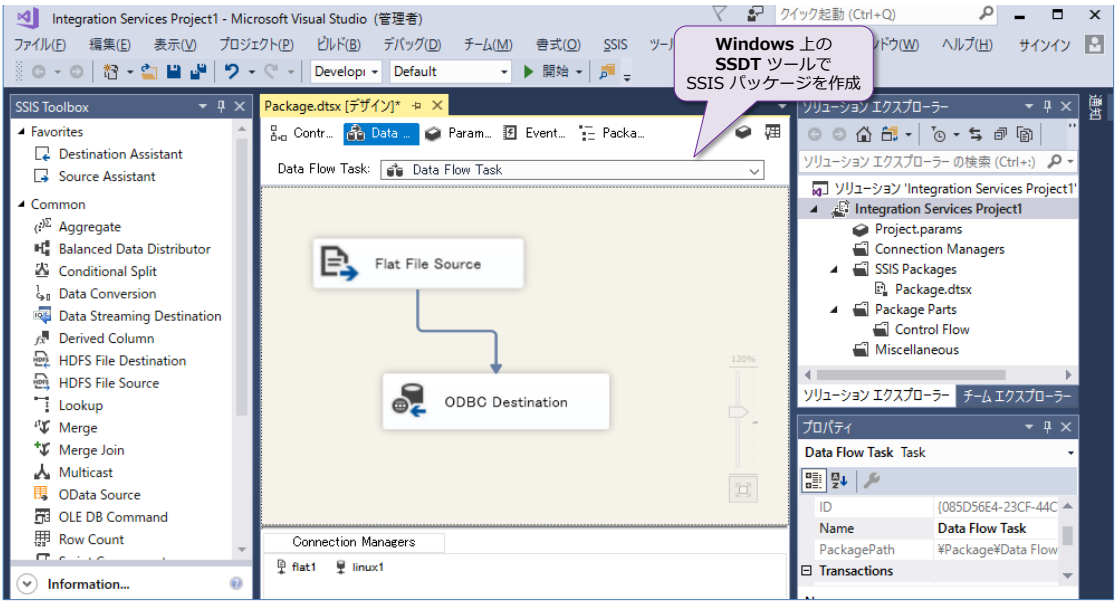


SSIS パッケージ（.dtsx）の実行

CSV ファイルのインポートやエクスポート、リモート サーバーとの連携は、**SSIS**（**SQL Server Integration Services**）を利用しても行えます。SQL Server on Linux では、**SSIS パッケージ（.dtsx）**の実行をサポートしています。



SSIS パッケージそのものは、**Windows** 上で **SSDT**（SQL Server Data Tools）を利用して作成する必要があり、それを Linux 上にコピーすれば、Linux 上でパッケージを実行できます。



なお、SSDT では、**Linux 形式のパス**（/tmp/fol1/aaa.csv 形式）をサポートしていないので、例えば Flat File Source としてこの形式のパスを記述すると、デザイン上のエラーになります。これを回避するには、Windows 上で同じディレクトリ構成を作成しておき、そこにファイルを配置するようにします（Linux 上で **/tmp/fol1/aaa.csv** という CSV ファイルを読み込みたいなら、Windows 上に **C:¥tmp¥fol1¥aaa.csv** というフォルダー構成でファイルを配置しておきます。C:¥ の部分は、Linux 上でのパッケージ実行時に自動的に外れてくれるので、/tmp～ パスとして認識してくれます）。その他の SSIS on Linux に関する情報は,以下が参考になります。

SSIS on Linux のインストール方法
https://docs.microsoft.com/en-us/sql/linux/sql-server-linux-setup-ssis

SSIS on Linux の制限事項など
https://docs.microsoft.com/en-us/sql/linux/sql-server-linux-ssis-known-issues

4.8 データベースのバックアップとリストア

次に、SQL Server におけるデータベースのバックアップとリストアについて説明します。バックアップは **BACKUP DATABASE**、リストアは **RESTORE DATABASE** ステートメントを利用して実行することができますが、どちらも非常にシンプルな構文です（以下）。

```
-- バックアップの構文
BACKUP DATABASE データベース名
 TO DISK = 'ファイルパス'
  [ WITH オプション ]

-- リストアの構文
RESTORE DATABASE データベース名
 FROM DISK = 'ファイルパス'
  [ WITH オプション ]
```

バックアップでは、**TO DISK** で出力先のファイル名を指定して、リストアでは **FROM DISK** でリストアしたいバックアップ ファイルの名前を指定するだけです。

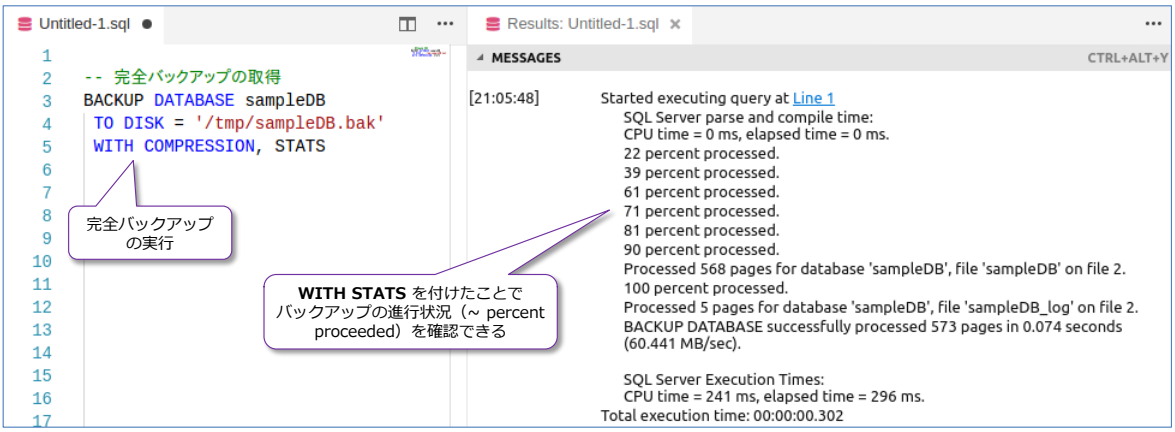
上記の BACKUP は、完全バックアップ（データベースの全体バックアップ）になり、差分をバックアップするには **WITH** オプションで **DIFFERENTIAL** を指定します。また、トランザクション ログのバックアップを行いたい場合は **BACKUP LOG** ステートメント（DATABASE の部分が LOG に変わるだけ）を利用します。

Let's Try ～完全バックアップの実行： BACKUP DATABASE～

それでは、これを試してみましょう。

まずは、データベース（**sampleDB**）の完全バックアップを取得してみます。

```
-- 完全バックアップの取得
BACKUP DATABASE sampleDB
 TO DISK = '/tmp/sampleDB.bak'
 WITH COMPRESSION, STATS
```



バックアップ先には、**/tmp** ディレクトリを指定していますが、皆さんの環境に合わせて適当な場所に変更してください。**WITH** オプションでは **COMPRESSION** と **STATS** を指定していますが、どちらも付けて実行するのがお勧めになります。

COMPRESSION は、圧縮バックアップを実行できるオプションで、これによってバックアップのファイルをサイズを小さくできるので、バックアップ時のディスクへの書き込み量を大幅に減らすことができます。バックアップ時間の大半はディスクへの書き込み待ちになることが多いので、圧縮バックアップにすることで、バックアップ時間を短縮できるようになります。

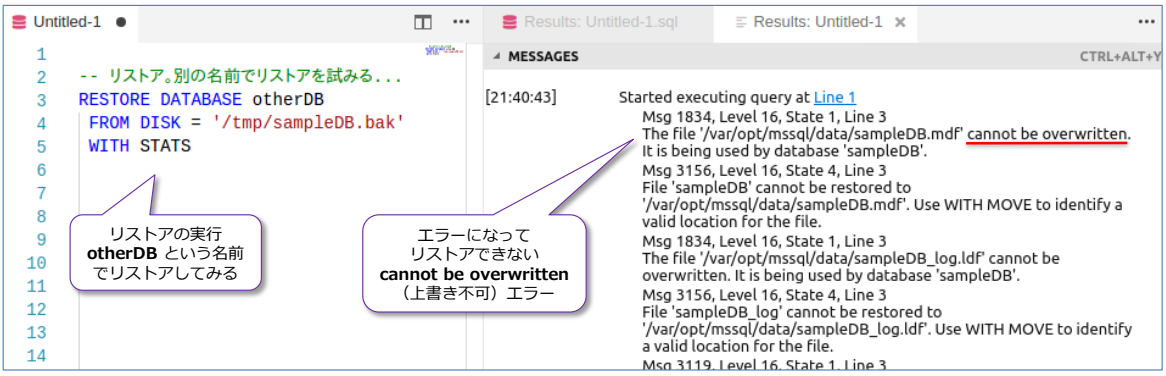
STATS は、バックアップの進行状況をメッセージで確認できるオプションで、データベース サイズが大きい場合に、どのぐらいバックアップが進行しているのかをこれで確認できるので、大変便利です。

なお、バックアップ ファイルの名前は **.bak** と付けていますが、任意の拡張子で大丈夫です（SQL Server では **.bak** と付ける慣習があります）。

リストアの実行 ～RESTORE DATABASE～

次に、リストアを試してみましょう。今取得したバックアップ ファイル（**/tmp/sampleDB.bak**）を利用して、違う名前のデータベースとしてリストアをしてみましょう。

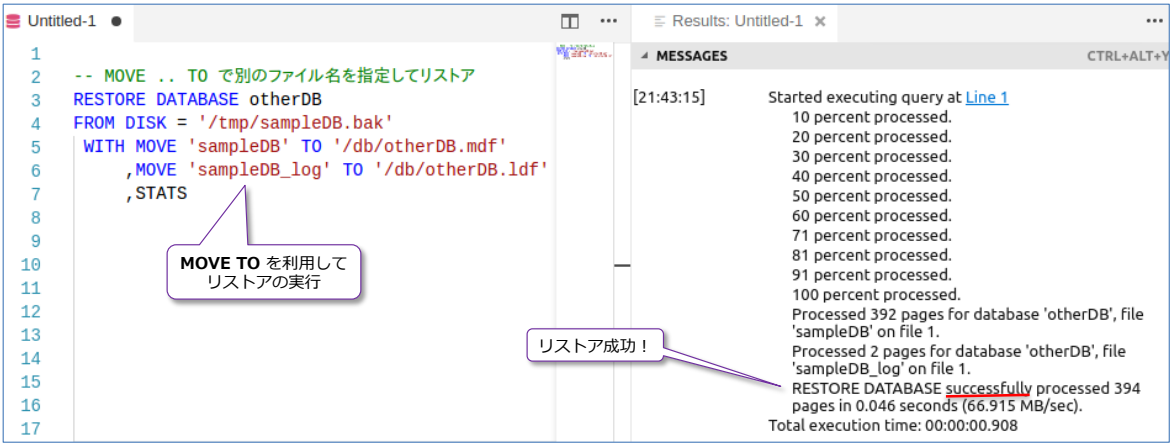
```
-- リストア。別の名前でリストアを試みる...
RESTORE DATABASE otherDB
 FROM DISK = '/tmp/sampleDB.bak'
```



しかし、これはエラーになってリストアすることができません。これは、**RESTORE DATABASE** ステートメントが、元々のデータベースと同じ名前のファイルを復元しようとしているために発生しています。バックアップを取得したデータベース（**sampleDB**）は、まだ存在しているので、RESTORE によって、同じ場所にファイルを復元しようとすると、該当ファイルが存在しているので、それを上書きすることを避けるようになっています。

RESTORE DATABASE ステートメントでは、**WITH** オプションで **MOVE .. TO** を利用することで、データベース ファイルのリストア先を変更することができます。これは非常に良く利用するオプション（バックアップしたマシンとリストアするマシンでディレクトリ構成が異なる場合には必須のオプション）になるので、次のように実行してみてください。

```
-- MOVE .. TO で別のファイル名を指定してリストア
RESTORE DATABASE otherDB
FROM DISK = '/tmp/sampleDB.bak'
 WITH MOVE 'sampleDB' TO '/db/otherDB.mdf'
     ,MOVE 'sampleDB_log' TO '/db/otherDB.ldf'
     ,STATS
```

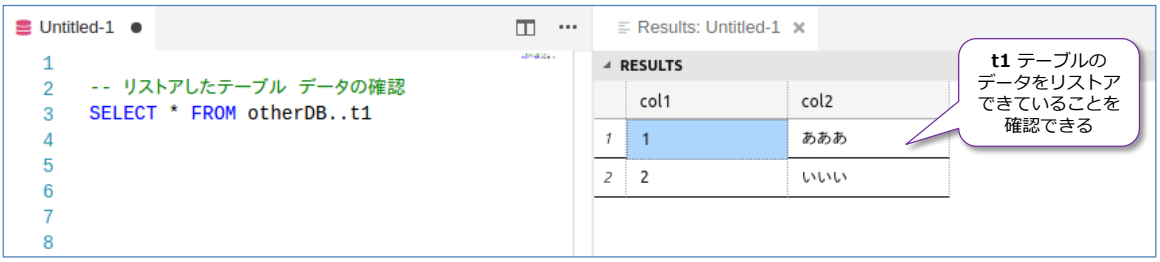


今度はリストアが成功します。**MOVE .. TO** は「**MOVE '論理名' TO '新ファイルパス'**」という形で利用しますが、論理名は **CREATE DATABASE** ステートメントでデータベースを作成するときに **NAME=** で指定した名前です。もし、「**CREATE DATABASE sampleDB**」という形で **NAME** を指定せずにデータベースを作成している場合は、データ ファイル（**.mdf**）の論理名はデータベース名と**同じ名前**、トランザクション ログ ファイル（**.ldf**）の論理名は「**データベース名_log**」に自動設定されています。

したがって、「**MOVE 'sampleDB' TO '/db/otherDB.mdf'**」で元々の sampleDB データベースのデータ ファイル (.mdf) を **/db** ディレクトリの下に **otherDB.mdf** という名前でリストアして、「**MOVE 'sampleDB_log' TO '/db/otherDB.ldf'**」でトランザクション ログ ファイル（.ldf）を **/db** ディレクトリの下に **otherDB.ldf** という名前でリストアするという指示ができます。

リストアが完了した後は、**t1** テーブルの中身を参照してみましょう。

```
-- リストアしたテーブル データの確認
SELECT * FROM otherDB..t1
```

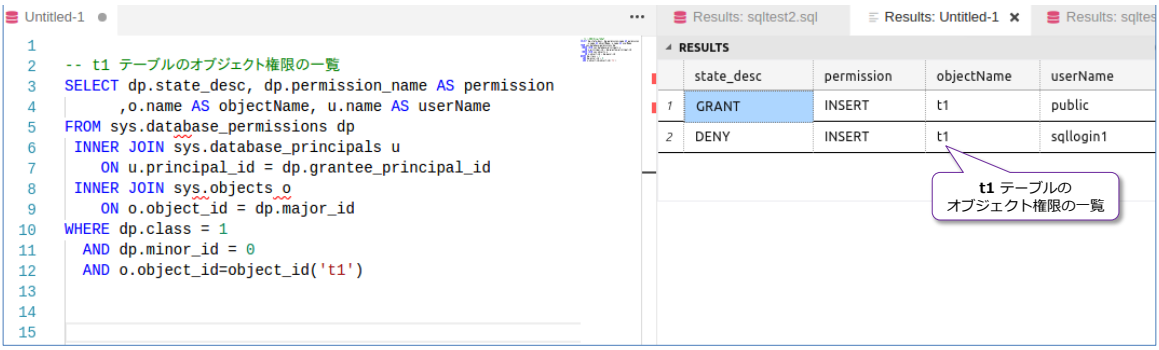


このように、データベースをリストアすれば、テーブル データを丸ごと復元することができます。また、テーブル データだけでなく、データベース内に作成していたものは、基本的にほぼ全てがリストアされるので、ストアド プロシージャやビュー、ユーザー、オブジェクト権限の設定、列スト

ア インデックスなども復元することができます。

オブジェクト権限の確認、DB ユーザーとログインの再マッピング

sampleDB データベースの **t1** テーブルには、Step 4.4 で **sqllogin1** データベース ユーザーに対して、**INSERT** 権限の **DENY**（拒否）、**public**（全ユーザー）に対しては **INSERT** 権限を **GRANT**（許可）していました。



また、**sqllogin1** データベース ユーザーは、**db_datareader** ロールを付与（全てのテーブルの読み取り権限を付与）していました。

ここでは、これらのオブジェクト権限がリストアされているかどうかを、リストアした **otherDB** データベースで確認してみますが、実際のリストア（本番環境における障害発生時のリストア）をシミュレートするために、次のように **sqllogin1** ログイン アカウントを削除した状態にしておきます。

```
-- sqllogin1 ログイン アカウントを削除しておく
USE master
DROP LOGIN sqllogin1
```

ログイン アカウントは、**master** データベース内に存在するものなので、障害が発生したときに、新規インストールした SQL Server や、別の SQL Server にデータベースをリストアする場合には、該当ログイン アカウントが存在しないという事態が発生します。これをシミュレートするために、**DROP LOGIN** でログイン アカウントを削除しています。

次に、新環境を想定して、ログイン アカウントを作成し直します（同じ名前・同じパスワードで作成します）。

```
-- 新環境を想定して同じ名前・同じパスワードでログイン アカウントを作成
CREATE LOGIN sqllogin1 WITH PASSWORD = 'P@ssword'
```

次に、sqlcmd ツールで **sqllogin1** データベース ユーザーでログインしてみます。

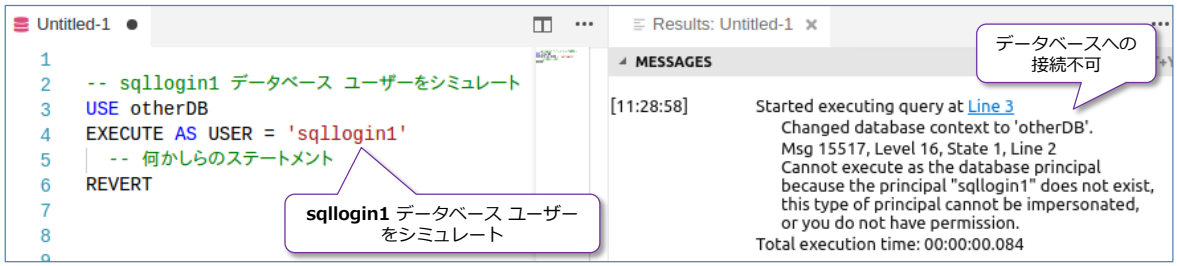
```
# sqlcmd ツールでログイン
/opt/mssql-tools/bin/sqlcmd -S localhost,1401 -U sqllogin1 -P 'P@ssword'
```

しかし、「**USE otherDB**」でリストアした **otherDB** データベースに接続しようとすると、エラーになります。**sqllogin1** データベース ユーザーは、**DROP LOGIN** で削除した **sqllogin1** ログイン アカウントに紐付いていた（マッピングしていた）ので、**CREATE LOGIN** で新しく作成した **sqllogin1** ログイン アカウントには紐付いていないために、このエラーが発生しています。

ログイン アカウントには、内部的に **SID** (Security ID) が割り当てられていて、データベース ユーザーとログイン アカウントの紐付け（マッピング）は、この SID を利用して行われているためです（新しくログイン アカウントを作成すると SID が異なるものが割り当てらることになります)。

なお、上の手順では、sqlcmd ツールを利用して、データベース ユーザーの操作を試しましたが、SQL Server では **EXECUTE** ステートメントで **AS USER** を利用することで、特定のデータベース ユーザーの動作をシミュレートする機能があります。これは、次のように利用できます。

```
-- sqllogin1 データベース ユーザーをシミュレート
USE otherDB
EXECUTE AS USER = 'sqllogin1'
  -- 何かしらのステートメント
REVERT
```



データベース ユーザーとログイン アカウントの再マッピング

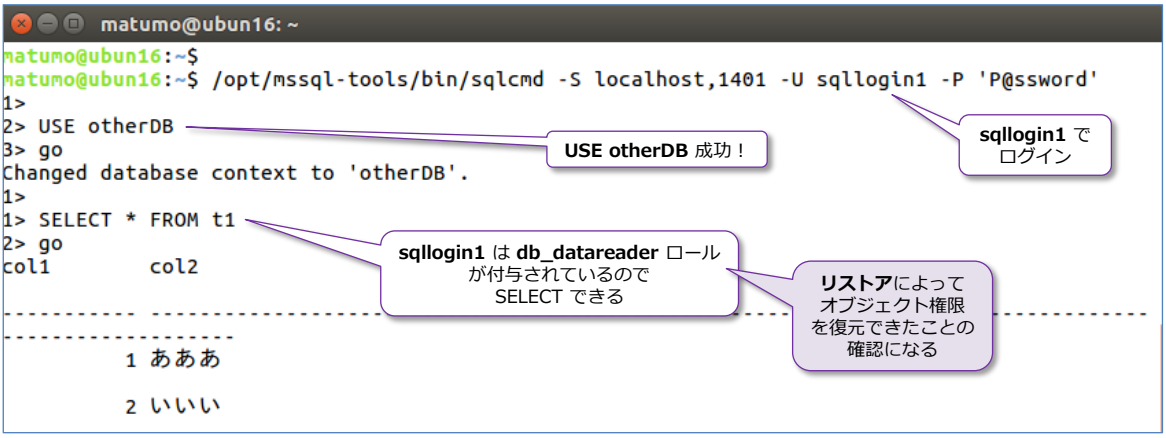
データベース ユーザーとログイン アカウントを再マッピングするには、次のように **ALTER USER** ステートメントを実行します。

```
-- マッピングを再作成
USE otherDB
ALTER USER sqllogin1
 WITH LOGIN = sqllogin1
```

otherDB データベースに接続した上で、**ALTER USER** ステートメントを実行し、**WITH LOGIN** に続けて紐付けたいログイン アカウントを指定すれば、再マッピングが完了です。

続いて、sqlcmd ツールで **sqllogin1** データベース ユーザーでログインしてみます。

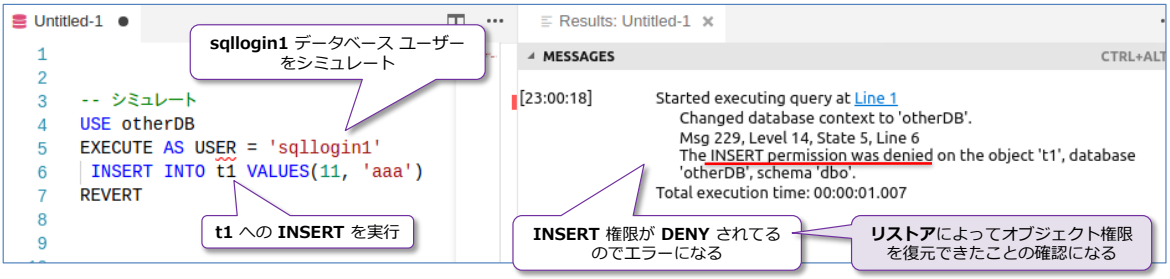
```
# sqlcmd ツールでログイン
/opt/mssql-tools/bin/sqlcmd -S localhost,1401 -U sqllogin1 -P 'P@ssword'
```



sqllogin1 データベース ユーザーは、**db_datareader** ロールが付与されているので **t1** テーブルを **SELECT** できることを確認できます。これは、**db_datareader** ロールの情報を、リストア（RESTORE DATABSE）によって復元できたことの証明にもなります。

EXECUTE AS USER でのシミュレートも次のように試せます。

```
-- シミュレート
USE otherDB
EXECUTE AS USER = 'sqllogin1'
  INSERT INTO t1 VALUES(11, 'aaa')
REVERT
```



なお、**EXECUTE AS USER** によるユーザー操作のシミュレートは、**REVERT** を実行するまでの間、有効になります。revert は「**前の状態に戻る**」という意味の英単語です。

このように、データベース ユーザーに関しては、再マッピングを行う必要がありますが、オブジェクト権限そのものは、データベースのリストアによって復元することができるので、リストア後に権限を再設定をする必要はありません。なお、Step 4.4 で説明した「**包含データベース**」を利用する場合には、そもそもログイン アカウントに紐付かないデータベース ユーザーを作成することができるので、リストア後のこういった再マッピング作業をしなくて済みます。

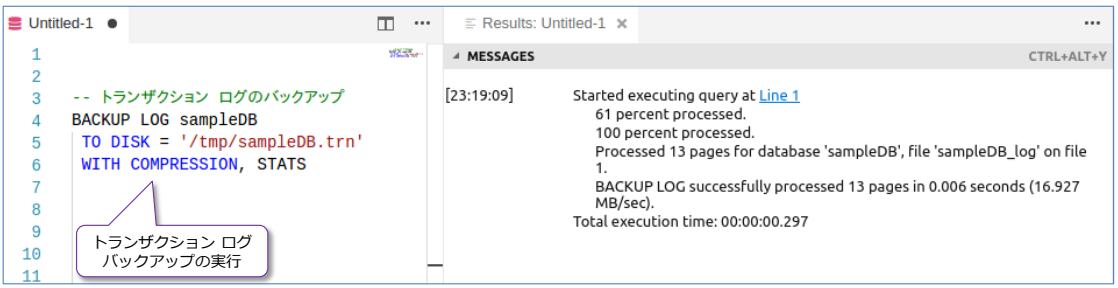
いずれにしても、SQL Server では簡単にバックアップを取得できて、簡単にリストアできることを確認できたのではないでしょうか？ リストアによって、データベース内に作成したオブジェクトはほぼ全て復元することができるので、ストアド プロシージャやビュー、インデックスなどを作成していたとしても、それらは全てバックアップ前と同じように利用することができます。

トランザクション ログのバックアップ（定期実行が必須。肥大化防止）

SQL Server を利用している上では、トランザクション ログのバックアップは欠かせません。もし、データベースに対して完全バックアップを一度も実行していない場合であれば、トランザクション ログ バックアップの定期実行は必須ではないのですが（∵チェックポイントごとにトランザクション ログが切り捨てられるため）、完全バックアップを一度でも実行している場合は、トランザクション ログは、ログ バックアップを実行しない限り**肥大化**し続けます。肥大化し続けると、ディスク領域を圧迫することになり、もしディスクの空き領域がなくなってしまった場合には、それ以上トランザクションを実行できない状態になります（トランザクション ログへの書き込みエラーが発生して、更新系のステートメントを実行できなくなります）。

トランザクション ログのバックアップは、**BACKUP LOG** ステートメントを利用しますが、データベースのバックアップで利用した **BACKUP DATABASE** と利用方法は同じで、次のように実行できます。

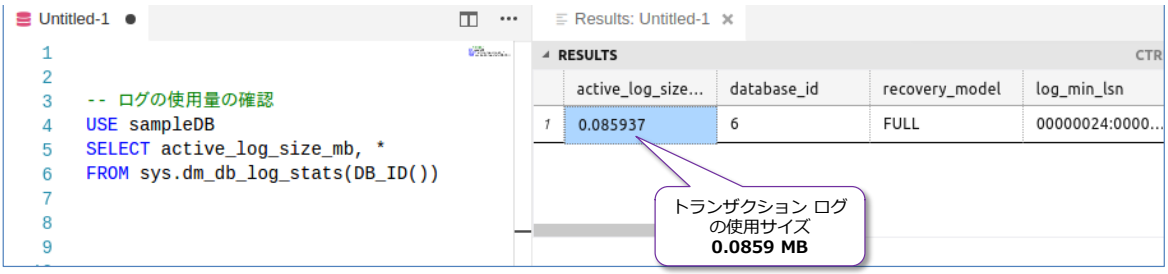
```
-- トランザクション ログのバックアップ
BACKUP LOG sampleDB
 TO DISK = '/tmp/sampleDB.trn'
 WITH COMPRESSION, STATS
```



ログ バックアップのファイル名は「**.trn**」を付ける慣習がありますが、任意の拡張子で大丈夫です。**COMPRESSION** は圧縮バックアップ、**STATS** は進行状況を表示するためのオプションでどちらも付けて実行することをお勧めします。

トランザクション ログの使用状況（どれぐらいのサイズを利用しているのか）は、**sys.dm_db_log_stats** システム ビューの **active_log_size_mb** 列を参照することで確認できます。

```
-- ログの使用量の確認
USE sampleDB
SELECT active_log_size_mb, * FROM sys.dm_db_log_stats(DB_ID())
```

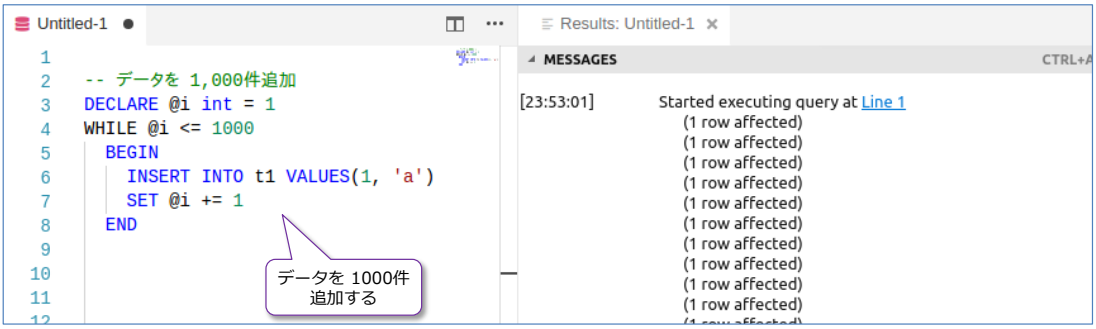


トランザクション ログの使用状況の確認

次に、トランザクションを実行して、トランザクション ログの使用量を増やしてみましょう。

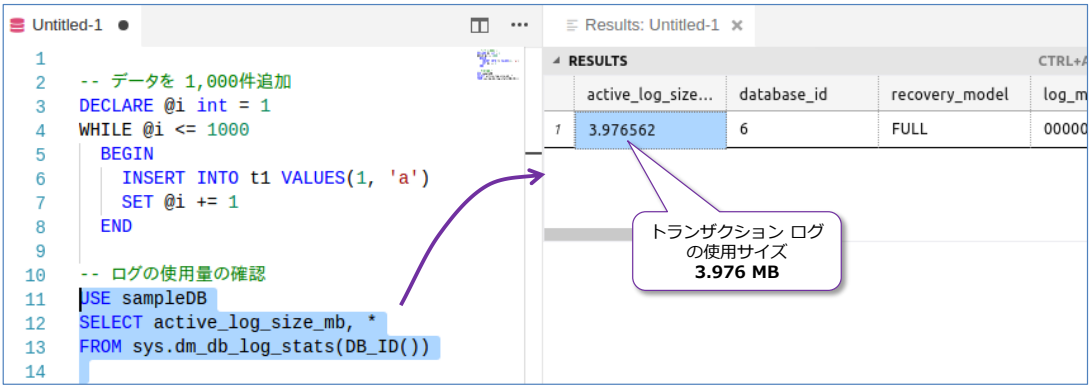
次のように **INSERT** ステートメントを実行して、データを **1000 件**追加してみます。

```
-- データを 1000件追加
DECLARE @i int = 1
WHILE @i <= 1000
  BEGIN
    INSERT INTO t1 VALUES(1, 'a')
  END
```



次に、**sys.dm_db_log_stats** システム ビューの **active_log_size_mb** 列を参照して、ログの使用量を確認します。

```
-- ログの使用量の確認
USE sampleDB
SELECT active_log_size_mb, * FROM sys.dm_db_log_stats(DB_ID())
```



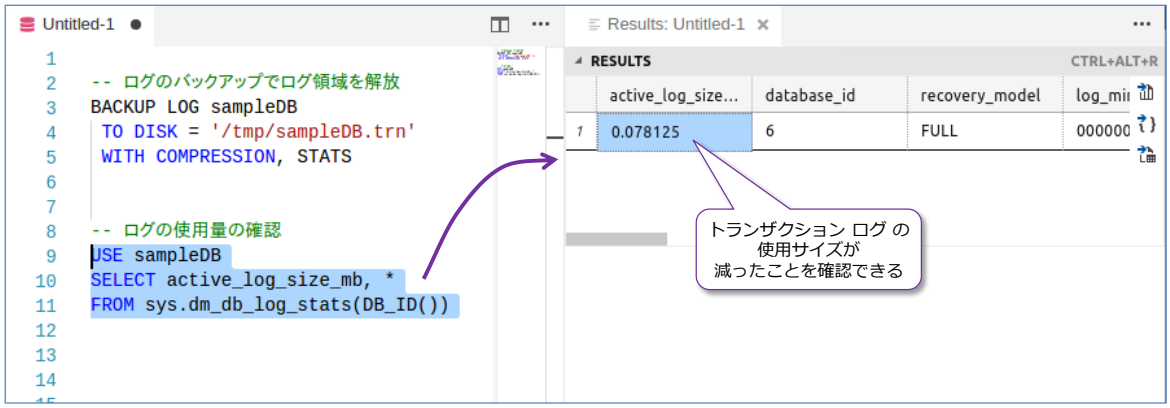
データを 1000 件追加したことで、トランザクション ログの使用量が増えたことを確認することができます。

次に、トランザクション ログのバックアップを実行して、ログ領域を解放できることを確認してみましょう。

```
-- トランザクション ログのバックアップでログ領域を解放
BACKUP LOG sampleDB
 TO DISK = '/tmp/sampleDB.trn'
 WITH COMPRESSION, STATS
```

バックアップが完了したら、再度、ログの使用量を確認してみましょう。

```
-- ログの使用量の確認
USE sampleDB
SELECT active_log_size_mb, * FROM sys.dm_db_log_stats(DB_ID())
```



トランザクション ログの使用サイズが小さくなっていることを確認できると思います。もし、サイズが小さくなっていない場合は、もう一度、ログ バックアップを実行してから、再度確認してみてください。

このように、SQL Server では、トランザクション ログのバックアップを実行することで、トランザクション ログの領域を解放して、ログの肥大化を防止することができます。データベースに対して、一度でも完全バックアップを実行した場合には、必ずトランザクション ログのバックアップを定期実行するようにしましょう。定期実行には、**cron** を利用しても良いですし、次に説明する **SQL Server Agent** 機能を利用することもできます。

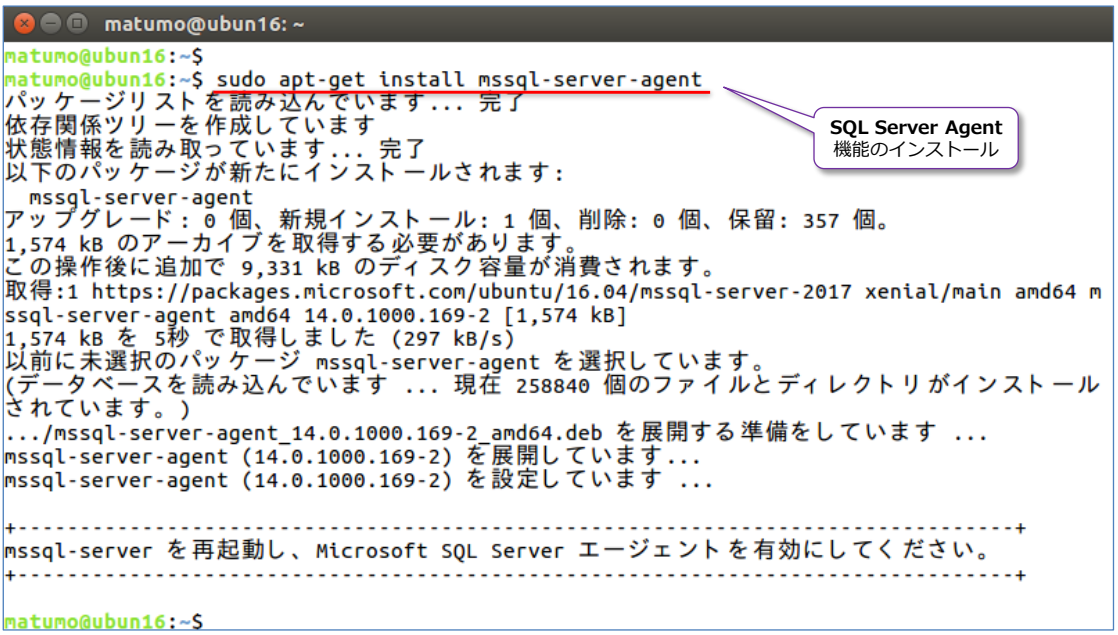
4.9 SQL Server Agent ジョブによる SQL の定期実行

SQL Server では、**SQL Server Agent** 機能を利用して、SQL ステートメントの定期実行（スケジューリング）を行うことができます。Windows 環境の場合には、SQL Server Agent は自動でインストールされますが、SQL Server on Linux の場合には、別途インストールが必要になります。

SQL Server Agent のインストール

SQL Server Agent は、次のようにインストールできます（Ubuntu の場合）。

```
# SQL Server Agent のインストール（Ubuntu の場合）
sudo apt-get update
sudo apt-get install mssql-server-agent
```



インストール後は、次のように実行して SQL Server を再起動します。

```
sudo systemctl restart mssql-server
```

RHEL（Red Hat Enterprise Linux）や **SUSE** Linux Enterprise Server の場合のインストール方法については、以下のヘルプが参考になると思います。

Create and run SQL Server Agent jobs on Linux
https://docs.microsoft.com/en-us/sql/linux/sql-server-linux-run-sql-server-agent-job

SQL Server Agent ジョブの作成

SQL Server Agent では、定期実行したいものを「**ジョブ**」という形で作成しますが、ジョブの中には、**ジョブ ステップ**（実際の実行 SQL を定義したもの）を含めて（ジョブ内に複数のステップを作成可能）、ジョブに対して**スケジュール**（どういう日時・間隔で実行するのか）を設定する形になります。

ジョブの作成は、**msdb** データベースに **USE** した状態で行う必要があり、**sp_add_job** でジョブの作成、**sp_add_jobstep** でジョブ ステップの追加、**sp_add_schedule** でスケジュールの追加、**sp_attach_schedule** でスケジュールをジョブに紐付け、**sp_add_jobserver** でジョブを SQL Server に登録、**sp_start_job** でジョブを開始するという流れになります。具体的には、次のように記述できます。

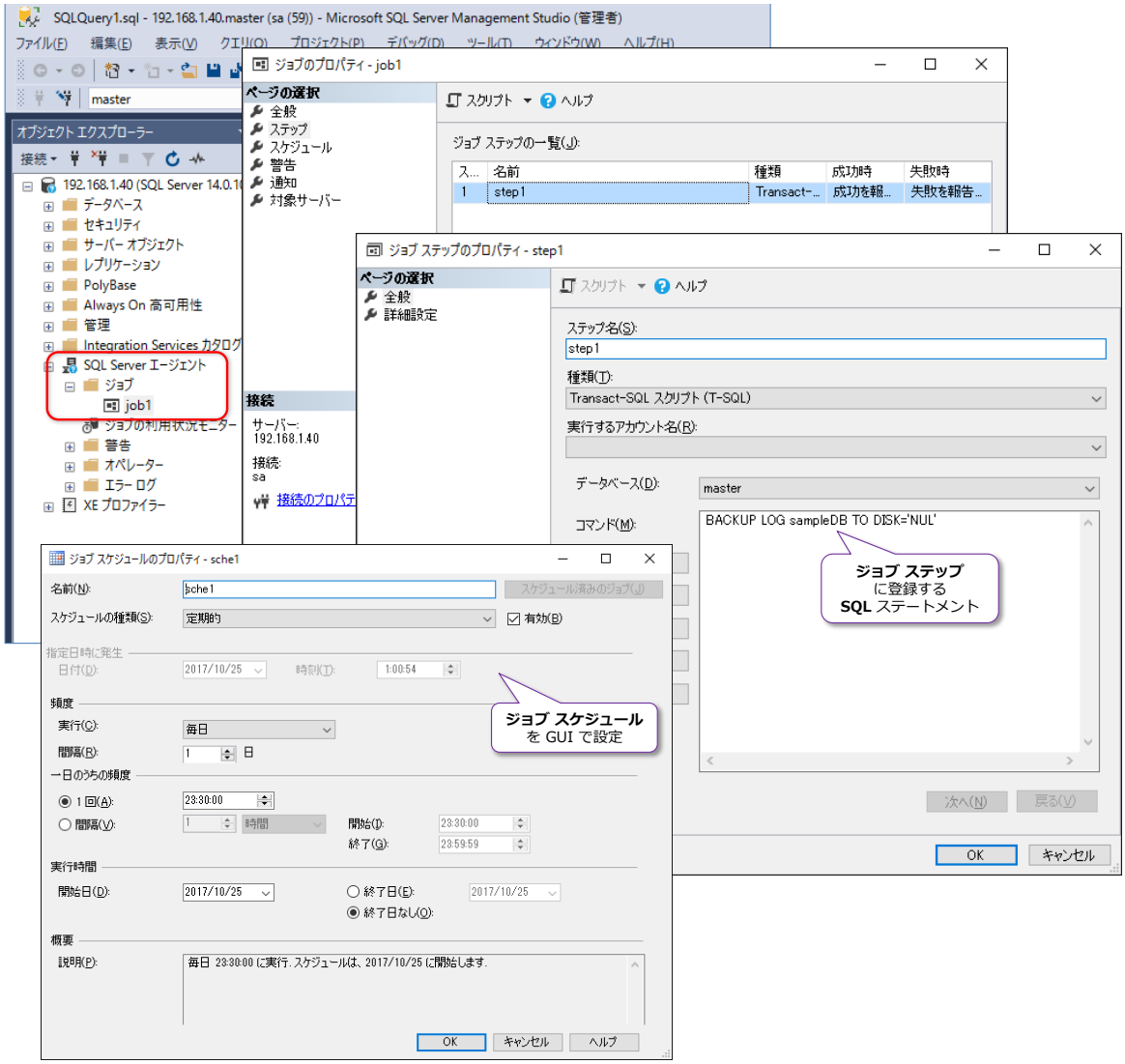
```
-- ジョブの作成は msdb システム DB に USE して行う
USE msdb

-- ジョブの作成
EXEC dbo.sp_add_job @job_name = N'job1'

-- ジョブ ステップの追加。ログのバックアップを定期実行したい場合
EXEC sp_add_jobstep
   @job_name = N'job1',
   @step_name = N'step1',
   @subsystem = N'TSQL',
   @command = N'BACKUP LOG sampleDB TO DISK=''NUL'' ',
   @retry_attempts = 5,
   @retry_interval = 5

-- 毎日のスケジュール作成. 毎日 23:30 分に実行
EXEC dbo.sp_add_schedule
   @schedule_name = N'sche1',
   @freq_type = 4,
   @freq_interval = 1,
   @active_start_time = 233000

-- スケジュールのアタッチ
EXEC sp_attach_schedule
   @job_name = N'job1',
   @schedule_name = N'sche1'

-- ジョブを SQL Server に登録
EXEC dbo.sp_add_jobserver
   @job_name = N'job1',
   @server_name = N'(LOCAL)'

-- ジョブの開始（ジョブが正常に動作するか確認するためのテスト実行）
EXEC dbo.sp_start_job N'job1'
```

このように、ジョブを作成すれば、毎日 23:00 にトランザクション ログのバックアップを実行することができます。ログのバックアップ先として **TO DISK = 'NUL'** と指定している部分は、

NULL デバイスへのバックアップ（ダミー バックアップ）と呼ばれていて、バックアップをファイルに保存することなく、バックアップ操作のみを実行するという役割があります。これによって、トランザクション ログのバックアップが実行できるので、ログの領域が解放できる（肥大化防止になる）、という形です。

もちろん、ログのバックアップは、万が一の障害発生時には、ログ バックアップを元に、できる限り障害が発生する直前の状態まで復旧するために必要なファイルになるので、通常は NULL デバイスではなく、ファイルとして保存して、万が一の事態に備えておくことが重要になります。

なお、Windows 環境の Management Studio から on Linux をリモート操作できる場合には、次のように GUI を利用して、ジョブを簡単に作成することができます。



SQL Server on Linux では、**データベース メール**機能を利用することもできます。これを SQL Server Agent の既定のメール機能に設定することもできるので、Agent オペレーターのメール通知として利用すれば、ジョブの成功や失敗をデータベース メールで送信する、といったことも簡単に行えます。これについては、以下の URL が参考になります。

https://docs.microsoft.com/en-us/sql/linux/sql-server-linux-db-mail-sql-agent

4.10 その他 SQL Server を利用する上での考慮事項

その他、SQL Server を利用する上での考慮事項は、次のとおりです。

- **PRIMARY KEY 制約**を作成すると**クラスター化インデックス**が自動作成される
  SQL Server ではクラスター化インデックスが基本のインデックスになり、
  これは、データ構造そのものをインデックス キーで並べ替えた構成になる
  （Oracle での索引構成表に相当）
- **UNIQUE 制約**または **UNIQUE インデックス**では **NULL** は 1 つだけ許可される
  （Oracle の場合は複数値 OK）
- **トランザクション**の開始には **BEGIN TARNSACTION** が必要
- **デッドロック**の発生時は、どちらかのトランザクションが丸ごとロールバックされる
- **ロック**や**読み取り一貫性**に関して
  SQL Server ではデータの読み取り時に共有ロックを取得するため、読み取り操作は排他ロックにブロックされる。これを回避するには NOLOCK ヒントや、読み取り一貫性機能（READ_COMMITTED_SNAPSHOT またはスナップショット分離レベル）を利用する。
  ロック エスカレーションの禁止も可能（テーブルごとに設定できる）。
  Oracle での FOR UPDATE NOWAIT は、WITH(UPDLOCK, NOWAIT) で利用可能
- **ROW_NUMBER** で SELECT 結果に対して行番号を取得（Oracle での ROWNUM）

既に説明したものには、以下のものがあります。

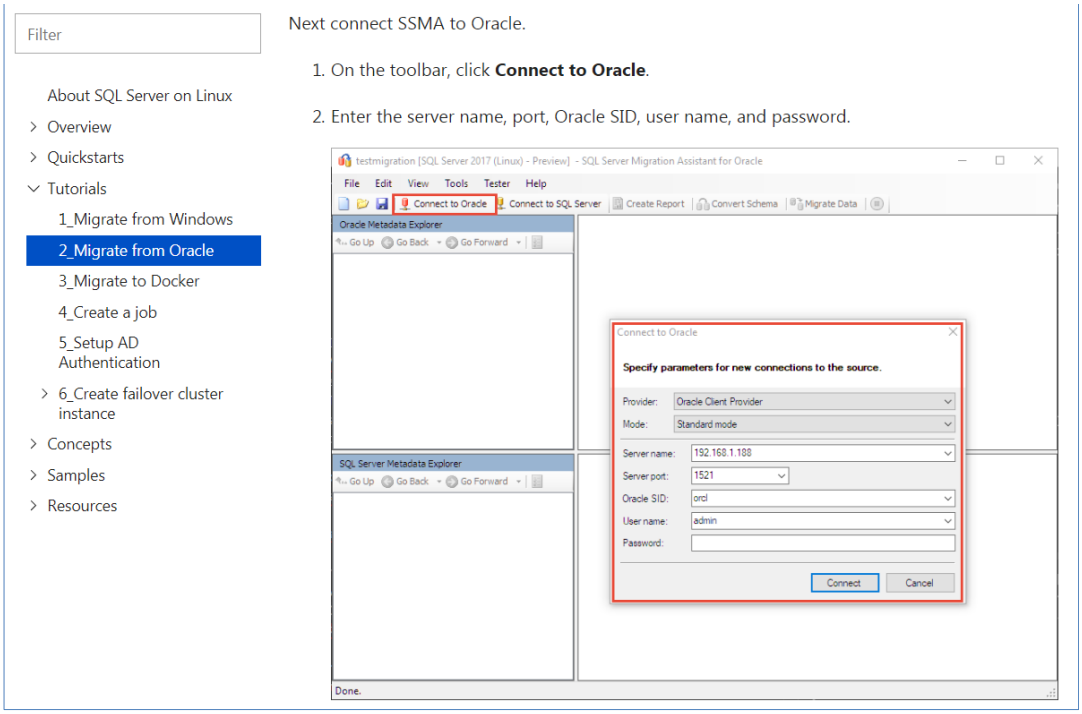
- **トランザクションのログの肥大化**防止にログ バックアップを定期実行する
  定期実行には SQL Server Agent ジョブ機能を利用できる
  なお、もし完全バックアップを一度も取得していない場合は、ログは肥大化しない
- **IDENTITY** による連番作成
  Oracle 同様、SEQUENCE もあり、同じように利用できるが、IDENTIY のほうが気軽に連番を作成できるので、SQL Server ユーザーはよく利用している
- **SELECT INTO** での新しいテーブル作成
  Oracle での CREATE TABLE AS SELECT に相当する機能
- **リンク サーバー**でリモート サーバーへの接続が可能
- **bcp** コマンドで CSV ファイルのインポート／エクスポートが可能
- 管理者アカウントは **sa** と **sysadmin** ロール。既定のスキーマ名は **dbo** になる
- **データベース ユーザー**はログイン アカウントの SID と紐尽く
  バックアップからリストアするときは、SID の再マッピングに注意。
  あるいは包含データベース機能を利用することで、ログイン アカウントに紐付かないデータベース ユーザーを作成することができる
- **大文字と小文字の区別**、かなとカナの区別、半角と全角の区別には**照合順序**が関係

照合順序は列レベルで変更できる。既定値は Japanese_CI_AS で上記はすべて区別しない。Japanese_BIN2 を利用することで、すべての文字をバイナリ レベルで区別可能

その他の参考情報 ～Oracle からの移行～

Oracle から SQL Server に移行する場合には、Oracle からの移行が簡単に行える **SSMA for Oracle**（SQL Server Migration Assistant）という便利な**移行支援ツール**が提供されています。SSMA や、Oracle からの移行に関しては、オンライン ブック（SQL Server のヘルプ）の以下のトピックが参考になります。

Migrate an Oracle schema to SQL Server 2017 on Linux with the SQL Server Migration Assistant
https://docs.microsoft.com/en-us/sql/ssma/oracle/sql-server-linux-convert-from-oracle?toc=%2fsql%2flinux%2ftoc.json



Windows 環境から Linux 環境への移行に関して

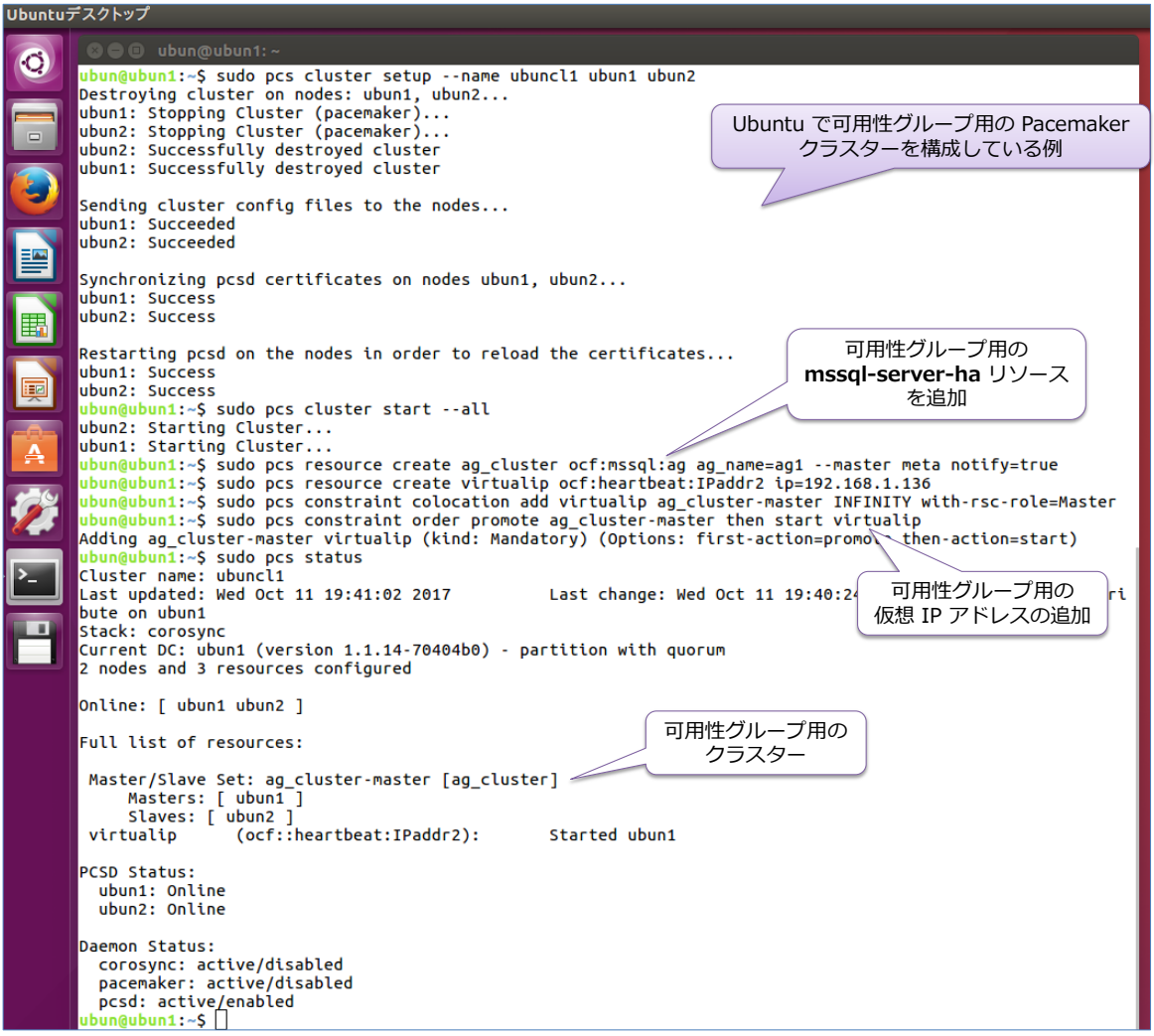
Windows 上の SQL Server 環境で取得したバックアップを、Linux 環境にリストアすることも、何の問題もなく行えるので、**移行**（マイグレーション）も簡単に行うことができます（リストア時の考慮事項は、Windows 環境でのリストアの場合と全く同様です）。

既存の Windows 環境に対して、Linux を含めた可用性グループを構成すれば、まったく同じデータベース（ミラー化した複製データベース）を Linux 上に作成することができるので、段間的な移行用途として可用性グループを利用することもできます。

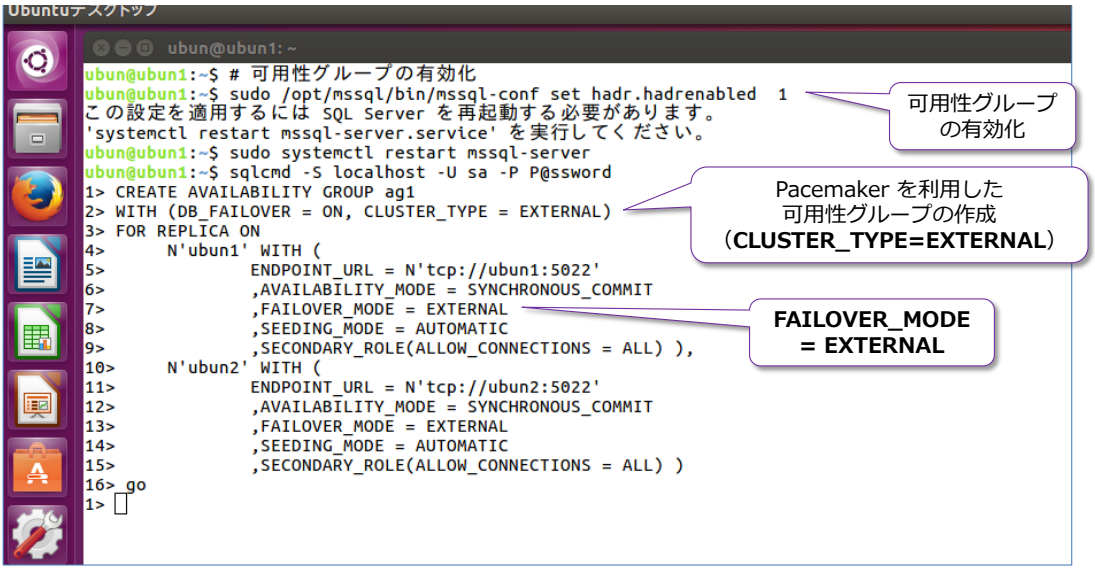
4.11 AlwaysOn 可用性グループによる冗長構成

SQL Server 2017 on Linux では、**AlwaysOn 可用性グループ**も利用することができます。Windows 環境の場合は、WSFC（Windows Server フェールオーバー クラスター）を利用して可用性グループを構成しますが、Linux 環境の場合は、**Pacemaker** を利用します。

Ubuntu で可用性グループ用の Pacemaker クラスターを作成している例



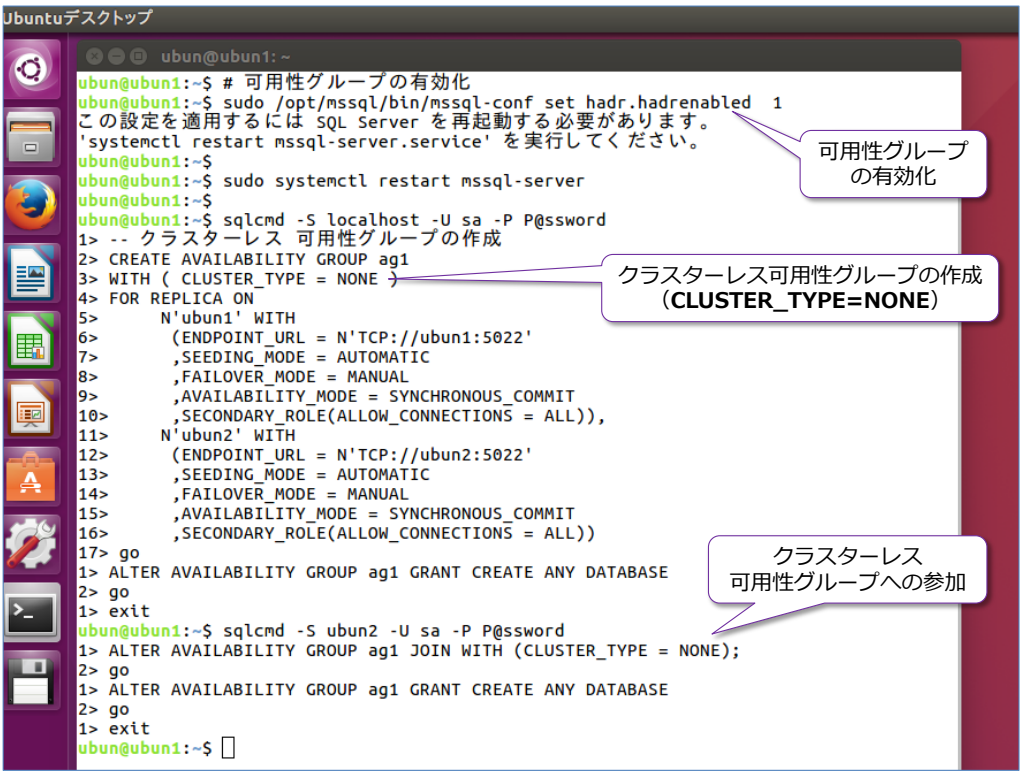
可用性グループを作成するときは、次のように「**CLUSTER_TYPE=EXTERNAL**」と指定することで Pacemaker を利用した可用性グループを利用できるようになります。



クラスター レス可用性グループ（読み取りスケール アウト構成）

可用性グループでは、**クラスター レス**（クラスター不要）の構成もサポートして、この場合は Pacemaker は必要ありません。ただし、これは高可用性が目的のものではなく、**読み取り専用スケール**（読み取り性能を向上させる目的）として可用性グループを利用することになります。

Ubuntu でクラスター レス可用性グループを作成している例



可用性グループは、Windows と Linux にまたがって構成することもできます。また、SQL Server 2016 からの新機能である**分散可用性グループ**（Distributed Availability Group）に、Windows と Linux の可用性グループを含めることもできます。

既存の Windows 環境に対して、Linux を含めた可用性グループを構成すれば、まったく同じデータベース（ミラー化した複製データベース）を Linux 上に作成することができるので、段間的な**移行**（マイグレーション）用途として可用性グループを利用することもできます。

可用性グループの参考情報

可用性グループの詳細については、オンライン ブック（SQL Server のヘルプ）の以下のトピックが参考になります。

Linux 上の SQL Server の Always On 可用性グループを構成
https://docs.microsoft.com/ja-jp/sql/linux/sql-server-linux-availability-group-configure-ha

Ubuntu クラスターと可用性グループ リソースを構成
https://docs.microsoft.com/ja-jp/sql/linux/sql-server-linux-availability-group-cluster-ubuntu

SQL Server 可用性グループに RHEL クラスターを構成
https://docs.microsoft.com/ja-jp/sql/linux/sql-server-linux-availability-group-cluster-rhel

SQL Server 可用性グループに SUES クラスターを構成
https://docs.microsoft.com/ja-jp/sql/linux/sql-server-linux-availability-group-cluster-sles

SQL Server on Linux の読み取りのスケール アウト可用性グループを構成
https://docs.microsoft.com/ja-jp/sql/linux/sql-server-linux-availability-group-configure-rs

なお、SQL Server on Linux では、ログ配布（Log Shipping）を利用することもできます。これについては、以下のトピックが参考になります。

Get started with Log Shipping on Linux
https://docs.microsoft.com/en-us/sql/linux/sql-server-linux-use-log-shipping

4.12 トレースフラグやメモリ使用量の制限設定（mssql.conf）

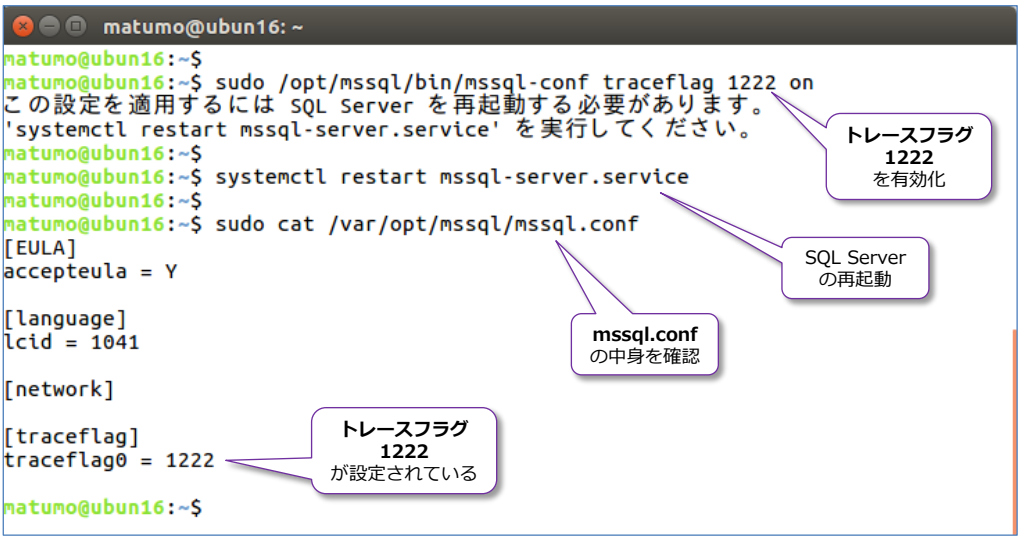
SQL Server 2017 on Linux では、一部の管理設定は、「**mssql.conf**」という構成ファイルを利用しています（**/var/opt/mssql** ディレクトリに存在）。

Windows 版の SQL Server では、**トレースフラグの設定**（SQL Server の起動時にトレース フラグを設定したい場合）は、「**SQL Server 構成マネージャー**」ツールを利用して、起動オプションを変更していましたが、SQL Server on Linux での **mssql.conf** はその代わりになるようなものです。

トレースフラグの設定

トレースフラグは、次のように設定できます。

```
# トレースフラグ 1222 を有効化する場合の例
sudo /opt/mssql/bin/mssql-conf traceflag 1222 on

# SQL Server の再起動
systemctl restart mssql-server.service

# 設定内容の確認
sudo cat /var/opt/mssql/mssql.conf
```

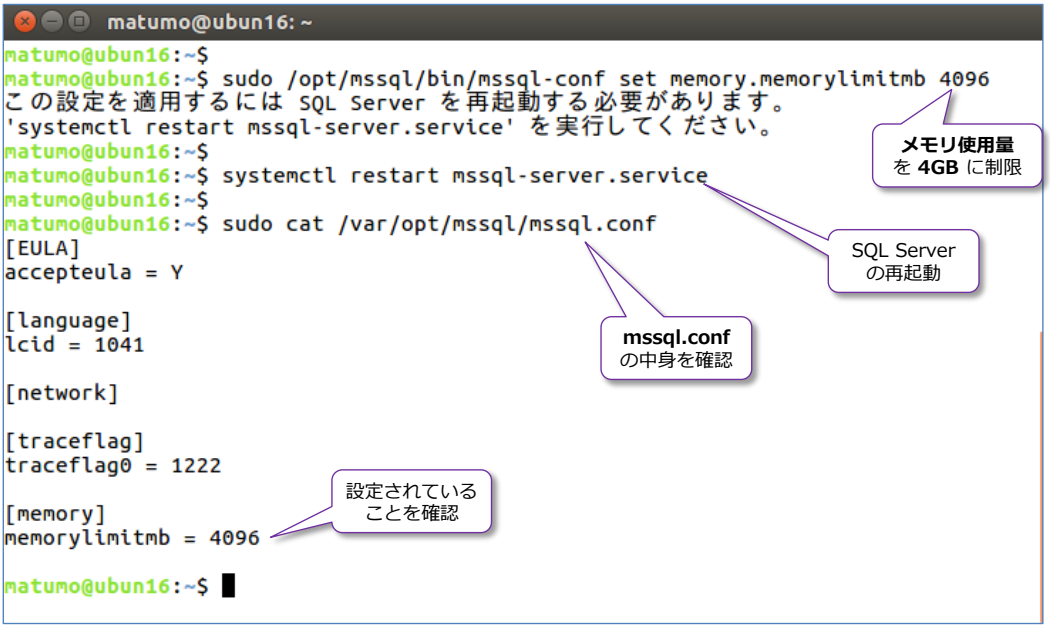


メモリ使用量の制限

SQL Server on Linux でのメモリ使用量の制限は、次のように設定できます。

```
# メモリ使用量を 4GB に制限する場合の例
sudo /opt/mssql/bin/mssql-conf set memory.memorylimitmb 4096
```

```
# SQL Server の再起動、設定内容の確認
systemctl restart mssql-server.service
sudo cat /var/opt/mssql/mssql.conf
```



その他の mssql.conf の設定

「**mssql.conf**」構成ファイルでは、既定のディレクトリ（データベースやバックアップが作成される場所の既定値）や、**TLS** の設定（後述のネットワーク接続の暗号化設定）、可用性グループの有効化などを設定できますが、これらについては、オンライン ブック（SQL Server のヘルプ）の以下のトピックが参考になります。

Mssql conf ツールを使用して Linux 上の SQL Server を構成
https://docs.microsoft.com/ja-jp/sql/linux/sql-server-linux-configure-mssql-conf

STEP 5. SQL Server on Linux でのセキュリティ強化

この STEP では、「**動的データ マスク**」や「**行レベル セキュリティ**」、「**バックアップ暗号化**」、「**TDE**」、「**監査**」、「**Always Encrypted**」など、SQL Server on Linux で利用できるセキュリティ機能について説明します。

この STEP では、次のことを学習します。

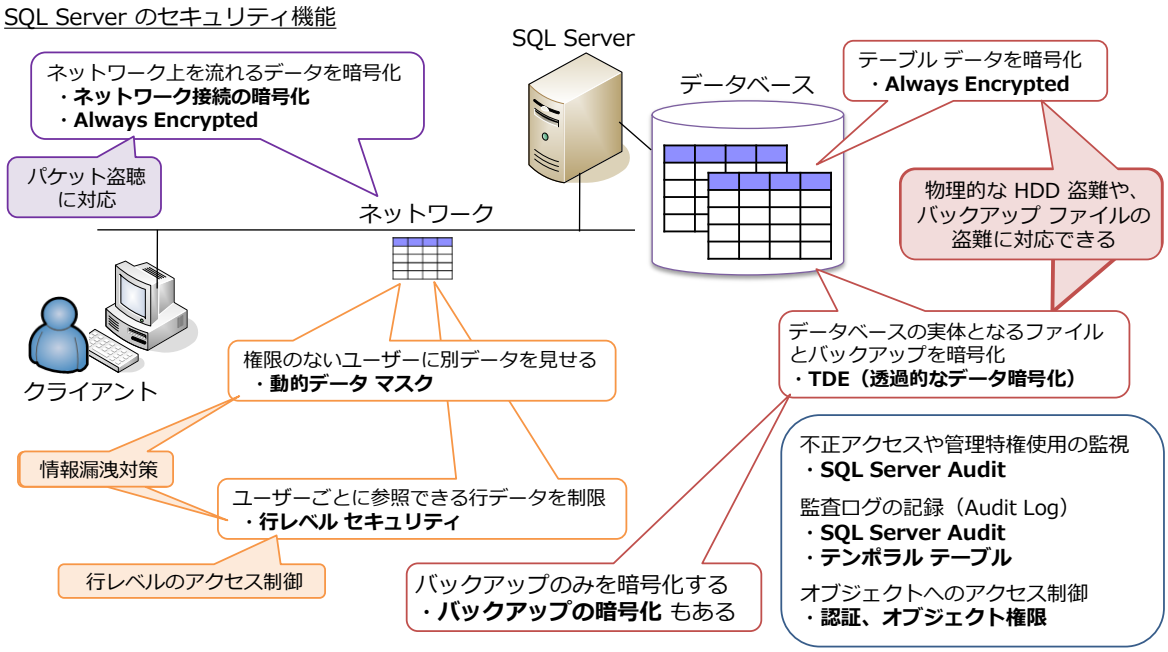
- ✓ 動的データ マスクによる情報漏洩対策
- ✓ 行レベル セキュリティによるセキュリティ強化
- ✓ バックアップ ファイルの暗号化
- ✓ ネットワーク接続の暗号化（パケット盗聴対策）
- ✓ TDE（透過的なデータ暗号化）による DB 暗号化
- ✓ 監査（SQL Server Audit）による操作履歴の記録
- ✓ Always Encrypted（常に暗号化）

5.1 SQL Server 2017 on Linux のセキュリティ

SQL Server 2017 on Linux では、SQL Server で利用できる**標準のセキュリティ機能**をそのまま利用することができます。その主なものは、次のとおりです。

- **動的データ マスク**（Dynamic Data Masking）
- **行レベル セキュリティ**（Row Level Security）
- **バックアップの暗号化**（Backup Encryption）
- **ネットワーク接続の暗号化**（Encrypting Connection）
- **TDE（透過的なデータ暗号化）**、Enterprise エディションでのみ利用可能
- **SQL Server Audit（監査）**
- **Always Encrypted による列データの暗号化**
- **テンポラル テーブル**
- **認証、オブジェクト権限**（第 4 章参照）

これらの機能は、次のような目的で利用できます。



情報漏洩対策として「**動的データ マスク**」や「**行レベル セキュリティ**」、ネットワークのパケット盗聴対策として「**ネットワーク接続の暗号化**」や「**Always Encrypted**」、HDD などのハードウェアの物理的な盗難への対策として「**TDE**」や「**バックアップの暗号化**」、不正アクセス対策や監査ログの記録（Audit Trail）として「**SQL Server Audit**」や「**テンポラリ テーブル**」、各種のオブジェクトへのアクセス制御として「**認証、オブジェクト権限**」といった形で、昨今求められているセキュリティ要件は、これらを利用することで簡単に満たすことができます。

なお、これらの機能のうち、TDE（透過的なデータ暗号化）の利用には Enterprise エディションが必要になりますが、その他の機能は Standard エディションでも利用することができます。どの

機能がどのエディションで利用できるかどうかについては、以下の URL が参考になります。

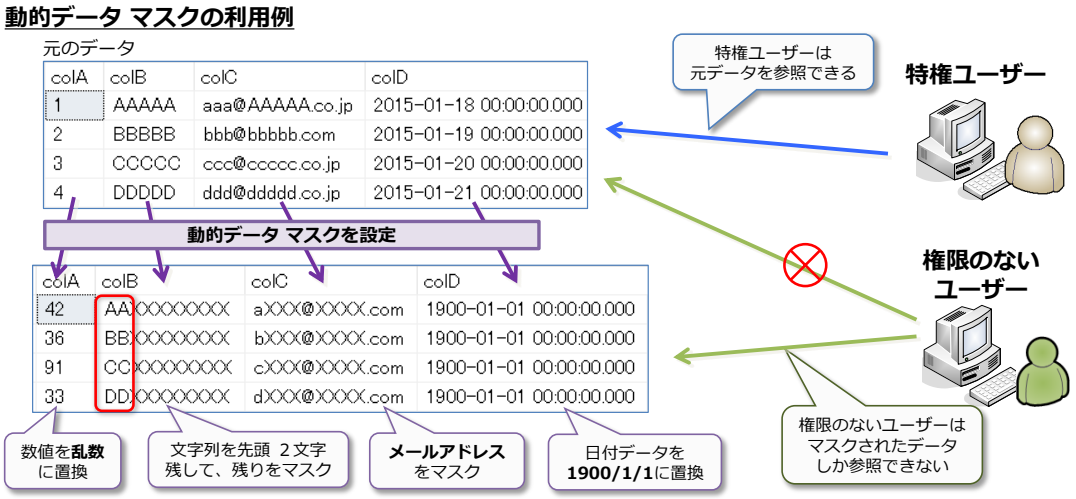
エディションと SQL Server 2017 on Linux のサポートされる機能
https://docs.microsoft.com/ja-jp/sql/linux/sql-server-linux-editions-and-components-2017

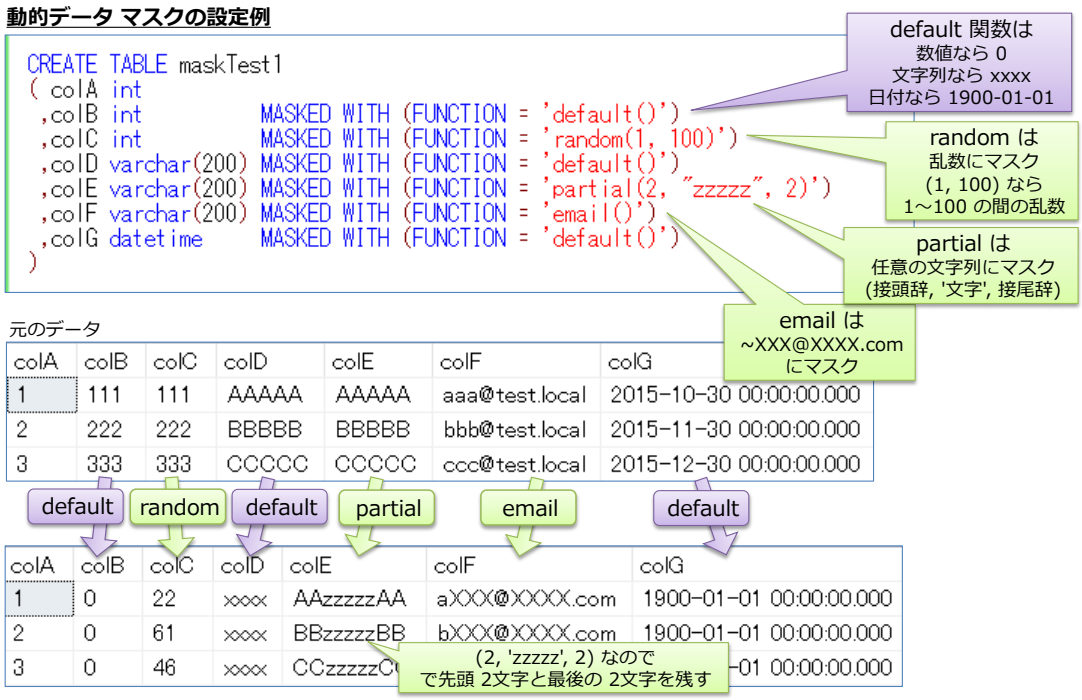
セキュリティ機能の概要

それぞれのセキュリティ機能の概要は、次のとおりです。

動的データ マスク（Dynamic Data Masking）

動的データ マスクは、顧客情報（クレジット カード番号やマイナンバーなど）や**機密情報をマスク**（別の値に置換）して、**情報漏洩**を防止できる機能です。





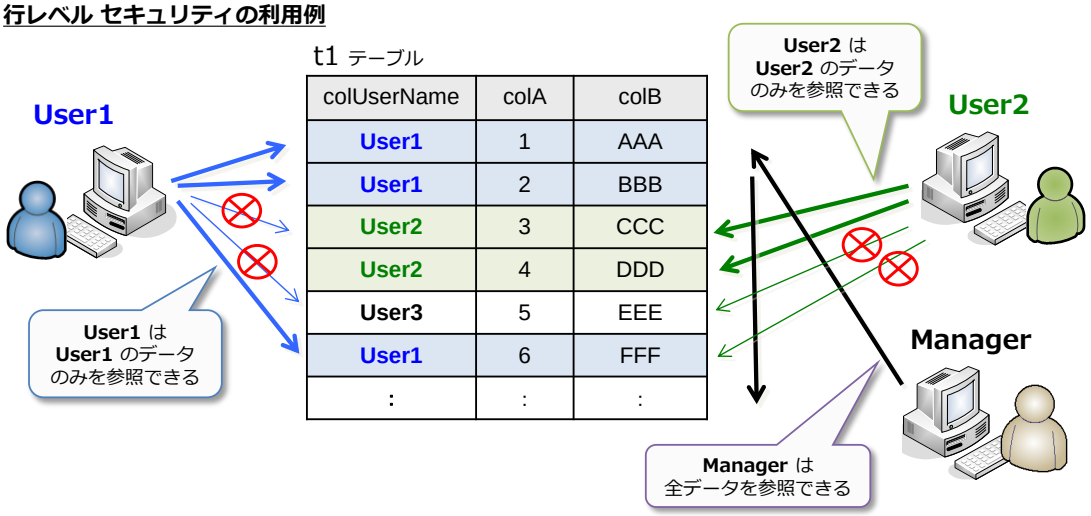
動的データ マスクでは、default 関数や random 関数、email 関数などを利用して、列データを

マスクすることができます。

行レベル セキュリティ（Row Level Security）

行レベル セキュリティは、**行レベルのアクセス制御**を実現できる機能で、ユーザーごとに、参照できる行データを制限することができます。

行レベル セキュリティの利用例



バックアップの暗号化（Backup Encryption）

バックアップの暗号化は、バックアップ データを暗号化できるので、バックアップ ファイルの盗難への対策になります。

```
BACKUP DATABASE Northwind
TO DISK = N'/tmp/North_enc.bak'
WITH
  ENCRYPTION
   ( ALGORITHM = AES_256,
     SERVER CERTIFICATE = MyTestDBBackupEncryptCert    )
go
```

バックアップ ファイルを暗号化できる

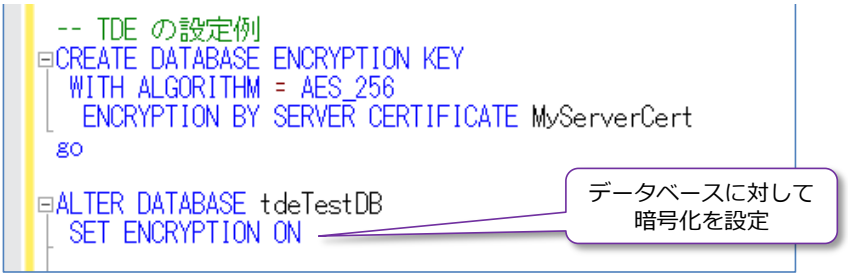
ネットワーク接続の暗号化（Network Encryption）

ネットワーク接続の暗号化は、ネットワーク上を流れるパケットを暗号化できるので、パケット盗聴への対策になります。SQL Server 2017 でネットワーク接続を有効化しておくと、クライアント（アプリケーション）からは、接続文字列に以下を追加することで暗号化接続を指定できるようになります。

```
JDBC
"encrypt=true; trustServerCertificate=false;"
ODBC
"Encrypt=Yes; TrustServerCertificate=no;"
ADO.NET
"Encrypt=True; TrustServerCertificate=False;"
```

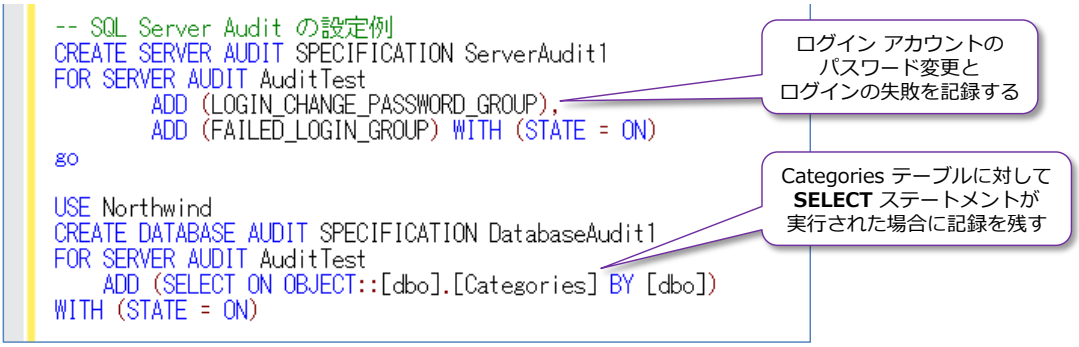
TDE（透過的なデータ暗号化）

TDE は、データベースを丸ごと暗号化することができるので、HDD などのハードウェアの物理的な盗難への対策になります。TDE を設定したデータベースは、バックアップ ファイルも暗号化されるので、前掲のバックアップの暗号化を利用しなくても、バックアップ データを暗号化して保護することができます。



SQL Server Audit（監査）

SQL Server Audit では、ユーザーの操作をすべて記録することができるので、**Audit Trail**（監査証跡）や、各種の**法令遵守**に利用できます（J-SOX や内部統制、PCI DSS など、コンプライアンスを実現するために欠かせない機能になります）。



Always Encrypted による列データの暗号化

Always Encrypted は、ネットワーク上を流れるデータも、データベース内に格納されるデータも、**すべて暗号化**して格納できる機能です（列データを暗号化して、アプリケーションも透過的に利用することができます）。これを利用している場合は、TDE を利用していなくても、データベース（の列データが暗号化されているので）物理的な盗難防止になります。また、前掲のネットワーク接続の暗号化を利用していなくても、ネットワーク上のデータを暗号化することができます。

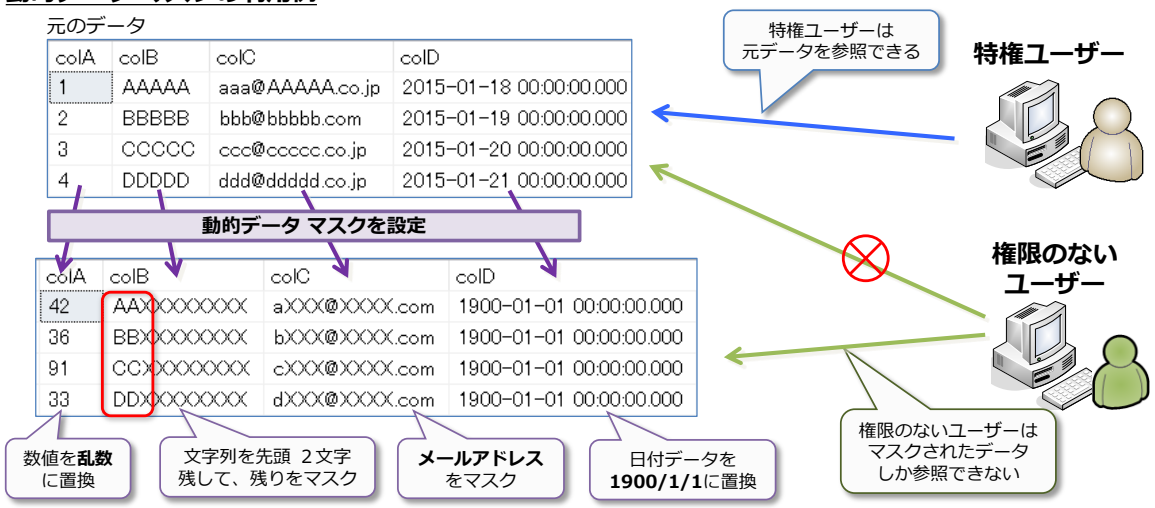
このように、SQL Server には、セキュリティを強化できる機能が非常に豊富に用意されているので、セキュリティ要件を簡単に満たすことができます。

以降では、これらの機能について、ステップ バイ ステップ形式で １つ １つ具体的に説明していきます。

5.2 動的データ マスクによる情報漏洩対策

動的データ マスク（Dynamic Data Masking）は、**クレジット カード番号**や**マイナンバー**、**メール アドレス**などの**顧客情報／機密情報をマスク**（異なる値に置換）して、セキュリティ強化（情報漏洩の防止）を実現できる機能です。これは、次のように利用することができます。

動的データ マスクの利用例



このように、動的データ マスクでは、権限のないユーザーに対して、重要な情報を参照できないようにする（マスクされたデータしか参照できないようにする）ことができるので、顧客情報や機密情報（クレジット カード番号やマイナンバーなど）の**データ流出を抑える**ことができるようになります。

動的データ マスクの設定方法（MASKED WITH）

動的データ マスクは、次のように **CREATE TABLE** ステートメントでのテーブル作成時または **ALTER TABLE** ステートメントでのテーブル変更時に、列に対して設定することができます。

```
-- 動的データ マスクの設定例
CREATE TABLE maskTest1
( colA int
 ,colB varchar(200) MASKED WITH (FUNCTION = 'default()')
)
```

列名、データ型に続けて、**MASKED WITH** を付けて、**FUNCTION='～'** でどのようにマスクするのかを指定します。上の例では「**default**」という関数を利用していますが、この場合は、文字列データ（**char** や **varchar** データ型）の場合に、「**xxxx**」という値にマスク（置換）することができます。**default** 関数は、数値データ（**int** データ型など）の場合には「**0**」、日付データ（**datetime** データ型など）の場合には「**1900-01-01 00:00:00.000**」という値にマスクすることができます。**FUNCTION** では、**default** 関数の他に、数値を乱数化する「**random**」や、電子メール アドレスをマスクできる「**email**」、任意の文字列にマスクすることができる「**partial**」などを利用することができます。

Let's Try

それでは、これを試してみましょう。

1. まずは、動的データ マスクを試すためのデータベースとユーザーを作成します。

```
-- データベース「maskTestDB」の作成
CREATE DATABASE maskTestDB
go
-- データベース ユーザー「UserX」の作成
USE maskTestDB
CREATE USER UserX WITHOUT LOGIN
```

データベース名は「**maskTestDB**」、データベース ユーザーは「**UserX**」という名前で作成しますが、データベース ユーザーは、**WITHOUT LOGIN** を付けることで、ログイン アカウントとは紐付かない**簡易的なユーザー**にしています。このように作成したユーザーは、ログインするためのユーザーとしては利用することができないので、現実的な利用方法ではないのですが、セキュリティ機能を試すときにのみ便利なユーザーになります（このユーザーは、ログインをすることはできませんが、後述の **EXECUTE AS USER** ステートメントを利用して、接続をシミュレートして利用することができます。セキュリティ機能では、複数のユーザーでの動作を試したい場面が多々あるので、こういった場合に役立つユーザーの作成方法です）。

2. 次に、**CREATE TABLE** ステートメントを利用してテーブルを作成します（テーブル名は **maskTest1** として、**colA**、**colB**、**colC**、**colD**、**colE** の 5 つの列を作成）。このときに、**MASKED WITH** を付けて、それぞれの列に動的データ マスクを設定します。

```
-- 動的データ マスクを設定
USE maskTestDB
CREATE TABLE maskTest1
( colA int          MASKED WITH (FUNCTION='random(1, 100)')
 ,colB varchar(10) MASKED WITH (FUNCTION='default()')
 ,colC varchar(10) MASKED WITH (FUNCTION='partial(2, "zzz", 2)')
 ,colD varchar(20) MASKED WITH (FUNCTION='email()')
 ,colE datetime    MASKED WITH (FUNCTION='default()') )
```

colA（**int** データ型）は、**FUNCTION='random(1,100)'** と指定することで、1～100 の間の乱数にマスクすることができます。**colB** と **colC** は **varchar(10)** データ型で、マスクには **default()** と **partial(2, "zzz", 2)** を利用しています（この 2 つの列には、次の手順で同じ値を格納しますが、どういった違いが出るかは、そのときに確認します）。

colD には **email()**、**colE**（**datetime** データ型）には **default()** マスクを設定しています。

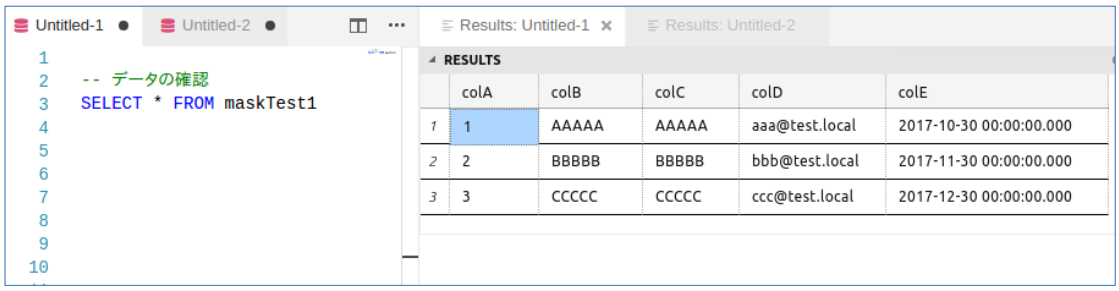
3. 次に、データを 3 件追加します（データは適当なものでかまいません）。

```
-- データを追加
INSERT INTO maskTest1
 VALUES (1, 'AAAAA', 'AAAAA', 'aaa@test.local', '2017/10/30')
```

```
        ,(2, 'BBBBB', 'BBBBB', 'bbb@test.local', '2017/11/30')
        ,(3, 'CCCCC', 'CCCCC', 'ccc@test.local', '2017/12/30')
```

4. 続いて、追加したデータを確認しておきます。

```
-- データの確認
SELECT * FROM maskTest1
```



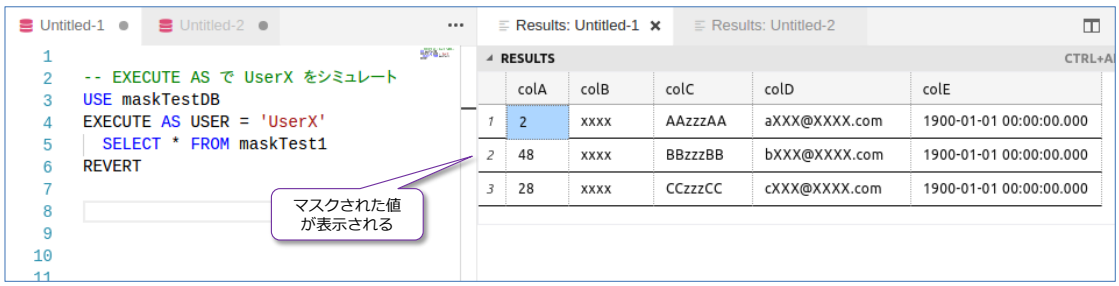
動的データ マスクでは、特権ユーザー（**sa** などの管理者アカウント）に関しては、通常どおりにデータを参照することができます。

5. 次に、一般ユーザーからのデータ アクセスを試すために、最初に作成したユーザー「**UserX**」に対して **SELECT** 権限を付与します。

```
-- データベース ユーザーに、テーブルに対する SELECT 権限を付与
GRANT SELECT ON maskTest1 TO UserX
```

6. 次に、**EXECUTE AS** ステートメントを利用して、「**UserX**」でログインしたときの動作をシミュレートします（**EXECUTE AS** ステートメントは、**USER=** で指定したユーザーでログインしたときをシミュレートすることができ、**REVERT** ステートメントを実行するまでの間、そのユーザーでのログインをシミュレートできます）。

```
-- EXECUTE AS で UserX をシミュレート
EXECUTE AS USER = 'UserX'
  SELECT * FROM maskTest1
REVERT
```



このように、一般ユーザーがテーブルにアクセスした場合は、動的データ マスクが有効になって、データがマスクされていることを確認できます。**colA** は **1～100** の間の乱数、**colB** は「**xxxx**」、**colC** は「**AAzzzAA**」や「**BBzzzBB**」にマスクされていることを確認できます。

colB は **default()** 関数を設定したので「**xxxx**」固定になりますが、**colC** には「**partial(2, "zzz", 2)**」のように **partial** 関数を利用しています。この関数は、3 つの引数を「**接頭辞, padding, 接尾辞**」という形で利用します。今回のように「**(2, "zzz", 2)**」と指定した場合は、接頭辞が **2**、接尾辞も **2** なので、先頭と最後の 2 文字はそのまま残して、間の文字を **zzz** で埋めて（パディングして）マスクすることができます。

colD には **email()** 関数を設定したので、「**aaa@test.local**」は「**aXXX@XXXX.com**」、「**bbb@test.local**」は「**bXXX@XXXX.com**」という値に変換されていることを確認できます。**email()** 関数では、先頭の 1 文字を残して、残りを「**XXX@XXXX.com**」に変換します。

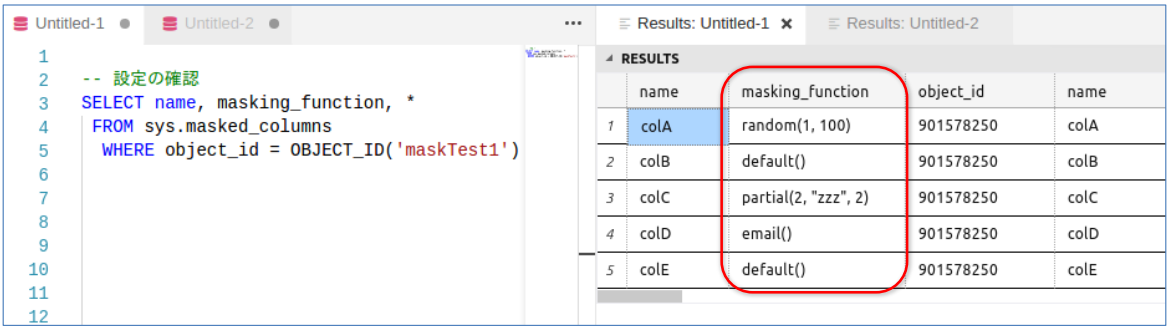
colE には **default()** 関数を設定したので、「**1900-01-01 00:00:00.000**」にマスクされています。

このように、動的データ マスクを利用すれば、一般ユーザーから簡単に情報を隠せるようになるので大変便利です。

設定の確認 ～masked_columns ビュー～

動的データ マスクの設定は、**masked_columns** システム ビューを利用して確認することができます。これは、次のように利用できます。

```
-- 設定の確認
SELECT name, masking_function, *
 FROM sys.masked_columns
  WHERE object_id = OBJECT_ID('maskTest1')
```



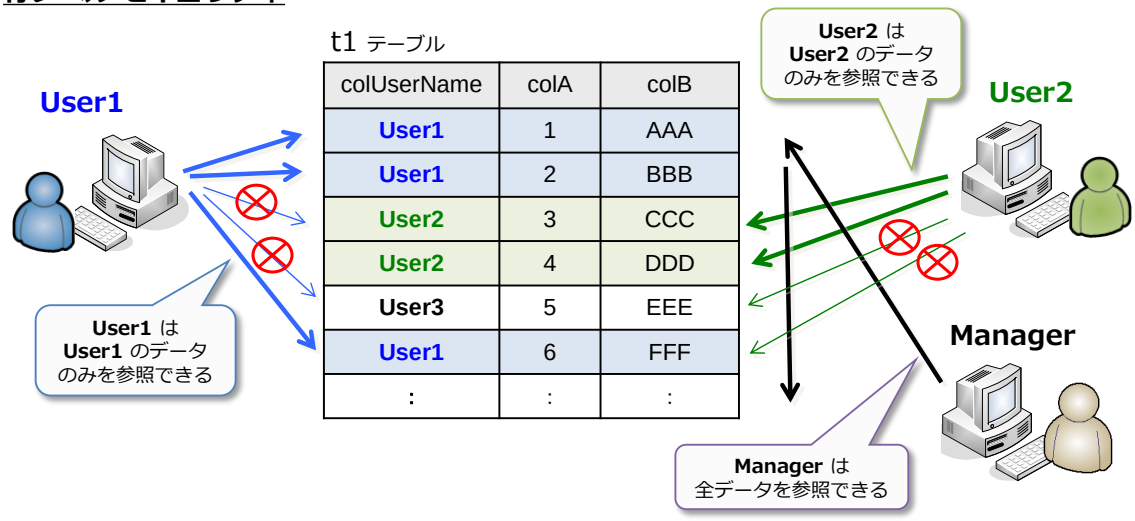
「**masking_function**」列で定義を確認することができます。

以上のように、**動的データ マスク**を利用すれば、データを簡単にマスク（違う値に変換）することができるので、**クレジット カード番号**や**マイナンバー**、**メール アドレス**などの**顧客情報／機密情報をマスク**する目的で利用することができます（セキュリティの強化、情報漏洩／データ流出の防止を実現することができます）。

5.3 行レベル セキュリティによるセキュリティ強化

行レベル セキュリティは、**行レベルのアクセス制御**を実現できる機能です。これを利用すれば、ユーザーに対して、参照できる「**行**」を制限できるようになるので、次のようにユーザーごとに、見せたい行を制限できるようになります。

行レベル セキュリティ



このように行レベル セキュリティを利用すれば、**User1** は **User1 のデータのみ**、**User2** は **User2 のデータのみ**しか参照できないように制限することができます。また、すべてのデータを参照できるユーザー（図では **Manager**）を設定することもできます。

Let's Try

それでは、これを試してみましょう。

1. まずは、行レベル セキュリティを試すためのデータベースとユーザーを作成します。

```
-- データベース「rsTestDB」の作成
CREATE DATABASE rsTestDB
go

-- データベース ユーザー User1、User2、Manager の作成
USE rsTestDB
CREATE USER User1 WITHOUT LOGIN
CREATE USER User2 WITHOUT LOGIN
CREATE USER Manager WITHOUT LOGIN
```

データベース名は「**rsTestDB**」、データベース ユーザーは「**User1**」、「**User2**」、「**Manager**」という名前で作成します（**WITHOUT LOGIN** によって、ログイン アカウントには紐付かないユーザーの作成になり、現実的なシナリオでは利用しない使い方になりますが、行レベル セキュリティを試すためだけの簡易的なユーザーの作成方法になります）。

2. 次に、**CREATE TABLE** ステートメントを利用してテーブルを作成します。

```
-- テーブル「rsTest1」の作成
CREATE TABLE rsTest1
( colUserName sysname
 ,colA int
 ,colB varchar(200) )
```

テーブル名は「**rsTest1**」とし、「**colUserName**」列は、アクセス可能なユーザーの名前を格納するために利用します。データ型には「**sysname**」を指定していますが、これは **nvarchar(128)** と同じ意味で利用できる、システム データ用の特殊なデータ型です。「**colA**」と「**colB**」列には、特に意味はないので、適当な列の名前で大丈夫です。

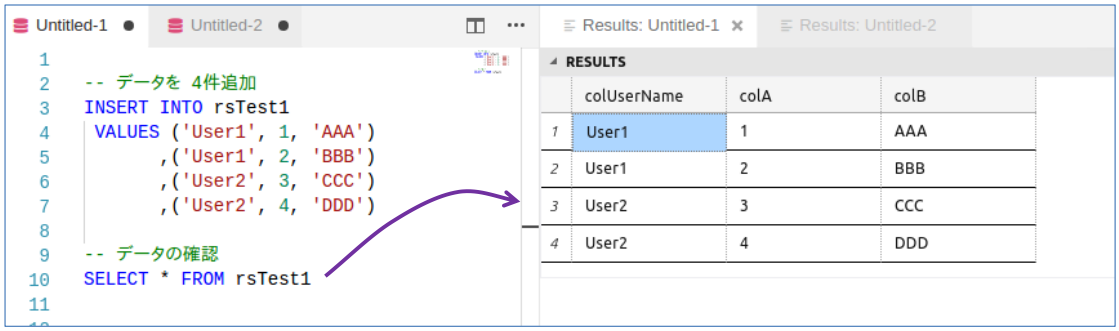
3. 次に、データを 4 件追加します。

```
-- データを 4件追加
INSERT INTO rsTest1
 VALUES ('User1', 1, 'AAA')
       ,('User1', 2, 'BBB')
       ,('User2', 3, 'CCC')
       ,('User2', 4, 'DDD')
```

1、2 件目は、**colUserName** 列に **User1** を格納して、**User1** のみが参照できるようにし、**3、4 件目**には **User2** を格納するようにします。

4. 次に、追加したデータを確認します。

```
-- データの確認
SELECT * FROM rsTest1
```



5. 続いて、データベース ユーザー（**User1**、**User2**、**Manager**）に対して、テーブルに対する **SELECT** 権限を付与します。

```
-- データベース ユーザーに、テーブルに対する SELECT 権限を付与
GRANT SELECT ON rsTest1 TO User1
GRANT SELECT ON rsTest1 TO User2
GRANT SELECT ON rsTest1 TO Manager
```

行レベル セキュリティのためのユーザー定義関数とセキュリティ ポリシーの作成

行レベル セキュリティは、**ユーザー定義関数**と**セキュリティ ポリシー**（**フィルター**）を作成することによって実現できます。

1. ユーザー定義関数は、次のように作成します。

```
-- 行レベル セキュリティを実現するためのユーザー定義関数の作成例
CREATE FUNCTION rsFunc1 (@UserName AS sysname)
 RETURNS TABLE
 WITH SCHEMABINDING
AS
 RETURN
   SELECT 1 AS result
    WHERE USER_NAME() = @UserName
       OR USER_NAME() = 'Manager'
```

ユーザー定義関数の名前は **rsFunc1**（任意の名前を設定可能）、引数に **@UserName** を追加して、**ユーザー名**を受け取れるようにします。関数の中では、**USER_NAME()** 関数を利用して接続中のユーザー名を取得し、引数として受け取ったユーザー名と等しいかどうか、あるいは **Manager** かどうかを判断して、ユーザー名が等しいまたは Manager の場合に **1** を返すようにしています。

2. ユーザー定義関数を作成した後は、**FILTER PREDICATE** を指定した**セキュリティ ポリシー**（**フィルター**）を作成します。これを行うには、次のように **CREATE SECURITY POLICY** ステートメントを記述します。

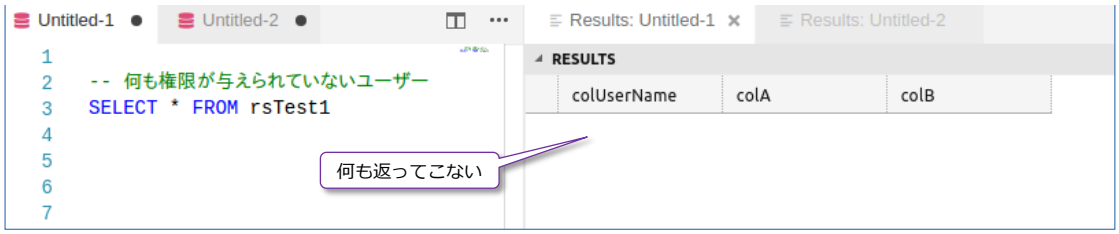
```
CREATE SECURITY POLICY rsFilter1
 ADD FILTER PREDICATE dbo.rsFunc1(colUserName)
  ON dbo.rsTest1
 WITH (STATE = ON)
```

セキュリティ ポリシーの名前は **rsFilter1**（任意の名前を設定可能）、**ADD FILTER PREDICATE** で「**dbo.rsFunc1(colUserName) ON dbo.rsTest1**」と指定することで、該当テーブル（**rsTest1**）に対して、**rsFunc1** 関数（前の手順で作成したユーザー定義関数）でフィルターをかけて、関数へ与える引数に **colUserName** 列を与えることができるようになります。末尾の **WITH (STATE = ON)** では、このセキュリティ ポリシーを有効化することができます。

以上で、行レベル セキュリティの設定が完了です。

3. 次に、**通常のユーザー**（何も権限が与えられていないユーザー）で、テーブルを **SELECT** してみます。

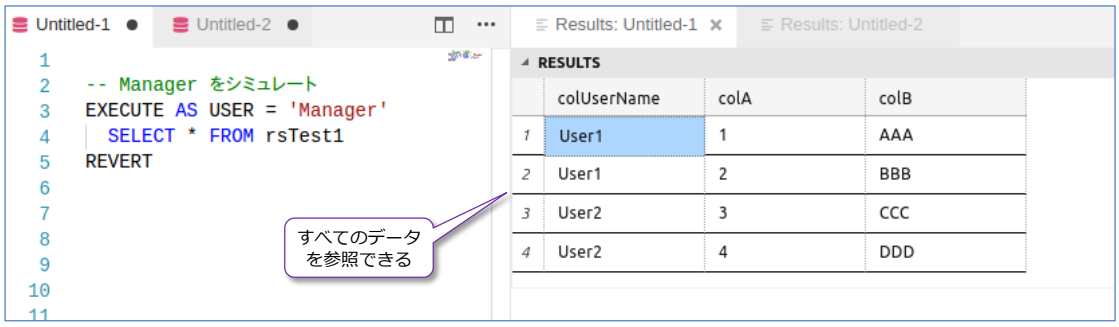
```
-- 何も権限が与えられていないユーザー
SELECT * FROM rsTest1
```



結果には、何も返ってきません。このように、テーブルに対して**フィルター**を設定している場合には、ユーザー定義関数で設定したユーザー以外は、テーブル データを参照できない形になります。

4. 次に、**Manager** ユーザーでアクセスしてみます（**EXECUTE AS USER** ステートメントで別のユーザーでの接続をシミュレートします）。

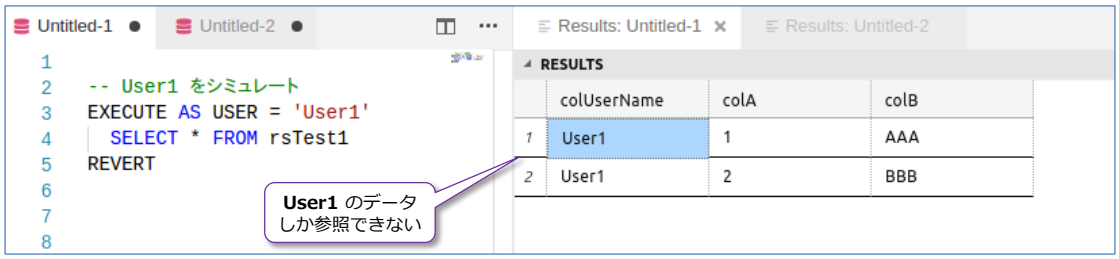
```
-- Manager をシミュレート
EXECUTE AS USER = 'Manager'
  SELECT * FROM rsTest1
REVERT
```



Manager ユーザーは、すべてのデータを参照することができます。内部的には、フィルターで指定したユーザー定義関数内の「**USER_NAME() = 'Manager'**」という部分が利用されていて、**Manager** ユーザーであれば、すべてのデータが返る形になります。

5. 次に、**User1** ユーザーでアクセスしてみます。

```
-- User1 をシミュレート
EXECUTE AS USER = 'User1'
  SELECT * FROM rsTest1
REVERT
```

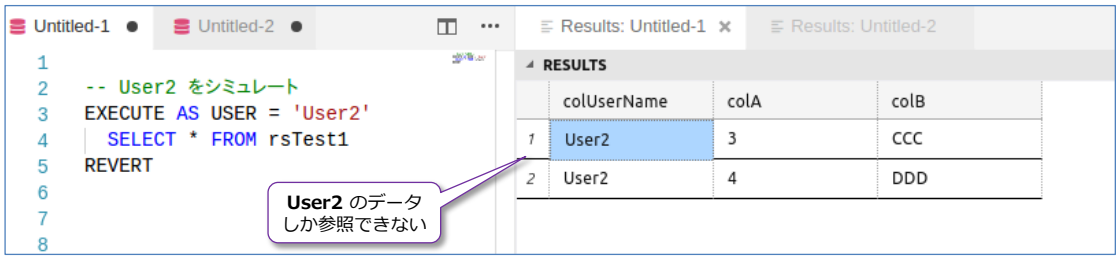


User1 ユーザーは、**colUserName** 列が **User1** のデータしか参照できないことを確認できます。

内部的には、フィルターで指定したユーザー定義関数内の「**USER_NAME() = @UserName**」という部分が利用されていて、**USER_NAME()** は **User1**（接続しているユーザーの名前）を返し、**@UserName** には、**colUserName** 列に格納されているデータ（**User1** や **User2**）が与えられるので、データが **User1** の場合には、その結果を返るという形です。

6. 次に、**User2** ユーザーでアクセスしてみます。

```
-- User2 をシミュレート
EXECUTE AS USER = 'User2'
  SELECT * FROM rsTest1
REVERT
```



User2 ユーザーは、**colUserName** 列が **User2** のデータしか参照できないことを確認できます。

このように、行レベル セキュリティを利用すれば、行レベルのアクセス制御を実現することができ、ユーザーごとに、見せたい行を制限できるようになります。

行レベル セキュリティを無効化したい場合

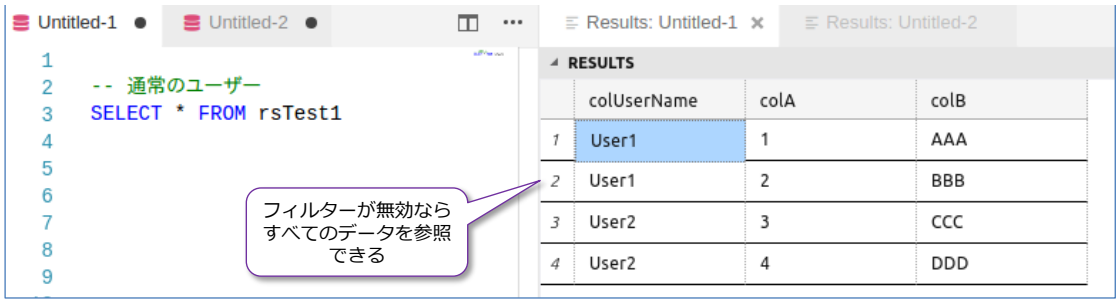
行レベル セキュリティを無効化したい場合には、作成した**フィルター**（**セキュリティ ポリシー**）を無効化するだけで完了します。これも試してみましょう。

1. フィルターを無効化するには、次のように **ALTER SECURITY POLICY** ステートメントで、**STATE** を **OFF** に設定します。

```
-- 行レベル セキュリティを無効化する（フィルターを無効化）
ALTER SECURITY POLICY rsFilter1
 WITH (STATE = OFF)
```

2. 無効化をした後は、**通常のユーザー**（何も権限が与えられていないユーザー）でも、データを参照できるようになります。

```
-- 通常のユーザー
SELECT * FROM rsTest1
```



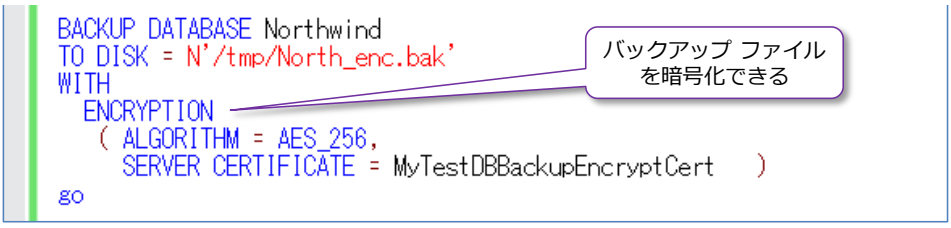
その他、行レベル セキュリティに関する情報は、オンライン ブック（SQL Server のヘルプ）の次のトピックが参考になると思います。

行レベルのセキュリティ
https://docs.microsoft.com/ja-jp/sql/relational-databases/security/row-level-security

5.4 バックアップ暗号化（Backup Encryption）

バックアップの暗号化は、バックアップ データを暗号化できるので、バックアップ ファイルの盗難への対策になります。

これを利用するには、次のように **BACKUP** ステートメントの実行時の **WITH** オプションで **ENCRYPTION** キーワードを利用します、



ALGORITHM には、暗号化アルゴリズムを指定して（画面は **AES_256** を利用している例）、**SERVER CERTIFICATE** には、サーバー証明書を指定しますが、これの作成方法については、この後、実際に試してみます。

Let's Try

それでは、バックアップ暗号化を試してみましょう。

1. まずは、**CREATE MASTER KEY** ステートメントを利用して、**master** データベース内に**データベース マスター キー**を作成します

```
-- データベース マスター キーの作成。パスワードには複雑なものを設定
USE master
CREATE MASTER KEY
 ENCRYPTION BY PASSWORD = '<StrongPassword>'
```

ENCRYPTION BY PASSWORD には、任意のパスワードを設定しますが、強固なパスワード（大文字と小文字、@ などの特殊文字を含むなど）を設定するようにします。

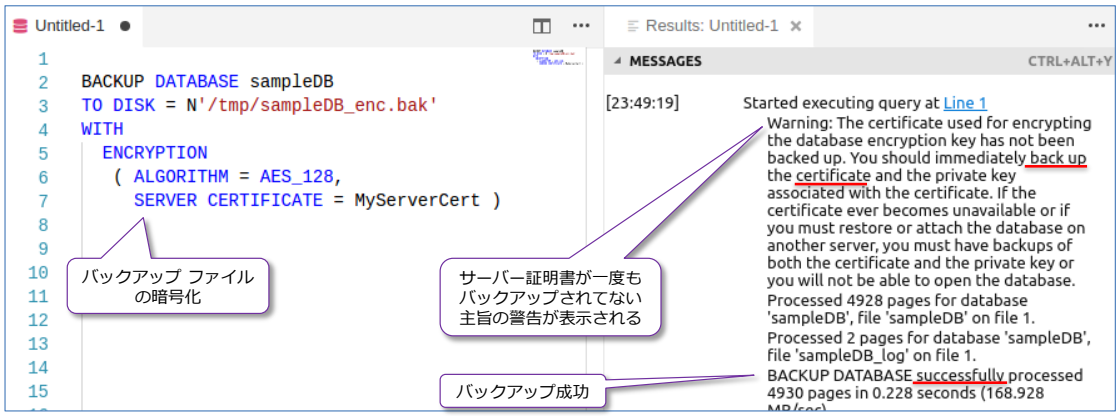
2. 次に、**CREATE CERTIFICATE** ステートメントを利用して、**サーバー証明書**を作成します（これも **master** データベース内に作成します）。

```
-- サーバー証明書の作成
CREATE CERTIFICATE MyServerCert
 WITH SUBJECT = 'My Certificate'
```

「**MyServerCert**」は、サーバー証明書の名前なので、任意の名前を設定できます。**SUBJECT** は証明書の説明になるので、これも任意の説明を設定できます。

3. 次に、バックアップ ファイルの暗号化を行います（**BACKUP** ステートメントを次のように記述します）。

```
-- バックアップ ファイルの暗号化
BACKUP DATABASE sampleDB
TO DISK = N'/tmp/sampleDB_enc.bak'
WITH
  ENCRYPTION
   ( ALGORITHM = AES_128,
     SERVER CERTIFICATE = MyServerCert )
```



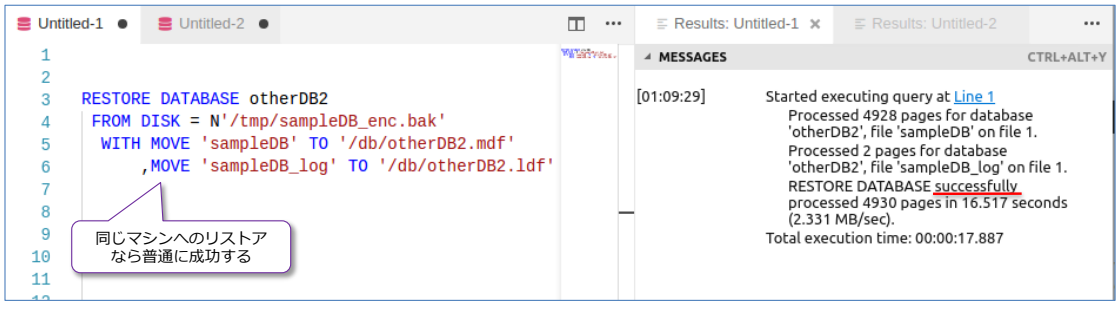
暗号化アルゴリズムでは、**AES_128** や **AES_192**、**AES_256** などを選択できますが、キーの長さに応じて（128bit より 192bit、192bit より 256bit に変更することで）、セキュリティの強度を上げることができます（その反面、性能とのトレードオフがあります）。

バックアップを実行すると、サーバー証明書が一度もバックアップされてない主旨（すぐにバックアップを取得すべきという主旨）の警告が表示されますが、サーバー証明書のバックアップ方法については、後述の TDE（透過的なデータ暗号化）のところで説明します。

リストア

1. 次に、暗号化したバックアップ ファイルを利用して、リストアを実行してみましょう。リストアは、前の章の Step 4.8 で説明した **RESTORE DATABASE** での **MOVE..TO** を利用して、別の場所に、別の名前のデータベースとして復元してみます。

```
-- 同じマシンに別の DB 名でリストア
RESTORE DATABASE otherDB2
 FROM DISK = N'/tmp/sampleDB_enc.bak'
  WITH MOVE 'sampleDB' TO '/db/otherDB2.mdf'
      ,MOVE 'sampleDB_log' TO '/db/otherDB2.ldf'
```



リストアは、何の問題もなく成功します。**同じマシン**の場合は、バックアップ ファイルが暗号化されていたとしても、リストアに関する変更点はありません。

2. 次に、**別のマシン**が用意できる場合は（Docker で別のコンテナを作成するなど）、次のように同じ **RESTORE** ステートメントを実行して、リストアを実行してみてください（リストアの実行前に、バックアップ ファイルは、別のマシン上にコピーしておく必要があります）。

```
-- 別のマシンでリストアを試みると...
RESTORE DATABASE otherDB2
 FROM DISK = N'/tmp/sampleDB_enc.bak'
  WITH MOVE 'sampleDB' TO '/db/otherDB2.mdf'
      ,MOVE 'sampleDB_log' TO '/db/otherDB2.ldf'
```

しかし、別のマシンでリストアを行おうとすると、次のようにエラーになって、リストアが失敗してしまいます。

エラー メッセージには、「**サーバー証明書が見つかりません**」とあり、サーバー証明書が原因でリストアできていないことを確認できます。

バックアップ ファイルは、暗号化された状態ですが、万が一持ち出されてたとしても、サーバー証明書がなければ、復元ができないようになっていて、簡単にはリストアできないようにセキュリティ強化されています（安全性がさらに高くなっています）。

別のマシンで、この暗号化されたバックアップ ファイルをリストアするには、全く**同じサーバー証明書**を別のマシンで作成する必要がありますが、この手順については、TDE（透過的なデータ暗号化）の場合の手順と同様なので、そこで詳しく説明します。

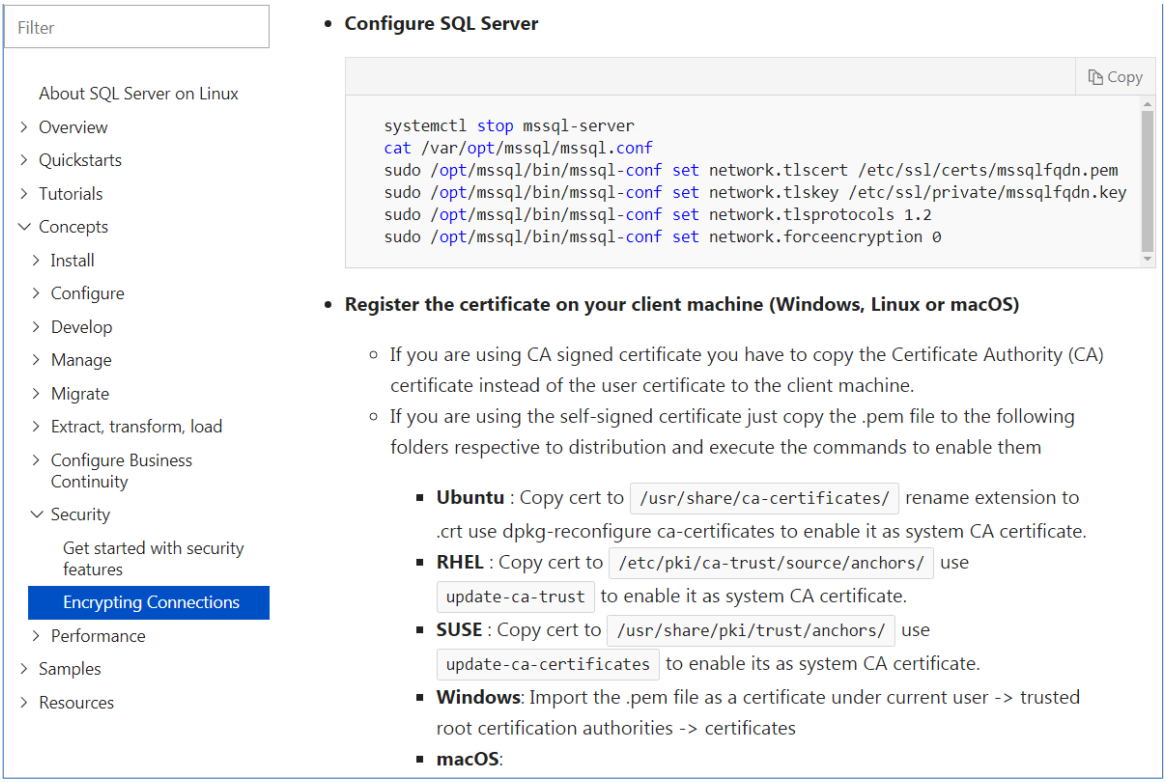
5.5 ネットワーク接続の暗号化（Network Encryption）

ネットワーク接続の暗号化は、ネットワーク上を流れるパケットを暗号化できるので、パケット盗聴への対策になります。SQL Server 2017 でネットワーク接続を有効化しておくと、クライアント（アプリケーション）からは、接続文字列に以下を追加することで暗号化接続を指定できるようになります。

```
JDBC
"encrypt=true; trustServerCertificate=false;"
ODBC
"Encrypt=Yes; TrustServerCertificate=no;"
ADO.NET
"Encrypt=True; TrustServerCertificate=False;"
```

ネットワーク接続の暗号化を SQL Server on Linux で有効化するには、**証明書**の作成や、SQL Server on Linux の構成ファイルである **mssql.conf**（**/var/opt/mssql** ディレクトリに存在）の設定変更などが必要になりますが、これらの設定手順については、オンライン ブック（SQL Server のヘルプ）の以下のページに詳しく記載されています。

Encrypting Connections to SQL Server on Linux
https://docs.microsoft.com/en-us/sql/linux/sql-server-linux-encrypted-connections



5.6 TDE（透過的なデータ暗号化）

TDE（Transparent Data Encryption: 透過的なデータ暗号化）は、データベースの実体となるファイル（.mdf／.ldf）およびバックアップ ファイルを暗号化することができる機能で、SQL Server 2008 から提供されています。1 つ前の SQL Server 2016 では、インメモリ OLTP に対応したり、TDE のオーバーヘッドを軽減させて、性能向上を図ったり強化されました。

TDE（透過的なデータ暗号化）の設定の流れ

TDE を設定するおおまかな流れは、次のとおりです（手順 1 と 2 は、バックアップ暗号化のところと同様の作業になります）。

1. **master** データベースに接続して、**データベース マスター キー**を作成（**CREATE MASTER KEY** ステートメントを利用）
2. **サーバー証明書**を作成（**CREATE CERTIFICATE** ステートメントを利用）
3. **データベース暗号化キー**を作成（**CREATE DATABASE ENCRYPTION KEY**）
4. **ALTER DATABASE .. SET ENCRYPTION ON** を利用して、データベースに対して暗号化を設定する

Let's Try

それでは、TDE を試してみましょう。バックアップ暗号化の手順を試している場合には、以下の手順のうち、手順 2（データベース マスター キーの作成）と手順 3（サーバー証明書の作成）は省略（スキップ）してください。

1. まずは、TDE を試すためのデータベースを「**tdeTestDB**」という名前で作成します。

```
-- データベースの作成
USE master
CREATE DATABASE tdeTestDB
```

2. 次に、**CREATE MASTER KEY** ステートメントを利用して、**master** データベース内に**データベース マスター キー**を作成します。

```
-- データベース マスター キーの作成。パスワードには複雑なものを設定
USE master
CREATE MASTER KEY
 ENCRYPTION BY PASSWORD = '<StrongPassword>'
```

ENCRYPTION BY PASSWORD には、任意のパスワードを設定しますが、強固なパスワー

ド（大文字と小文字、@ などの特殊文字を含むなど）を設定するようにします。

3. 次に、**CREATE CERTIFICATE** ステートメントを利用して、**サーバー証明書**を作成します（これも **master** データベース内に作成します）。

```
-- サーバー証明書の作成
CREATE CERTIFICATE MyServerCert
 WITH SUBJECT = 'My Certificate'
```

「**MyServerCert**」は、サーバー証明書の名前なので、任意の名前を設定できます。**SUBJECT** は証明書の説明になるので、これも任意の説明を設定できます。

4. 次に、**CREATE DATABASE ENCRYPTION KEY** ステートメントを利用して、**データベース暗号化キー**を作成します（これは該当データベースに接続して実行します）。

```
-- データベース暗号化キーの作成
USE tdeTestDB

CREATE DATABASE ENCRYPTION KEY
 WITH ALGORITHM = AES_128
  ENCRYPTION BY SERVER CERTIFICATE MyServerCert
```

ALGORITHM では、暗号化アルゴリズムを指定しますが、**AES_128** や **AES_192**、**AES_256** などを選択できますが、キーの長さに応じて（128bit より 192bit、192bit より 256bit に変更することで）、セキュリティの強度を上げることができます（その反面、性能とのトレードオフがあります）。

ENCRYPTION BY SERVER CERTIFICATE では、上の手順で作成した「**MyServerCert**」サーバー証明書を指定します。

5. 次に、データベースに対して、TDE を有効化します。これは、**ALTER DATABASE .. SET ENCRYPTION ON** を利用します。

```
-- データベースに対して TDE を有効化
ALTER DATABASE tdeTestDB
 SET ENCRYPTION ON
```

以上で TDE の設定が完了です。

TDE を設定すると、データベースの実体となるファイルが暗号化されることはもちろんのこと、バックアップ ファイルについても暗号化されるようになります（バックアップ暗号化を利用しなくても、TDE が設定されていれば、バックアップ ファイルを暗号化することができます）。

別マシンでのリストアはエラー、サーバー証明書の複製

TDE（透過的なデータ暗号化）を設定した**データベースのバックアップ**は、別のマシンでリストアしようとするとエラーが発生します。これは、バックアップ ファイルが持ち出された場合に、簡単にリストアできないように**セキュリティを強化**している機能です（かつ、バックアップ ファイルの中身そのものは暗号化されているので、さらにセキュリティが高いものになっています）。

また、SQL Server では、データ ファイル（.mdf）およびログ ファイル（.ldf）を複製して、別のマシンでアタッチする（.mdf と .ldf）という機能がありますが、TDE を設定した場合は、これもできないようにセキュリティを高めています（簡単にアタッチできないようにしています）。

それでは、これを試してみましょう。

1. まずは、TDE を設定したデータベースの**バックアップ**を取得します。

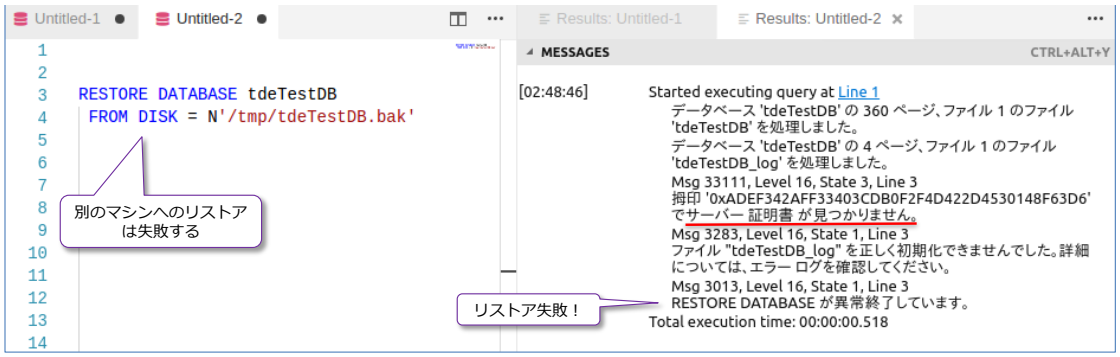
```
-- バックアップの取得
BACKUP DATABASE tdeTestDB
 TO DISK = N'/tmp/tdeTestDB.bak'
```

2. 次に、取得したバックアップ ファイル（**tdeTestDB.bak**）を別のマシンにコピーします。

3. コピーが完了したら、別のマシンで**リストア**を行ってみます。

```
-- 別マシンでリストア失敗
RESTORE DATABASE tdeTestDB
 FROM DISK = N'/tmp/tdeTestDB.bak'
```

結果は、次のようにエラーになります。



エラー メッセージには、「**サーバー証明書が見つかりません**」とあり、サーバー証明書が原因でリストアできていないことを確認できます。

別のマシンでリストアを行うには、元のマシンと全く**同じサーバー証明書**が必要になります。同じ証明書を作成するには、サーバー証明書のバックアップとリストアを利用します。開発機から本番機へ、あるいはその逆へなど、別マシンにデータベースを移動したい場合には、お互いに同じサーバー証明書を保持しておくようにします。

サーバー証明書のバックアップとリストア

1. サーバー証明書のバックアップは、**BACKUP CERTIFICATE** ステートメントを利用して、次のように実行します（これは TDE を設定したマシン側で行います）。

```
-- サーバー証明書のバックアップ（master に接続して実行する）
USE master
BACKUP CERTIFICATE MyServerCert
 TO FILE = '/tmp/MyServerCert'
 WITH PRIVATE KEY
         ( FILE = '/tmp/MyPrivKey'
          ,ENCRYPTION BY PASSWORD = 'P@ssword' )
```

BACKUP CERTIFICATE に続けてバックアップしたいサーバー証明書を指定し、**TO FILE=** でバックアップ ファイルの保存場所（**/tmp** ディレクトリに **MyServerCert** というファイル名で保存）を指定、**WITH PRIVATE KEY** で秘密キーのバックアップも行います。このファイルの場所は **FILE=** で指定して（上の例では **MyPrivKey** という名前で保存）、**ENCRYPTION BY PASSWORD** でパスワードで保護しておきます。上の例は P@ssword を指定していますが、実際に利用するときは、複雑なパスワードを設定するようにしてください（推測されにくい強固なパスワードを設定して、そのパスワードが簡単に漏れないように注意が必要です）。

2. 次に、バックアップしたサーバー証明書（**/tmp** ディレクトリの **MyServerCert** と **MyPrivKey** ファイル）を別のマシンにコピーします。

3. コピーが完了したら、**CREATE CERTIFICATE** ステートメントを利用して、別マシンでサーバー証明書のリストア（インポート）を行いますが、これを行うには、**データベース マスター キー**を事前に作成しておく必要があります。

```
-- 別マシン側： データベース マスター キーの作成（同じものでなくても OK）
USE master
CREATE MASTER KEY
 ENCRYPTION BY PASSWORD = '<StrongPassword>'

-- 別マシン側： サーバー証明書のリストア
CREATE CERTIFICATE MyServerCert
 FROM FILE = '/tmp/MyServerCert'
 WITH PRIVATE KEY
         ( FILE = '/tmp/MyPrivKey'
         , DECRYPTION BY PASSWORD = 'P@ssword' )
```

データベース マスター キーは、**CREATE MASTER KEY** で作成し、**ENCRYPTION BY PASSWORD** で設定するパスワード（複雑なものを設定）は、元のマシンに合わせる必要はありません（異なるパスワードで大丈夫です）。

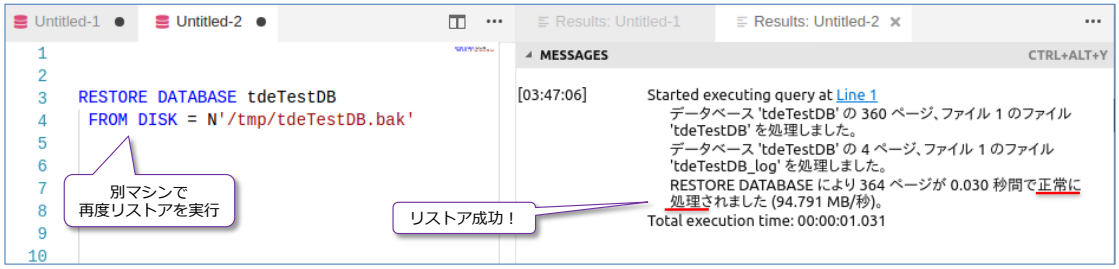
サーバー証明書のリストアには、**CREATE CERTIFICATE** ステートメントを利用しますが、**FROM FILE=** でコピーした **MyServerCert** ファイル（**/tmp** ディレクトリにある場合）、

WITH PRIVATE KEY の **FILE=** でコピーした **MyPrivKey** ファイルを指定して、**DECRYPTION BY PASSWORD** では、秘密キーをバックアップしたときに設定したのと同じパスワード（復号のためのパスワード）を指定します。

以上で、サーバー証明書のリストア（全く同じサーバー証明書の作成）が完了です。

4. 次に、データベースのリストアをもう一度実行して、今度はリストアが成功することを確認しておきましょう。

```
-- リストアが成功することを確認
RESTORE DATABASE tdeTestDB
 FROM DISK = N'/tmp/tdeTestDB.bak'
```



このように、TDE を設定すると、データベース本体やバックアップ ファイルが暗号化されるだけでなく、簡単にリストアできないようにもなっているので、セキュリティが非常に高くなります。

TDE は、ここ数年、弊社へのお問い合わせが非常に多くなっている機能です（マイナンバーや各種の情報漏洩事件など、業界全体のセキュリティ意識の高まりが大きな要因だと思っています）。TDE は、オーバーヘッドも軽く（弊社のお客様環境では、数%～ 5%程度）、アプリケーションは全く変更する必要がないので、ぜひ活用してみてください。

その他、TDE の詳細（アーキテクチャなど）については、オンライン ブック（SQL Server のヘルプ）の以下のトピックがお勧めです。

透過的なデータ暗号化（TDE）
https://docs.microsoft.com/ja-jp/sql/relational-databases/security/encryption/transparent-data-encryption

5.7 SQL Server Audit（監査）による操作履歴の記録

SQL Server Audit（SQL Server 監査）機能は、SQL Server に対して発行されたすべての操作を監査（ログ記録）できる機能で、SQL Server 2008 から提供されています。SQL Server Audit は、**J-SOX 法**（日本版 SOX 法）や**内部統制**、**PCI DSS**（クレジット カード情報のセキュリティ基準）などの**コンプライアンス**（**法令遵守**）を実現するために欠かせない機能になります。

SQL Server Audit を利用すれば、「**いつ**」「**誰が**」「**どのデータベースに対して**」「**どんなステートメント**」を実行したのかを記録したり、**特権ユーザー**（管理権限を持ったログイン アカウント）の不正利用を記録したりできるようになるので、J-SOX 法で求められる財務諸表に対する「**監査（Audit)**」（ログ記録による虚偽表示対策）の実現や、顧客情報の漏えい対策、内部統制の実現など、昨今求められているセキュリティ要件を実現することができます。

Let's Try

それでは、SQL Server Audit を試してみましょう。

1. まずは、新しく「**監査**」を作成して、監査ログの記録場所を指定します。監査を作成するには、次のように **CREATE SERVER AUDIT** ステートメントを利用します（これは **master** データベースに接続した状態で実行します）。

```
-- 監査の作成
USE master
CREATE SERVER AUDIT AuditTest1
 TO FILE
  ( FILEPATH = N'/AuditTest1' )
```

監査の名前は「**AuditTest1**」など、任意の名前を入力して、**TO FILE** の **FILEPATH** に監査ログを記録するための**ディレクトリ**へのパスを記述します（ファイル名ではなく、ディレクトリ名へのパスを設定します。ここでは **/AuditTest1** という名前のディレクトリを事前に作成済みで、そこを指定しています）。

2. 次に、作成した**監査を有効化**します。監査を有効化するには、**ALTER SERVER AUDIT** ステートメントを利用して、**STATE=ON** を設定します。

```
-- 監査の有効化
ALTER SERVER AUDIT AuditTest1
 WITH ( STATE = ON )
```

3. 次に、**ログインの失敗**や、**ログイン アカウントの作成**、ログイン アカウントに対する**パスワード変更**などを監査ログに記録するために、**サーバー監査の仕様**（Server Audit Specification）というものを作成します。これは、**CREATE SERVER AUDIT SPECIFICATION** ステートメントを次のように記述します（これも **master** データベースに接続した状態で実行します）。

```
-- サーバー監査の仕様を作成。ログインの失敗やログイン作成、パスワード変更を監査
CREATE SERVER AUDIT SPECIFICATION ServerAuditTest1
 FOR SERVER AUDIT AuditTest1
    ADD ( FAILED_LOGIN_GROUP )
   ,ADD ( SERVER_PRINCIPAL_CHANGE_GROUP )
   ,ADD ( LOGIN_CHANGE_PASSWORD_GROUP )
 WITH ( STATE = ON )
```

サーバー監査の仕様の名前は「**ServerAuditTest1**」にして、**FOR SERVER AUDIT** で前の手順で作成した監査（**AuditTest1**）を指定しています。

ADD に続けて、**FAILED_LOGIN_GROUP** を指定することで、**ログインの失敗**、**SERVER_PRINCIPAL_CHANGE_GROUP** で**サーバー プリンシパルの変更**（ログイン アカウントは、内部的にはサーバー プリンシパルと呼ばれています）、**LOGIN_CHANGE_PASSWORD_GROUP** でログイン アカウントに対する**パスワードの変更**があった場合を監査ログに残せるようになります。

また、**STATE=ON** を設定して、このサーバー監査の仕様を有効化しています。

4. 次に、**sampleDB** データベース内の **t1** テーブルに対するアクセスを監査ログに記録するために、**データベース監査の仕様**（Database Audit Specification）というものを作成します。これは、**CREATE DATABASE AUDIT SPECIFICATION** ステートメントを利用して、次のように記述します（これは、**sampleDB** データベースに接続した状態で実行します）。

```
-- データベース監査の仕様を作成。t1 テーブルに対する SELECT、INSERT を監査する
USE sampleDB
CREATE DATABASE AUDIT SPECIFICATION DbAuditTest1
 FOR SERVER AUDIT AuditTest1
   ADD ( SELECT ON OBJECT::dbo.t1 BY dbo )
  ,ADD ( INSERT ON OBJECT::dbo.t1 BY dbo )
 WITH ( STATE = ON )
```

データベース監査の仕様の名前は「**DbAuditTest1**」にして、**FOR SERVER AUDIT** で監査（**AuditTest1**）を指定しています。

ADD に続けて、**SELECT ON OBJECT::dbo.t1 BY dbo** と指定することで、**t1** テーブルに対する **SELECT** ステートメントが、**BY dbo**（dbo によって）実行されたときに監査ログに残せるようになります。また、**INSERT** についても同様に設定しているので、INSERT ステートメントの実行も監査ログに残せます。

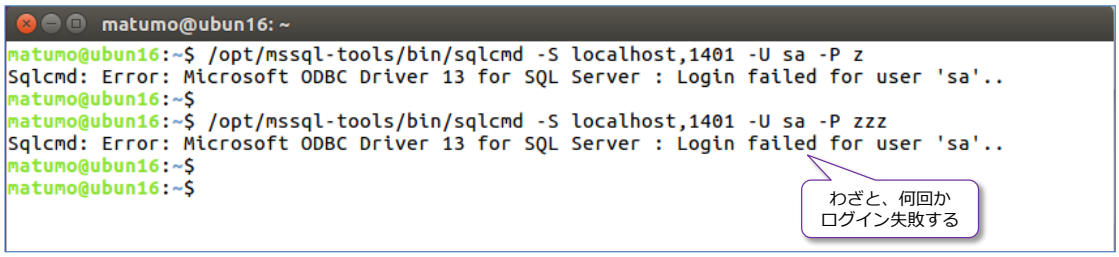
また、**STATE=ON** を設定して、このサーバー監査の仕様を有効化しています。

以上で、監査の設定が完了です。

後は、実際に操作の記録が残るかどうかをチェックしてみましょう。

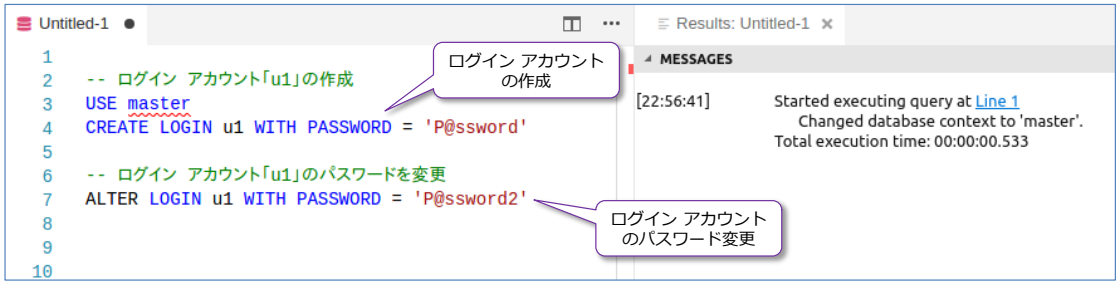
5. まずは、わざと**ログインに失敗**してみます。次のようにターミナルから **sqlcmd** ツールを利用して、**sa** のパスワードを適当に間違えて、何回かログインを試みてください。

```
# わざと 2～3回 ログインに失敗する
/opt/mssql-tools/bin/sqlcmd -S localhost,1401 -U sa -P 適当なパスワード
```



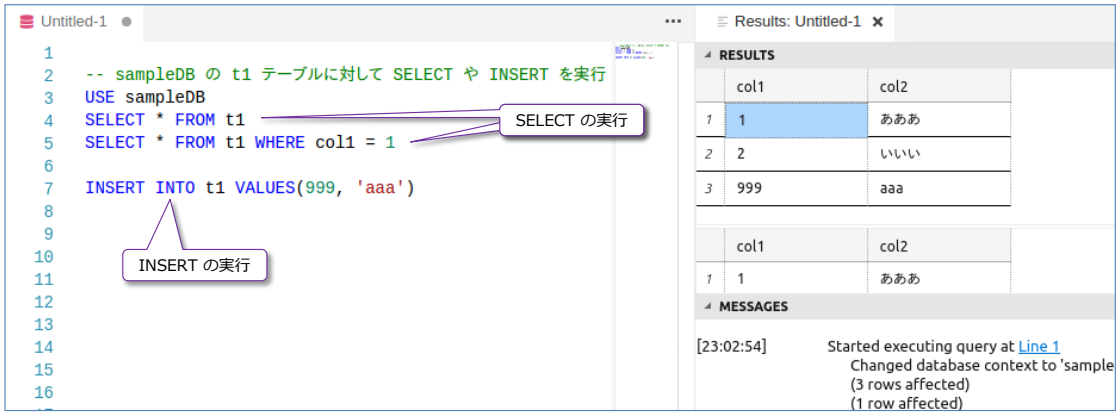
6. 次に、**ログイン アカウントを作成**して、そのログイン アカウントの**パスワードを変更**してみましょう。

```
-- ログイン アカウント「u1」の作成
USE master
CREATE LOGIN u1 WITH PASSWORD = 'P@ssword'

-- ログイン アカウント「u1」のパスワードを変更
ALTER LOGIN u1 WITH PASSWORD = 'P@ssword2'
```



7. 次に、**sampleDB** データベースの **t1** テーブルに対して、**SELECT** や **INSERT** を実行してみましょう。

```
-- sampleDB の t1 テーブルに対して SELECT や INSERT を実行
USE sampleDB
SELECT * FROM t1
SELECT * FROM t1 WHERE col1 = 1

INSERT INTO t1 VALUES(999, 'aaa')
```



監査ログの参照 ～fn_get_audit_file～

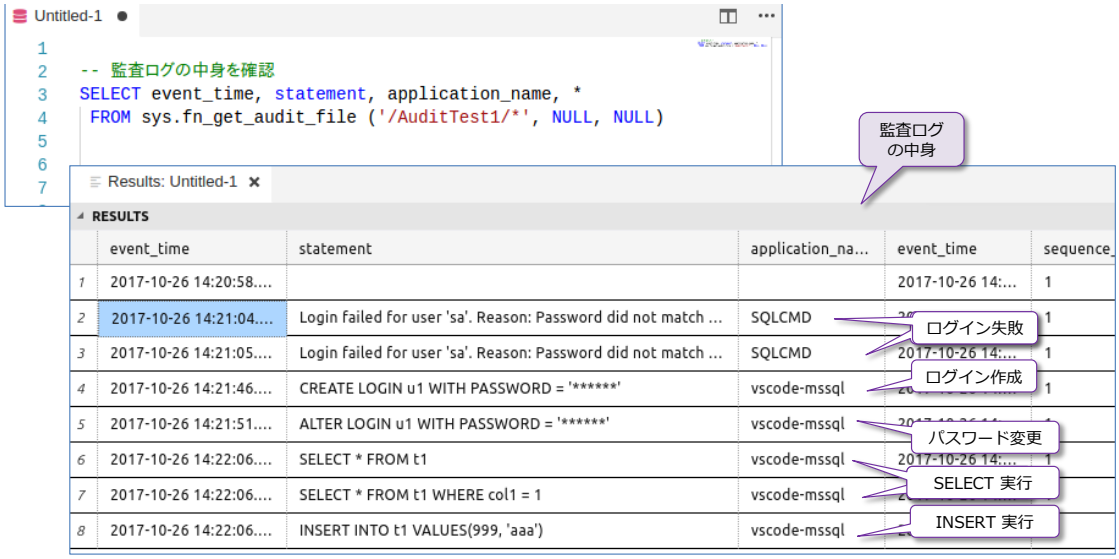
監査で記録したログを参照するには、**sys.fn_get_audit_file** というシステム関数を利用します。

1. **fn_get_audit_file** システム関数は、次のように利用します。

```
-- 監査ログの中身を確認
SELECT event_time, statement, application_name, *
 FROM sys.fn_get_audit_file ('/AuditTest1/*', NULL, NULL)
```

この関数では、第１引数に監査ログを記録しているディレクトリのパス（**/AuditTest1**）に **/*** を加えて、ディレクトリ配下のすべての監査ログ（ファイル）を参照するようにしています。第２引数と第３引数では、どのファイルから読み取るのかや、ファイルのどこから読み取るのかを設定できますが、今回はすべてを読み取るので **NULL, NULL** と指定しています。

結果は、次のように表示されます。

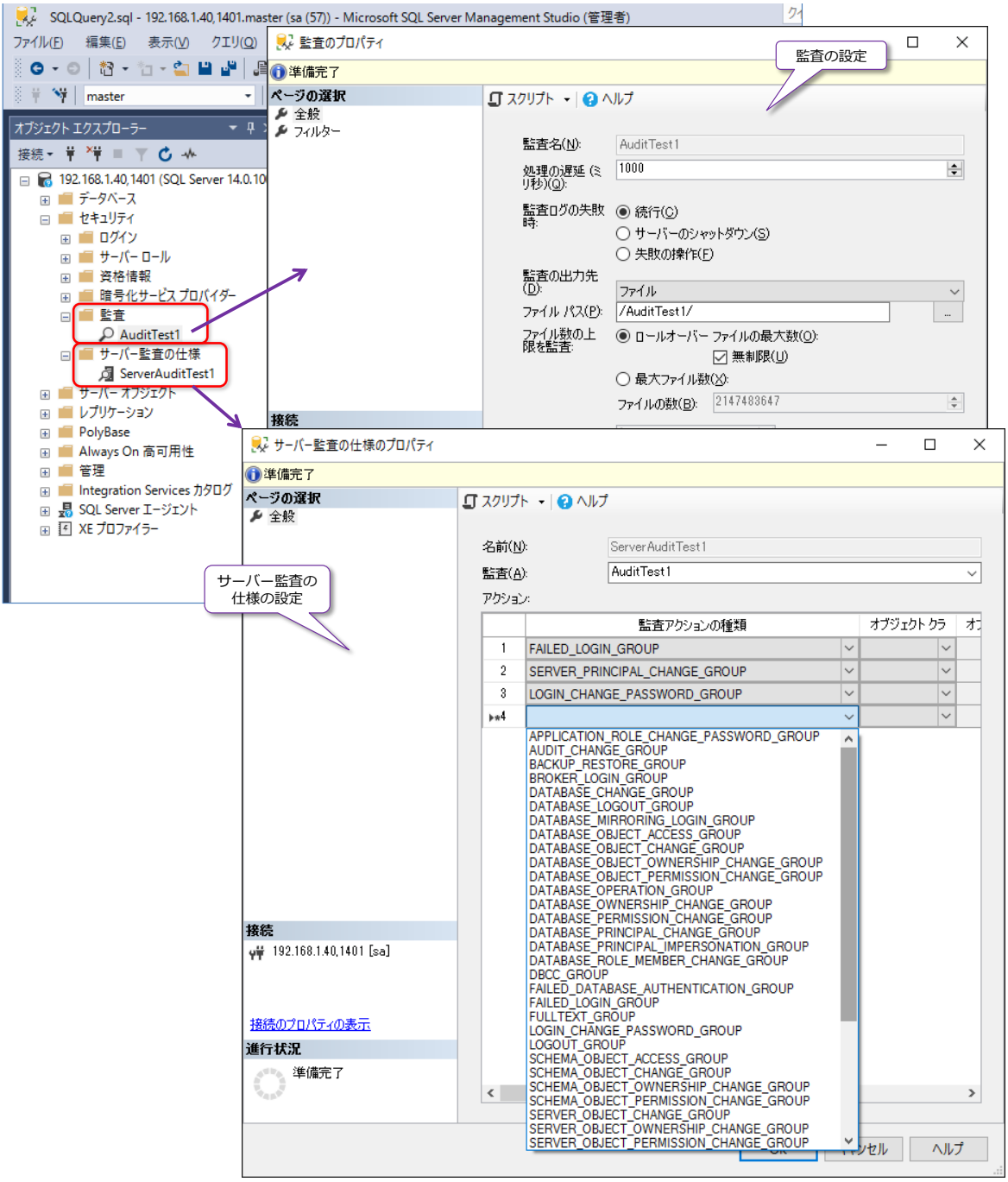


ログインの失敗や、ログイン アカウントの作成、パスワードの変更、SELECT／INSERT ステートメントの実行などが記録されていること確認できると思います。この結果に対して、GROUP BY 演算を組み合わせて実行すれば、セキュリティに関する集計レポート（Weekly レポートや Monthly レポートなど）も簡単に作成することができます。

このように、SQL Server Audit 機能を利用すれば、ログイン アカウントに対する変更履歴をすべてログへ記録できるようになるので、ログイン アカウントを不正利用（管理者アカウントのパスワードを変更されたり、管理者アカウントが勝手に作られるなど）をログへ記録できるようになります（特権ユーザーの不正使用を記録できます）。

データベースに関する操作は、今回は **t1** テーブルのみを対象にしましたが、SQL Server Audit 機能では、SQL Server に対するあらゆる操作を監査できるので、どんなテーブルでも、どんなユーザーからでも、どういった操作（SELECT／INSERT／UPDATE／DELETE／EXEC など）なのかも、設定次第ですべて記録することができます。

なお、Windows 環境の **Management Studio** から Linux 上の SQL Server をリモート操作できる場合には、次のように GUI を利用して、監査を簡単に作成／変更することができます。



その他のサーバー監査の仕様

サーバー監査の仕様では、**FAILED_LOGIN_GROUP** や **LOGIN_CHANGE_PASSWORD_GROUP** などを利用しましたが、その他にも、「**SERVER_PERMISSION_CHANGE_GROUP**」で SQL Server レベルの権限の変更を記録したり、「**AUDIT_CHANGE_GROUP**」で監査の設定そのものが変更（作成／変更／削除）されたことを記録できるなど、セキュリティ強化／コンプライアンス実現のための仕様がたくさん用意されています。これらは、オンライン ブック（SQL

Server のヘルプ）の以下のトピックに詳しく記載されているので、参考になると思います。

SQL Server 監査のアクション グループとアクション
https://docs.microsoft.com/ja-jp/sql/relational-databases/security/auditing/sql-server-audit-action-groups-and-actions

概要
アクション グループとアクション
監査レコード
サーバー監査およびサーバー監査の仕様を作成する
サーバー監査およびデータベース監査の仕様を作成する
SQL Server 監査ログを表示する
セキュリティ ログへの SQL Server 監査イベントの書き込み

サーバー レベルの監査アクション グループ

サーバー レベルの監査アクション グループは、 SQL Server セキュリティ監査イベント クラスに似ています。 詳しくは、「 SQL Server Event Class Reference」をご覧ください。

サーバー レベルの監査アクション グループと、同等の SQL Server イベント クラス (存在する場合) を次の表に示します。

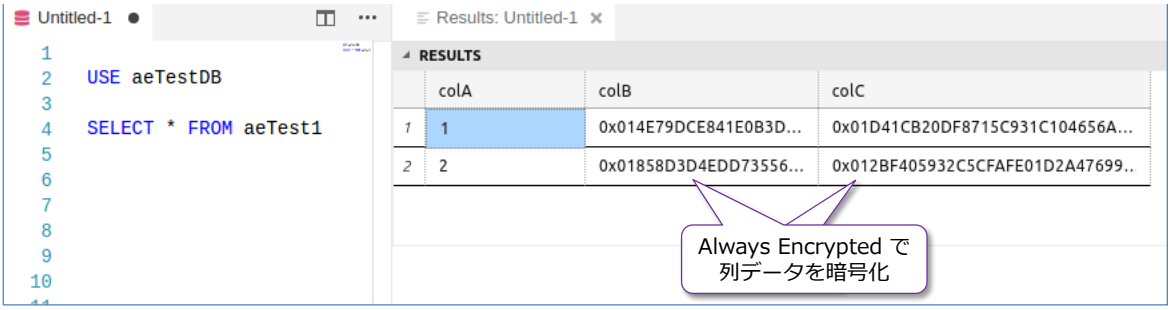
アクション グループ名	説明
APPLICATION_ROLE_CHANGE_PASSWORD_GROUP	このイベントは、アプリケーション ロールのパスワードが変更されるたびに発生します。 Audit App Role Change Password Event Classと同じです。
AUDIT_CHANGE_GROUP	このイベントは、任意の監査が作成、変更、または削除されるたびに発生します。 このイベントは、任意の監査の仕様が作成、変更、または削除されるたびに発生します。 監査に対する変更はすべてその監査内で監査されます。 Audit Change Audit Event Classと同じです。
BACKUP_RESTORE_GROUP	このイベントは、バックアップまたは復元のコマンドが実行されるたびに発生します。 Audit Backup/Restore イベント クラスと同じです。
BROKER_LOGIN_GROUP	このイベントは、Service Broker トランスポート セキュリティに関する監査メッセージを報告するために発生します。 Audit Broker Login Event Classと同じです。
DATABASE_CHANGE_GROUP	このイベントは、データベースが作成、変更、または削除されるときに発生します。 このイベントは、任意

このように、SQL Server Audit を利用すれば、データベースに対するあらゆる操作を監査（ログ記録）することができるので、「**得意先**」や「**顧客**」テーブルなどを監査対象にしておけば、顧客情報（クレジット カードや個人情報など）の漏えいが発生した場合の証跡を残せるようになります。また、特権ユーザー（管理者権限を持ったアカウント）の操作履歴（SQL Server に対する管理操作やデータ操作）をすべて監査しておけば、不正利用（管理者アカウントを悪用したデータの盗難やトロイの木馬の設置など）を監視できるようになります。

以上のように、SQL Server Audit は、**J-SOX 法**（日本版 SOX 法）や**内部統制**、**PCI DSS**（クレジット カード情報のセキュリティ基準）などのコンプライアンス（法令遵守）の実現およびセキュリティの強化には、不可欠な機能になります。

5.8 Always Encrypted による列データの暗号化

Always Encrypted（常に暗号化）は、ネットワーク上を流れるデータも、データベース内に格納されるデータも、すべて暗号化して格納できる機能です。内部的には、次のように**列データを暗号化**して格納することで、これを実現しています（アプリケーションからも透過的に利用できます）。



Let's Try

それでは、Always Encrypted を試してみましょう。Always Encrypted では、**証明書**（マスターキー）を利用して暗号化を行うのですが、この証明書の作成は、コマンドでの操作だとちょっと大変なので、**Management Studio** ツールを利用して、ウィザードで簡単に作成する手順を説明します。SQL Server on Linux をリモート操作可能な Windows マシンに、Management Studio 17.x をインストールして、以降の手順を試してみてください。

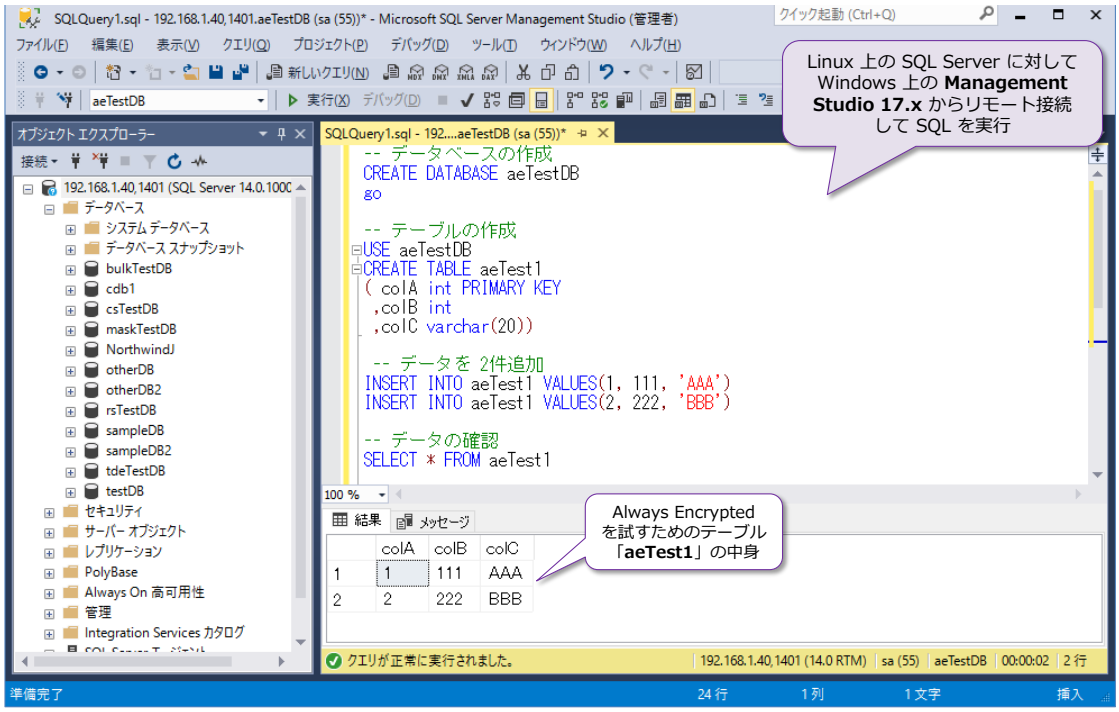
1. まずは、Always Encrypted を試すためのデータベースとテーブルを作成します。

```
-- データベースの作成
CREATE DATABASE aeTestDB
go

-- テーブルの作成
USE aeTestDB
CREATE TABLE aeTest1
( colA int PRIMARY KEY
 ,colB int
 ,colC varchar(20))

 -- データを 2件追加
INSERT INTO aeTest1 VALUES(1, 111, 'AAA')
INSERT INTO aeTest1 VALUES(2, 222, 'BBB')

-- データの確認
SELECT * FROM aeTest1
```

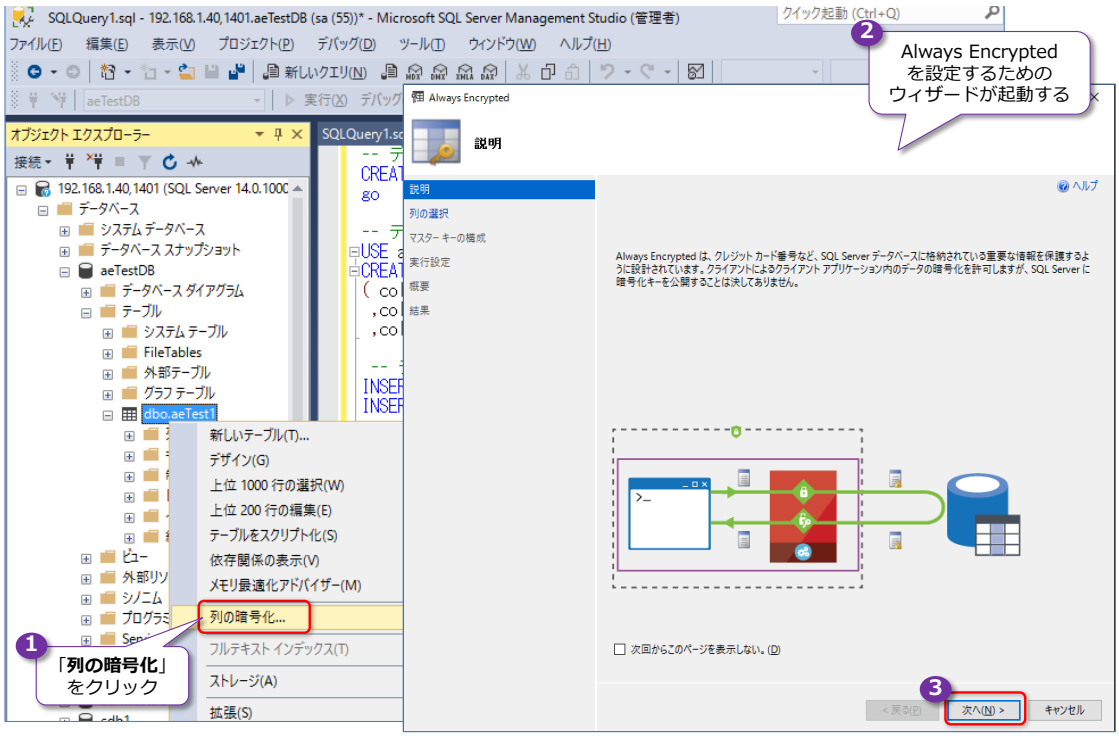
データベース名は「**aeTestDB**」として、この中に「**aeTest1**」という名前のテーブルを作成し、データを 2 件追加しておきます。

テーブルには、「**colA**」と「**colB**」、「**colC**」の 3 列を作成していますが、この後の手順で「**colB**」

と「**colC**」列を暗号化していきます。



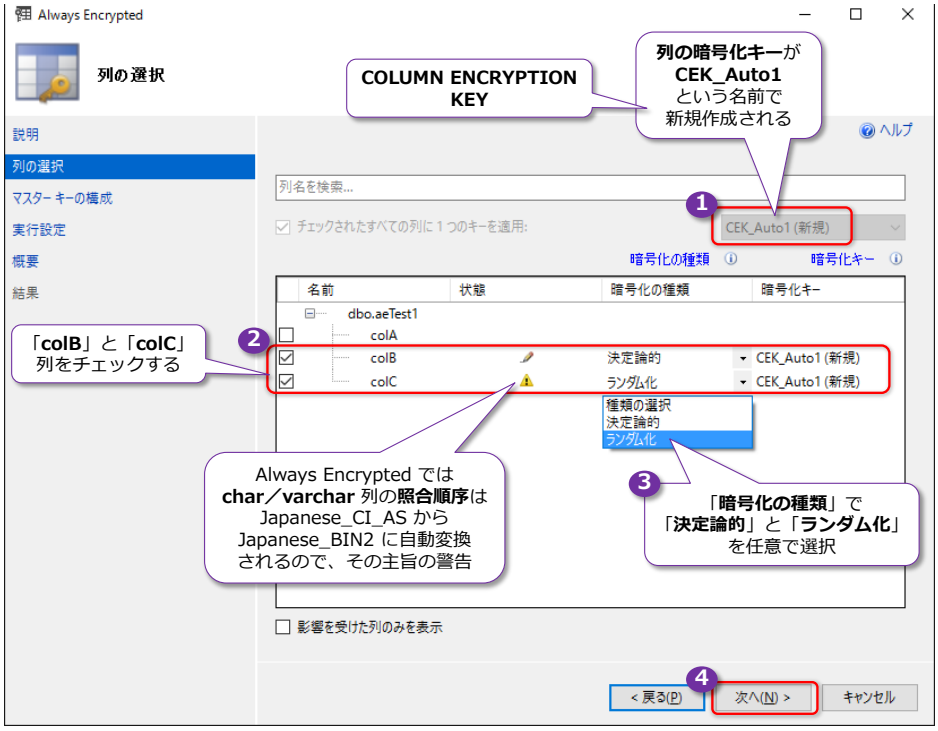
2. 次に、Management Studio のオブジェクト エクスプローラーで、作成したデータベース「**aeTestDB**」を展開して、作成したテーブル「**aeTest1**」を右クリックして、[**列の暗号化**] をクリックします。



これによって、[**Always Encrypted**] ウィザードが起動するので、最初のページでは [**次へ**] ボタンをクリックします。

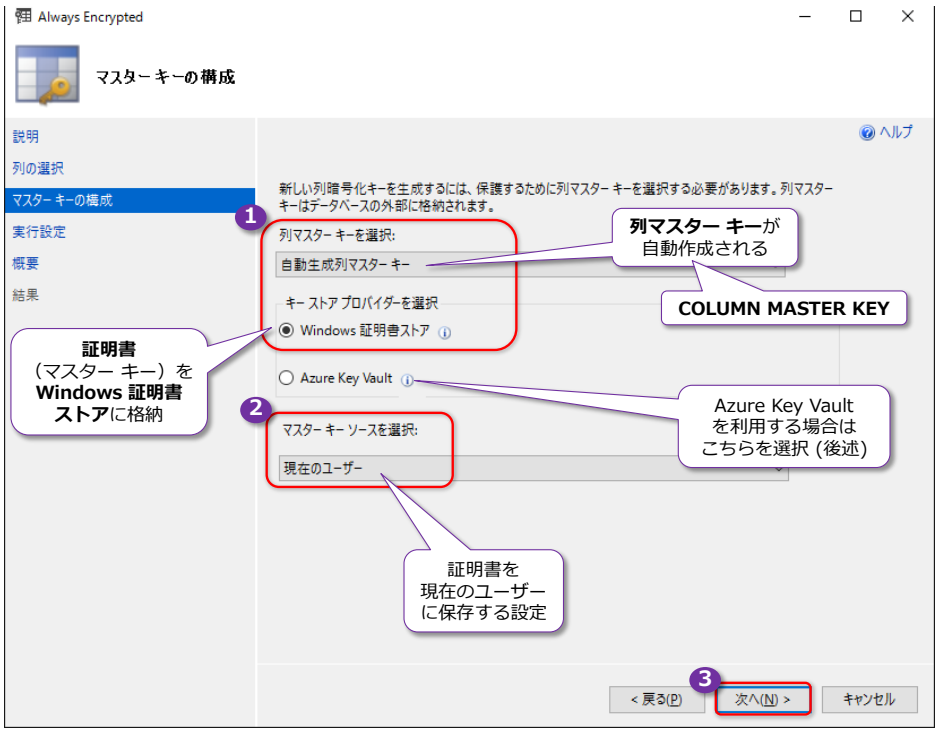
3. 次の［**列の選択**］ページでは、［**CEK_Auto1（新規）**］が選択されていることを確認します。

これは、列データを暗号化するための「**列の暗号化キー**」（**COLUMN ENCRYPTION KEY**）になるもので、ウィザードによって自動的に作成されます。



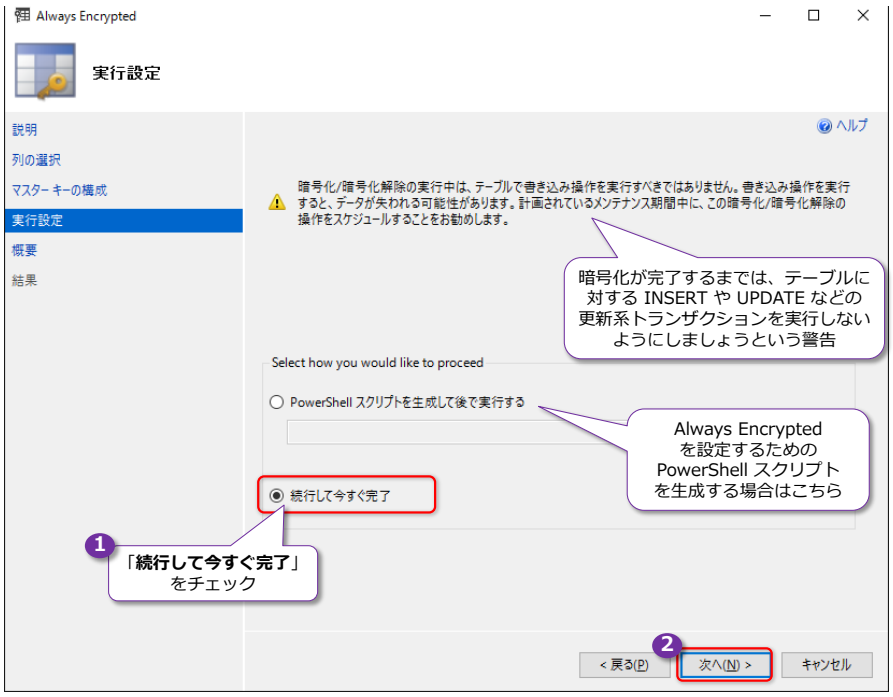
「**aeTest1**」テーブルの列の一覧からは、「**colB**」と「**colC**」列をチェックして、これを暗号化するようにします。［**暗号化の種類**］では、［**決定論的**］または［**ランダム化**］を任意に設定します。

4. 次の［**マスター キーの構成**］ページでは、**列マスター キー**（**COLUMN MASTER KEY**）をどこに作成するのかを設定します（これが**証明書**を作成する設定になります）。

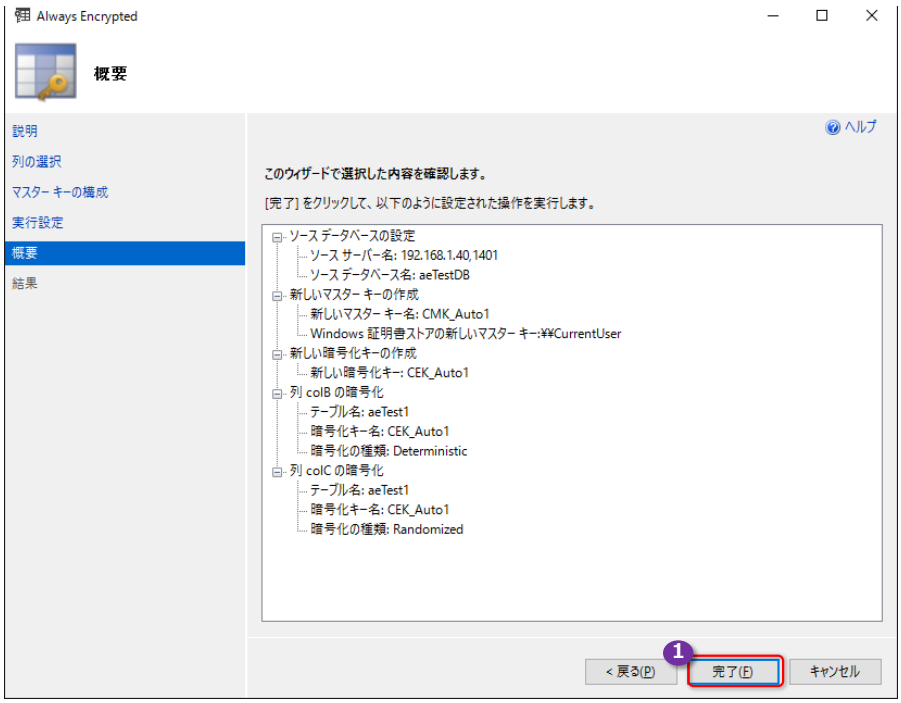


ここでは、[**列マスター キーを選択**] で [**自動生成列マスター キー**]、[**キー ストア プロバイダーを選択**] で [**Windows 証明書ストア**]、[**マスター キー ソースを選択**] で [**現在のユーザー**] が選択されていること（すべて既定値）を確認します。これで、ユーザーの証明書ストアに列マスター キーを作成することができます。なお、ここで **Azure Key Vault** を選択する方法については後述します。

5. 次の［**検証ページ**］では、［**続行して今すぐ完了**］を選択して、次のページに進みます。



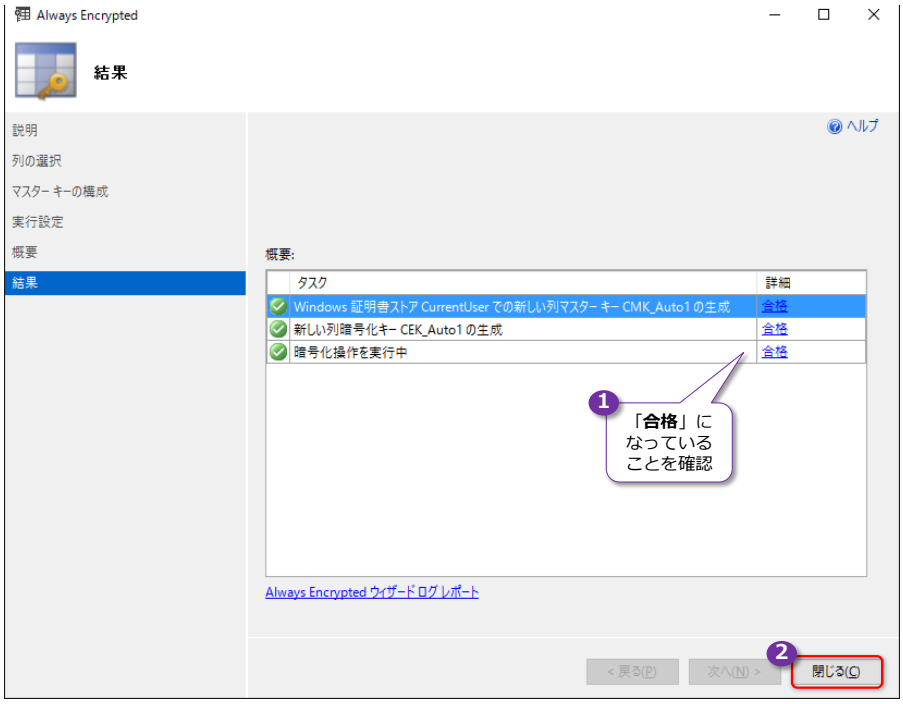
6. 最後の［**概要**］ページでは、設定内容を確認して、［**完了**］ボタンをクリックします。



［**完了**］ボタンをクリックすると、証明書（マスター キー）の作成や、暗号化キーの作成、こ

れらのキーを利用した実際の列データの暗号化などが行われます。

7. すべての作成／暗号化が完了すると、次のように［**結果**］ページで［**合格**］と表示されます。



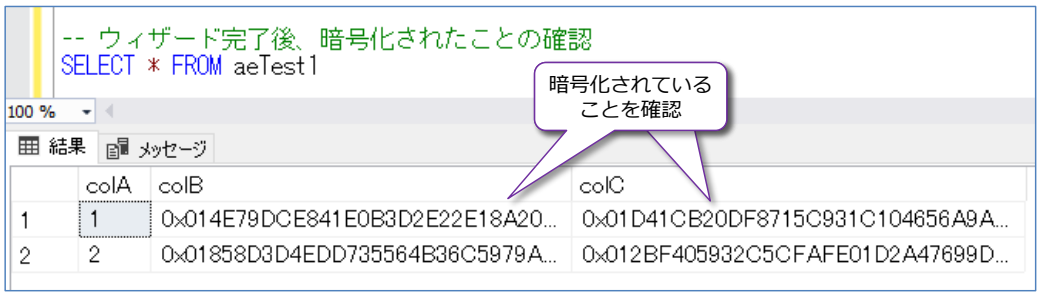
以上で、**Always Encrypted のすべての設定（列マスター キー**や、**列の暗号化キー**の作成、**colB** と **colC** 列に対する暗号化の実行）が完了です。

暗号化されていることの確認、証明書（キー）の確認

次に、**Always Encrypted** によって、列データが暗号化されたことを確認しておきましょう。

1. 次のように **SELECT** ステートメントを実行して、「**aeTest1**」テーブルの中身を参照します。

```
SELECT * FROM aeTest1
```



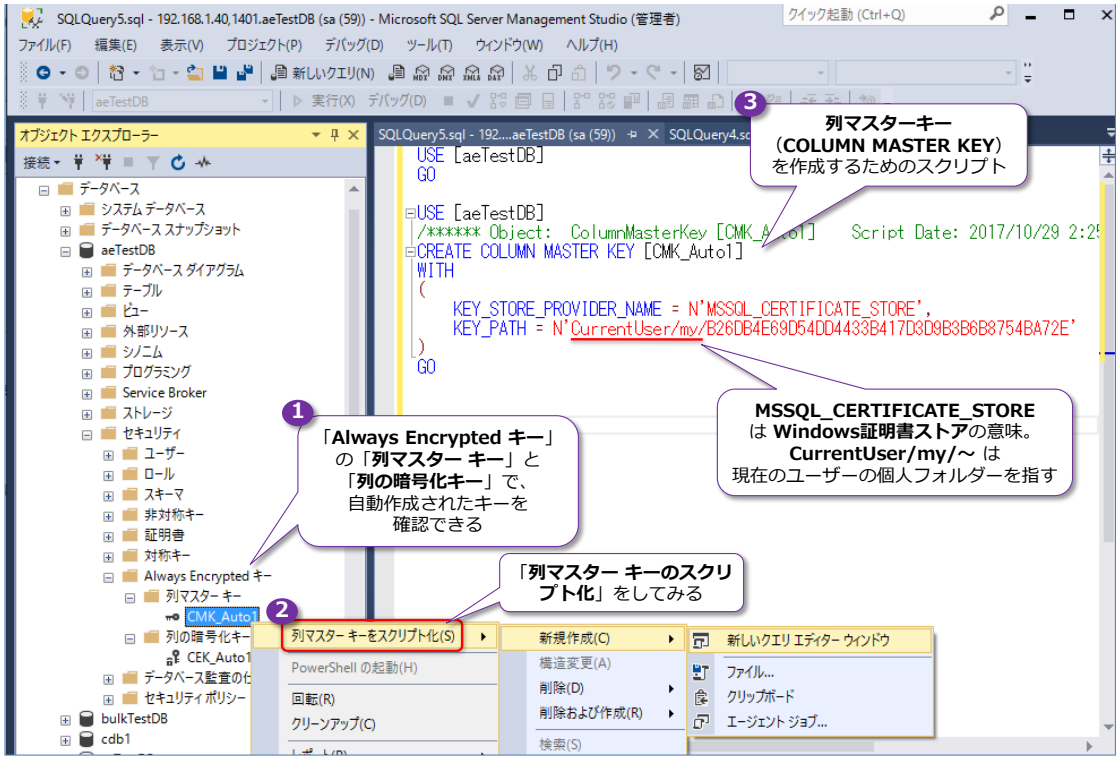
colB と **colC** 列が暗号化されていることを確認できます。このように、**Always Encrypted** を設定すると、列データを暗号化することができます。

このように、Always Encrypted ではデータベース内のデータそのものが暗号化されているので、前掲の TDE を利用したデータベースの暗号化を実施しなくても、データが暗号化されて

います（TDE は必要ありません）。また、ネットワーク上を流れるデータも、この暗号化されたデータがそのまま流れるので、前掲のネットワーク接続の暗号化を利用しなくても、ネットワーク上のデータが暗号化されています。

Always Encrypted では、詳しくは後述しますが、クライアント側（正確には SQL Server にアクセスするための接続モジュール）が暗号化および復号化を行うことで、SQL Server 側には暗号化されたデータのみしか配置しない、という状態を実現することができます。

2. 次に、ウィザードによって自動作成された**列マスター キー**（**COLUMN MASTER KEY**）と**列の暗号化キー**（**COLUMN ENCRYPTION KEY**）を確認してみます。次のようにオブジェクト エクスプローラーで［**セキュリティ**］フォルダーの［**Always Encrypted キー**］フォルダーを展開します。

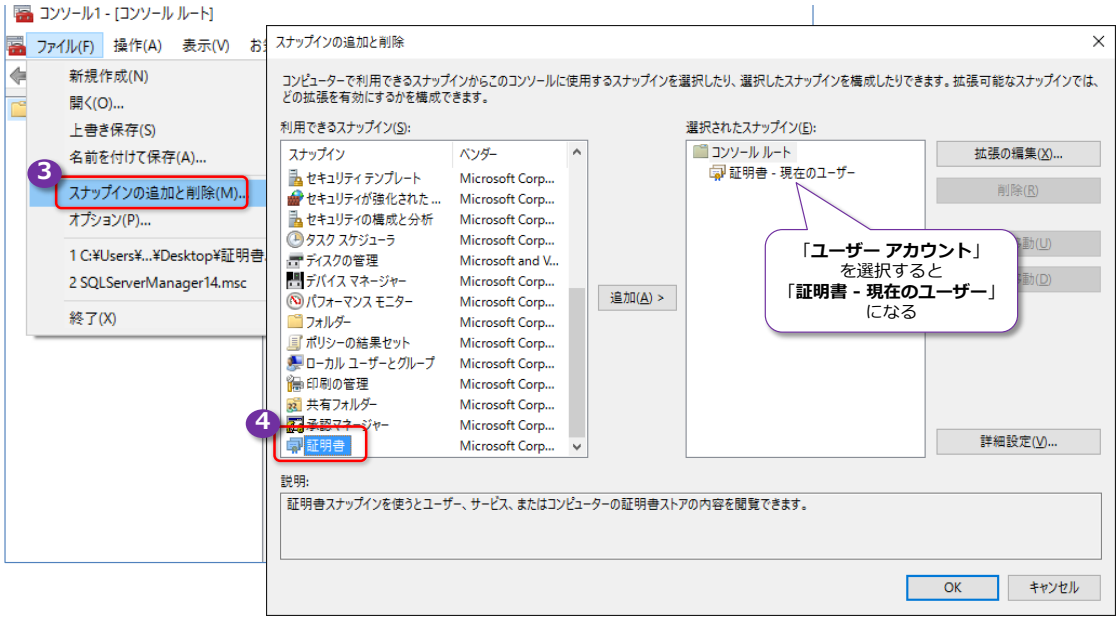


「**CMK_Auto1**」という名前の**列マスター キー**と、「**CEK_Auto1**」という名前の**列の暗号化キー**が作成されていることを確認できます。これらのキーは、右クリックして、［**～をスクリプト化**］からスクリプトを生成することで、ウィザードによってどのようにキーが作成されたのかを確認することができます。

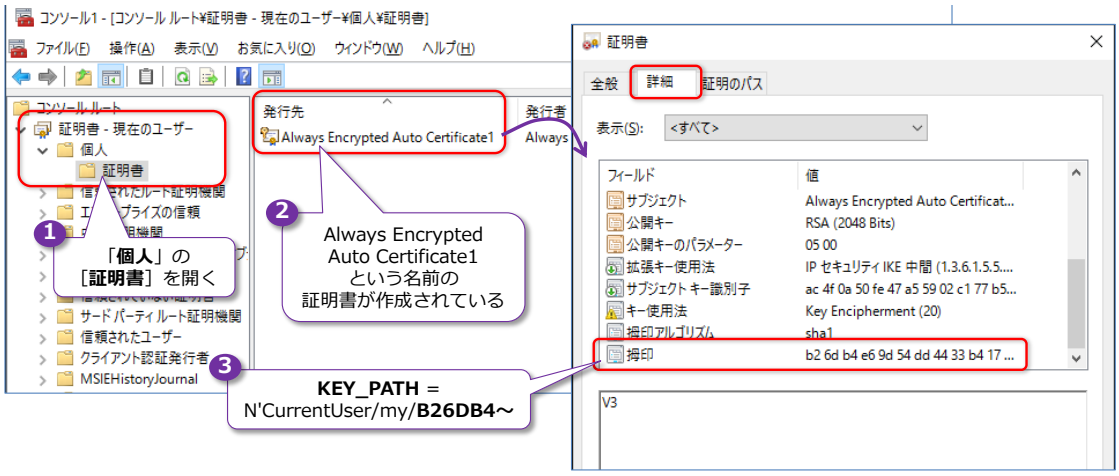
列マスター キー（**COLUMN MASTER KEY**）のスクリプトを生成すると、**KEY_STORE_PROVIDER_NAME**（キーを格納するプロバイダーの名前）に「**MSSQL_CERTIFICATE_STORE**」が設定されていますが、これは「**Windows 証明書ストア**」の意味で、**KEY_PATH**（キーのパス）の「**CurrentUser/my/～**」は、現在のユーザーの個人フォルダーを指していて、これがマスター キーが格納されている証明書へのパスになっています。

3. 次に、**証明書**（マスター キーの場所）を確認してみましょう。これを行うには、［**スタート**］ボタンを右クリックして、［**ファイル名を指定して実行**］から「**mmc**」と入力し、管理コンソールを起動して、［**ファイル**］メニューの［**スナップインの追加と削除**］で［**証明書**］スナップ

インを追加します（[**ユーザー アカウント**]を選択して追加します）。



［**証明書**］スナップインを追加したら、［**個人**］の［**証明書**］フォルダーを展開します（ここが **CurrentUser/My** というパスの場所になります）。



ここに「**Always Encrypted Auto Certificate1**」という名前の証明書が作成されていることを確認できます。**拇印**が、スクリプト生成で確認した **KEY_PATH** のパスになっています。

Always Encrypted では、この証明書（マスター キー）を利用して、データの暗号化を行っていますが、証明書は、Windows 上に作成されていて、Linux 上の SQL Server にはこの証明書がないのもポイントの 1 つです（詳しくは後述します）。

アプリケーションからの接続（Java、Python、PHP、C# など）

Always Encrypted を設定したテーブルに対して、**Java**（JDBC）や **Python**（pyodbc）、**PHP**、**ADO.NET** などのアプリケーションからアクセスするには、接続文字列に以下を追加することで、暗号化された列データを取得（復号化）できるようになります。

```
Java： JDBC (Microsoft JDBC Driver 6.2 for SQL Server 以上)
"columnEncryptionSetting=Enabled;"

Python： pyodbc (Microsoft ODBC Driver 13.1 for SQL Server 以上)
"ColumnEncryption=Enabled;"

ADO.NET (.NET Framework 4.6 以上)
"Column Encryption Setting=enabled;"

PHP： SQLSRV (PHP Driver for SQL Server 5.1.0-preview 以上と MSODBC 17 preview 以上)
"Driver" => "ODBC Driver 17 for SQL Server",
"ColumnEncryption" => "Enabled",
```

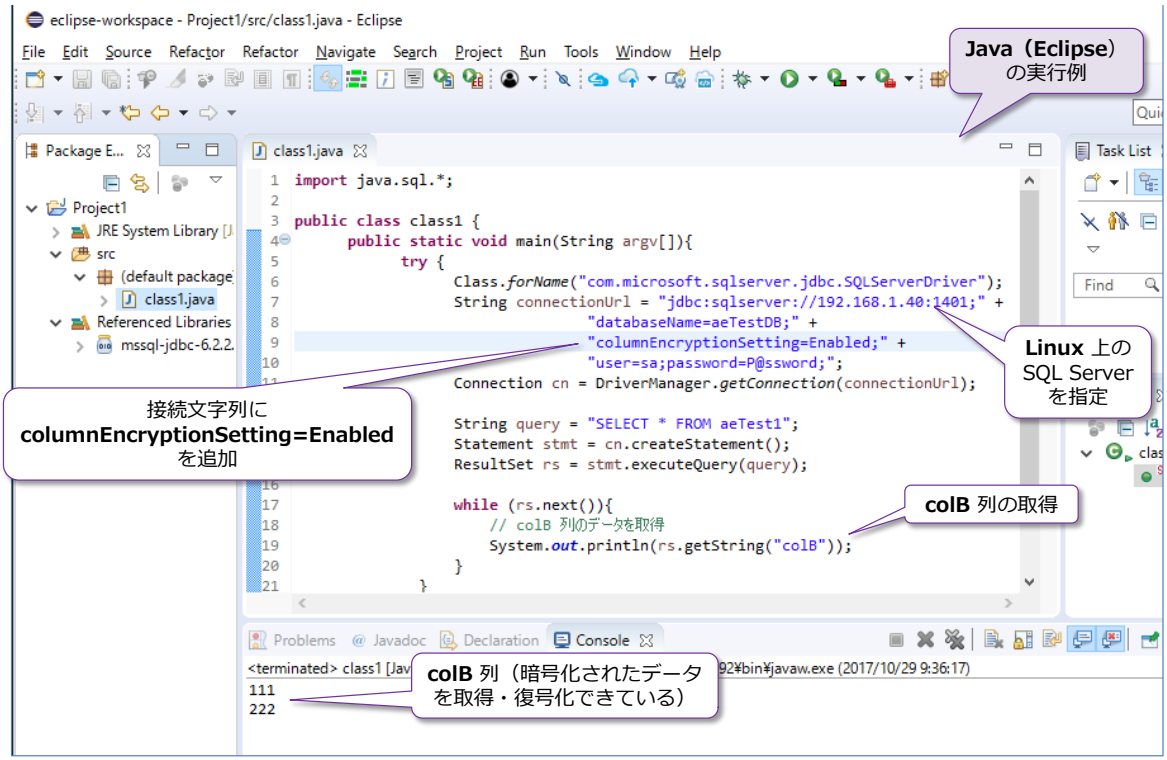
なお、**JDBC** の場合は、追加の手順が ２つあり、１つは JCE（Java Cryptography Extension）の無制限強度の管轄ポリシーファイルが必要で、JDK 1.8 用の場合は以下からダウンロードしておくようにします。

JCE の無制限強度の管轄ポリシーファイル（JDK 1.8 用）
http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html

.zip をダウンロードしたら、これを解凍して、「**US_export_policy.jar**」と「**local_policy.jar**」ファイルを、**%JAVA_HOME%¥jre¥lib¥security** フォルダに上書きコピーします。

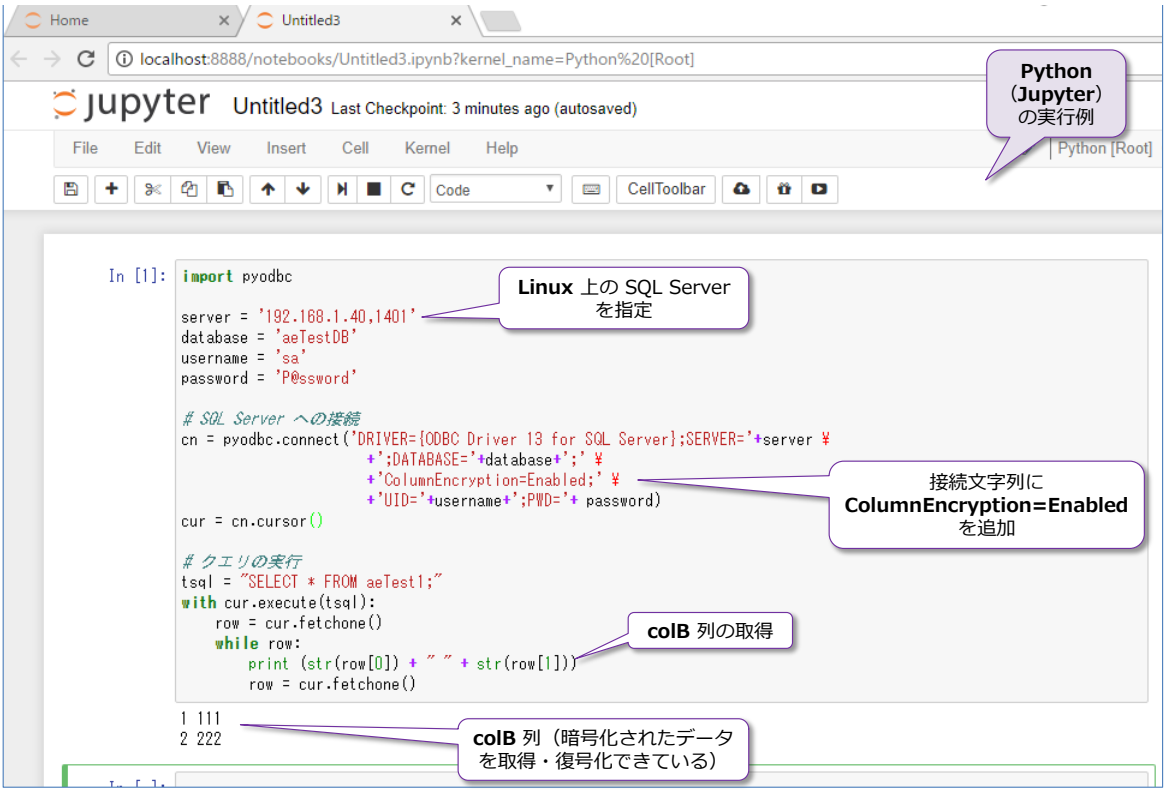
追加手順の ２つ目は、Microsoft JDBC Driver 6.2 に含まれる「**sqljdbc_auth.dll**」ファイルを **C:¥Windows¥system32** フォルダーにコピーしておくようにします。

これを行えば、接続文字列に、前掲の「**columnEncryptionSetting=Enabled**」を追加することで、Always Encrypted で暗号化したデータを取得（復号化）できるようになります。

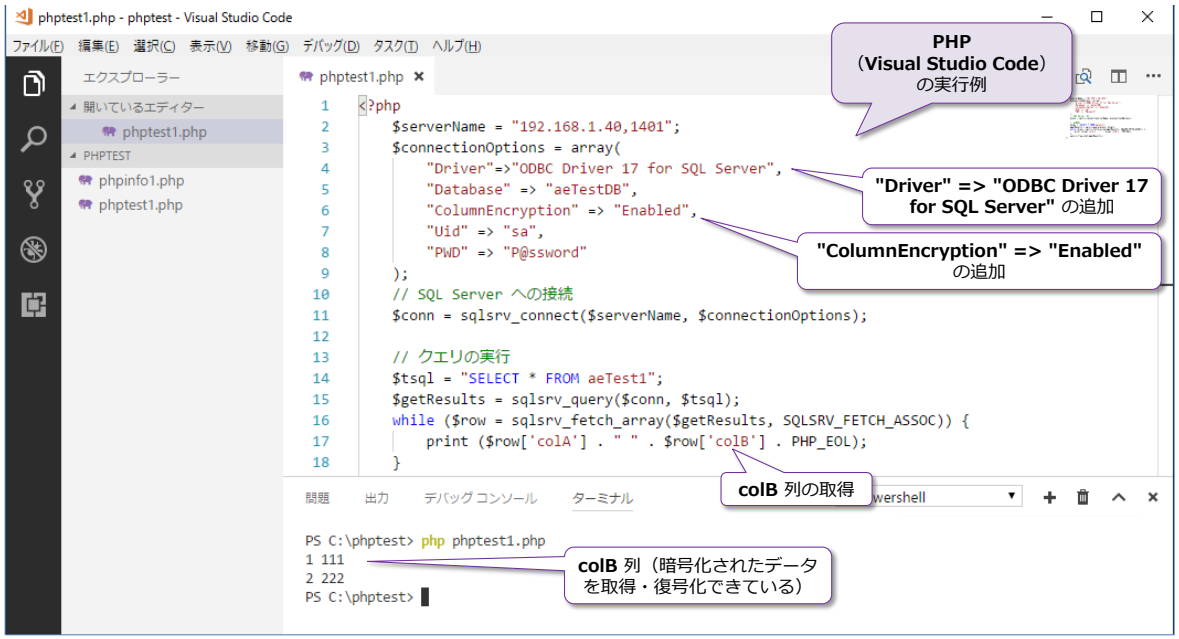


なお、接続文字列に「**columnEncryptionSetting=Enabled**」を追加していない場合は、次のように暗号化されたままのデータを取得する形になります。

Python（pyodbc）の場合は、次のように接続文字列に「**ColumnEncryption=Enabled**」を追加することで、Always Encrypted で暗号化されたデータを取得することができます。



PHP（SQLSRV）の場合は、**PHP Driver for SQL Server 5.1.0-preview** と **Microsoft ODBC Driver 17 for SQL Server preview**（どちらも執筆時点での最新のプレビュー版）が必要になりますが、接続文字列に「**"Driver" => "ODBC Driver 17 for SQL Server", "ColumnEncryption" => "Enabled",**」を追加することで、Always Encrypted で暗号化されたデータを取得することができます。

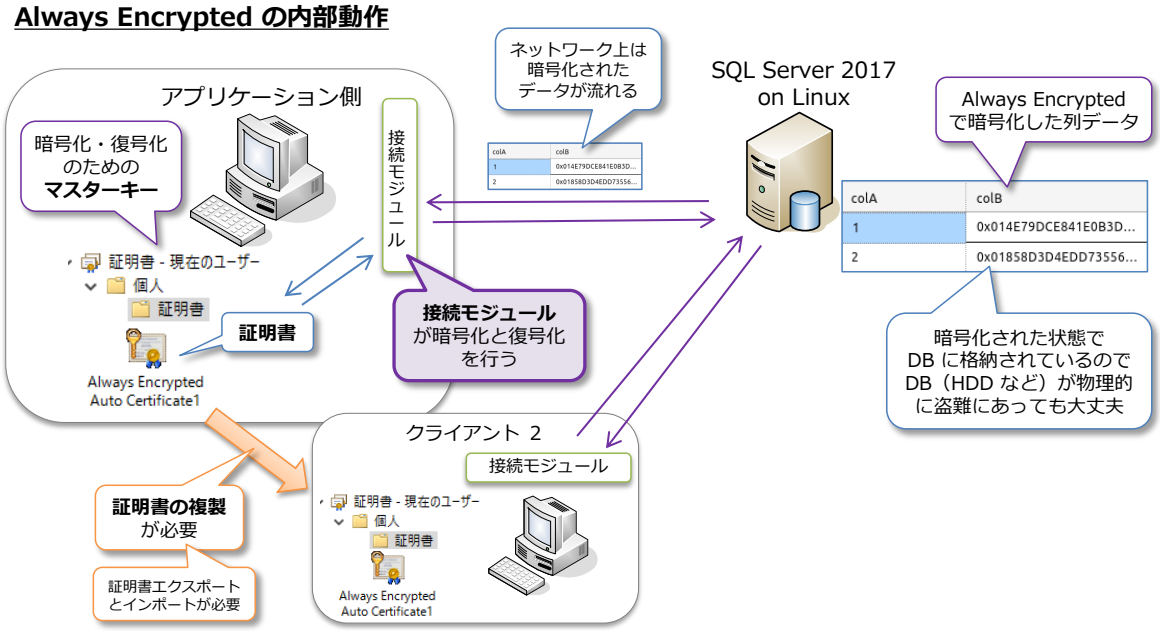


以上のように、Always Encrypted で暗号化したデータは、（基本的には）接続文字列を変更するだけで、そのほかのアプリケーション コードを変更することなく、データ取得（復号化）することができます。

Always Encrypted の内部動作（接続モジュールが暗号化／復号化を実施）

Always Encrypted では、クライアント（アプリケーション）側の「**接続モジュール**」が暗号化および復号化を行う仕組みになっています。

Always Encrypted の内部動作



このように、クライアント側で暗号化と復号化を行うので、SQL Server 側には暗号化されたデータのみが格納されることになり、SQL Server 上の HDD などのハードウェアが物理的な盗難にあったとしても、中身を参照できない状態になります（セキュアです）。また、ネットワーク上を流れ

るデータも暗号化されたものが流れているので、パケット盗聴対策にもなります。

一方で、**接続モジュール**や**証明書**を利用した機能ゆえに、接続モジュールのバージョンが非常にシビアになっています。具体的には、**接続モジュールの要件**が次のようになっています。

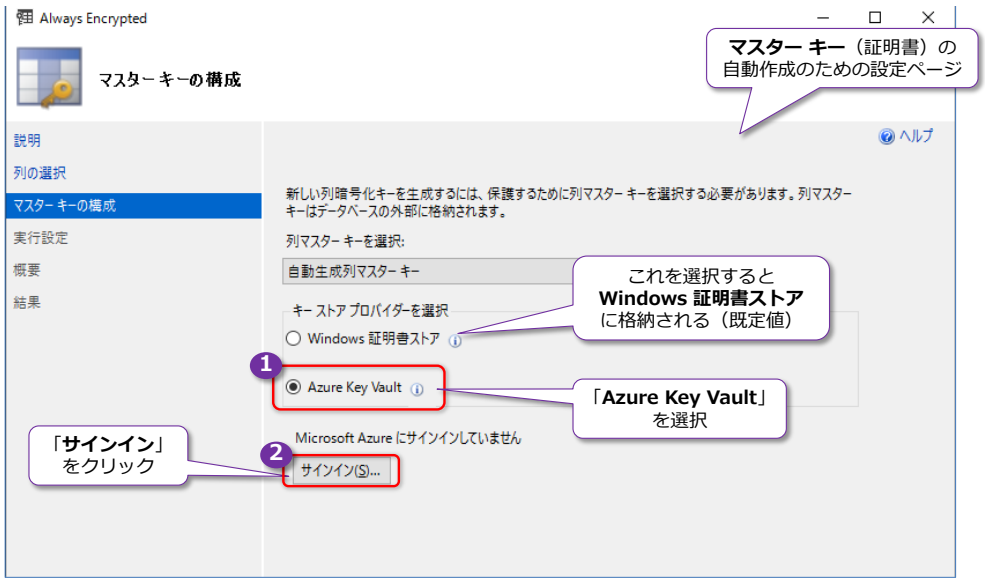
- Java： Microsoft JDBC Driver **6.2** for SQL Server 以上
- Python： Microsoft ODBC Driver **13.1** for SQL Server 以上
- ADO.NET： .NET Framework **4.6** 以上（.NET Core 2.0 は未対応）
- PHP： PHP Driver for SQL Server **5.1.0-preview** 以上
  かつ Microsoft ODBC Driver **17** preview 以上）

また、暗号化／復号化の**マスター キー**には**証明書**を利用しているので、クライアントが複数台ある場合には、それぞれのクライアントに**同じ証明書**を複製しておく必要があります（もちろん、PHP や ASP.NET、Python Django などの Web アプリケーションの場合であれば、Web サーバーにのみ証明書があれば大丈夫です）。

また、本自習書の手順では、**証明書**（マスターキー）の作成／格納場所として **Windows 証明書ストア**（**MSSQL_CERTIFICATE_STORE**）を利用しましたが、これは、現在のところ Linux では未サポートになっているので、Linux をクライアントとしたアプリケーションの場合は利用することができません。これに対応するためには、Microsoft Azure の「**Azure Key Vault**」（クラウド上に証明書／キーを保存できるサービス）などを利用するようにします。

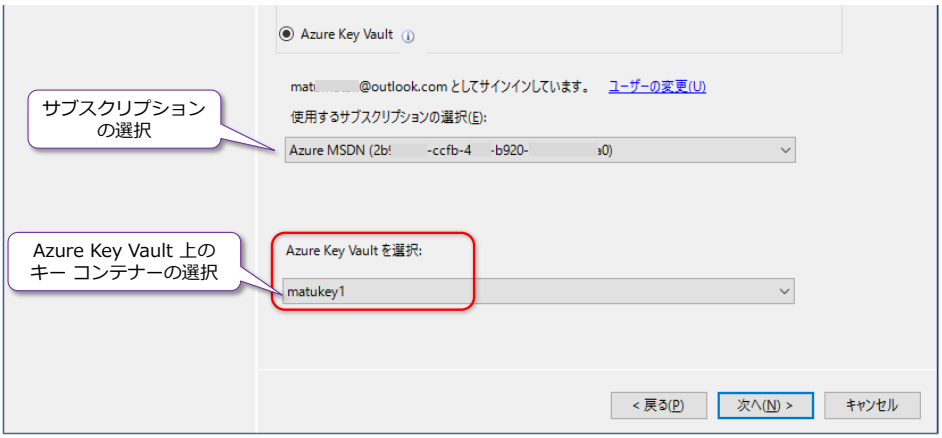
Azure Key Vault に証明書（マスタ キー）を格納する場合

Azure Key Vault に証明書を格納するには、Always Encrypted を設定するためのウィザードの［**マスター キーの構成**］ページで、次のように［**Azure Key Vault**］を選択します。

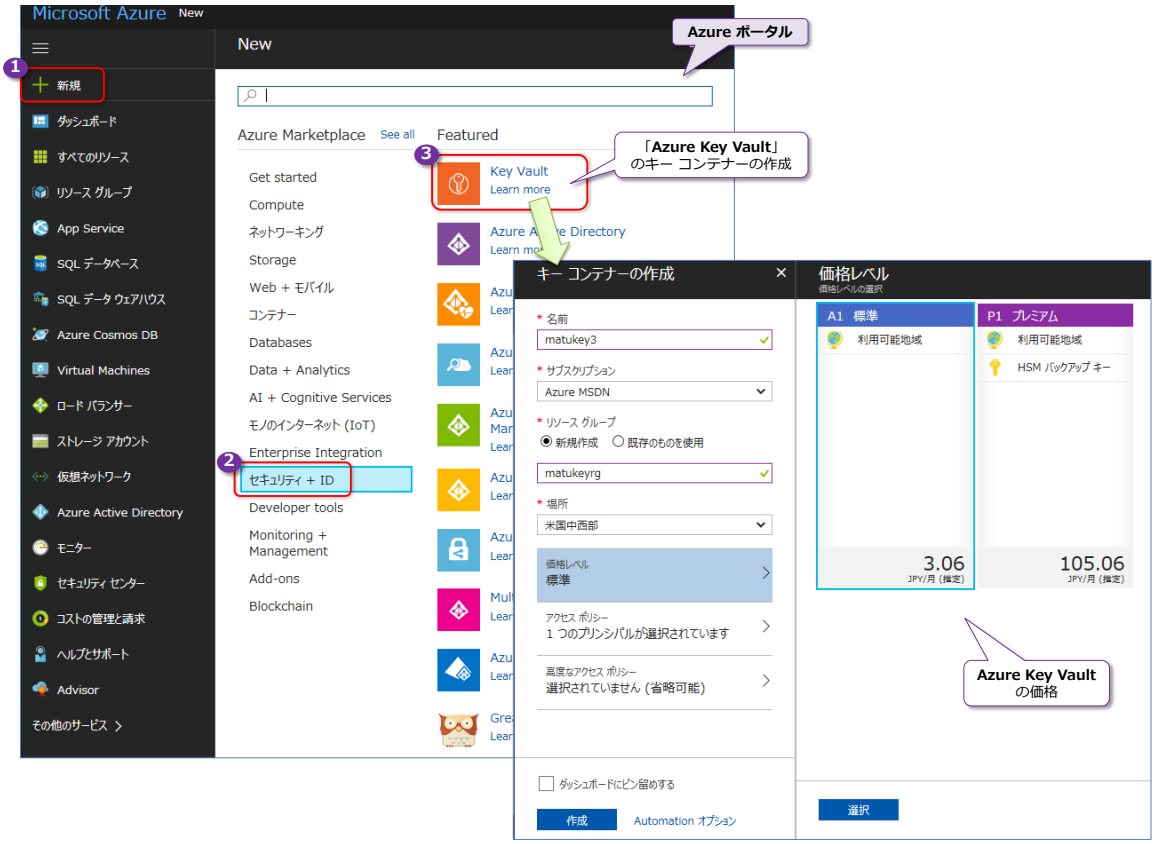


［**サインイン**］ボタンをクリックすると、Microsoft Azure へのサインイン画面が表示されるので、サインインを行います。これで、Azure Key Vault 上に作成した**キー コンテナー**を選択できるよ

うになります。



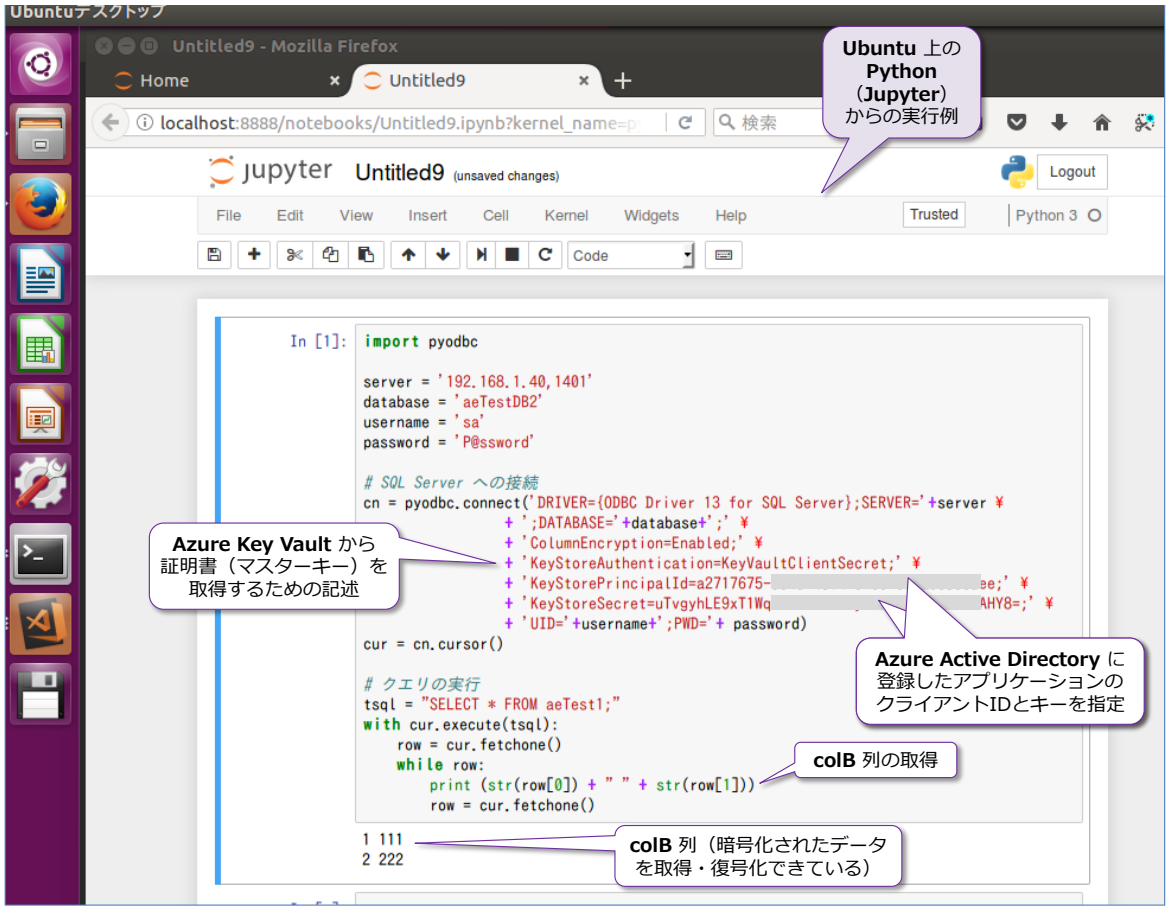
キー コンテナーは、次のように **Azure ポータル**から簡単に作成することができますが、PowerShell スクリプトから作成することもできます。



PowerShell スクリプトで作成する手順については、以下のドキュメントが参考になります（対象は Azure SQL Database ですが、SQL Server 2017 にもほとんどの内容が当てはまります）。

Always Encrypted: SQL Database で機密データを保護し Azure Key Vault で暗号化キーを格納
https://docs.microsoft.com/ja-jp/azure/sql-database/sql-database-always-encrypted-azure-key-vault

証明書（マスター キー）を Azure Key Vault 上のキー コンテナーに格納しておけば、Python の場合であれば、次のように Linux 上で動作するアプリケーションを記述することができます。



このように、**Python** (pyodbc) であれば、接続文字列に「**KeyStoreAuthentication=KeyVaultClientSecret**」と「**KeyStorePrincipalId=クライアント ID**」、「**KeyStoreSecret=クライアント ID のキー**」(クライアント ID は Azure Active Directory に登録したアプリケーションの ID) を追加することで、Azure Key Vault 上の証明書（マスター キー）を利用して、暗号化されたデータを取得（復号化）することができます。

なお、**PHP** の PHP Driver for SQL Server **5.1.0-preview** では、現在のところ Azure Key Vault に対応していないため、Azure Key Vault を利用したアプリケーションを Linux 上で利用することができません。また、**.NET Core 2.0** は、現在のところ Always Encrypted に対応していないため、.NET Core を利用したアプリケーションも Linux 上で利用することができません。

Java に関しては、GitHub 上の azure-sdk-for-java や azure-activedirectory-library-for-java を利用しますが、以下のドキュメントが参考になると思います。

JDBC ドライバーで Always Encrypted を使用する
https://docs.microsoft.com/ja-jp/sql/connect/jdbc/using-always-encrypted-with-the-jdbc-driver

以上のように、現状、SQL Server 2017 on Linux 上の Always Encrypted を利用するにあたっては、クライアントとして Windows を利用する場合や、Windows 上の Web アプリケーション（PHP、ASP.NET、Python Django など）を介して利用する場合には、**Windows 証明書ストア**を利用することができるので、言語の選択肢が多いのですが（Java でも、Python でも、PHP でも、C# でも、VB でもアプリケーションを開発できます)、Linux をクライアントとして考えた場

合は、選択肢が狭くなってきます。

Always Encrypted は、SQL Server 2016 から提供された機能で、これからどんどん進化していくことが予想されるので、接続モジュールの新しいバージョンが出たときには、対応状況が変わってくる部分が出てくるので、ぜひそれにも期待してみてください。

もちろん、Always Encrypted を利用しなくても、前掲の TDE（透過的なデータ暗号化）を利用すれば、HDD などの物理的な盗難対策になりますし、ネットワーク接続の暗号化を利用すれば、パケット盗聴への対策ができるので、アプリケーションの要件や性能要件などと比較しながら、どのセキュリティ機能を利用するのが良いかを検討してみてください。

その他、Always Encrypted に関しては、以下のドキュメントが参考になります。

PHP Driver for SQL Server 5.1.0-preview
https://github.com/Microsoft/msphpsql/releases/tag/v5.1.0-preview

Using Always Encrypted with the ODBC Driver 13.1 for SQL Server
https://docs.microsoft.com/en-us/sql/connect/odbc/using-always-encrypted-with-the-odbc-driver

Azure Key Vault の価格
https://azure.microsoft.com/ja-jp/pricing/details/key-vault/

Always Encrypted (データベース エンジン)
https://docs.microsoft.com/ja-jp/sql/relational-databases/security/encryption/always-encrypted-database-engine

おわりに

最後まで試された皆さん、いかがでしたでしょうか？ SQL Server 2017 on Linux でも、ほとんどの操作を Windows 版の SQL Server と同じように利用できることを確認できたのではないでしょうか。

SQL Server がついにマルチ プラットフォーム化して、Mac OS や Linux でも動かせるようになったのは本当に衝撃です。Visual Studio Code を利用すれば、Windows を利用することなく、Mac OS や Linux だけで SQL Server への接続／各種の操作が可能で、さらにはアプリケーション開発（Java や Python、PHP、C#、ASP.NET、node.js など）もできてしまいます。

SQL Server 2017 on Linux は、ただ単に Linux 上で SQL Server を動かせるようにしただけはなく、ほとんどの SQL Server の機能（データベース エンジンに関する機能）を、Windows 上の SQL Server とまったく同じように利用することができます。本自習書では扱っていないものとしては、SQL Server 2017 からの新機能である**自動チューニング**や**グラフ データベース**などがあり、これも同じように利用できますし、**クエリ ストア**も、**データ パーティション**も、**インメモリ OLTP** も全く同じスクリプトで利用することができます。

もし、Windows 上の Management Studio から Linux 上の SQL Server をリモート操作できるなら、Linux 上で動作させているのを忘れてしまうぐらい、まったく同じように GUI で操作することができます。

SQL Server 2017 には、まだまだたくさんの新機能が追加されているので、そういった新機能については、本自習書シリーズの No.1「**SQL Server 2017 の新機能の概要**」編で詳しく説明しているので、こちらもぜひご覧いただければと思います。

SQL Server 2017 では、**Python** を**データベース エンジンに統合**（ビルトイン）したことによって、**ディープ ラーニング**（**GPU** 利用も OK）も可能になりました。これについては、本自習書シリーズの No.3「**SQL Server 2017 Machine Learning Services**」編（現在制作中）で詳しく説明しているので、こちらもぜひご覧いただければと思います。

執筆者プロフィール

有限会社エスキューエル・クオリティ（http://www.sqlquality.com/）

SQLQuality（エスキューエル・クオリティ）は、**日本で唯一の SQL Server 専門の独立系コンサルティング会社**です。過去のバージョンから最新バージョンまでの SQL Server を知りつくし、多数の実績と豊富な経験を持つ、OS や .NET にも詳しい **SQL Server の専門家（キャリア 20 年以上）がすべての案件に対応します**。人気メニューの「**パフォーマンス チューニング サービス**」は、100%の成果を上げ、過去すべてのお客様環境で驚異的な性能向上を実現。チューニング スキルは**世界トップレベル**を自負、検索エンジンでは（英語情報を含めて）ヒットしないノウハウを多数保持。ここ数年は **BI／DWH システム構築支援**のご依頼が多く、支援だけでなく実際の構築も行う。

主なコンサルティング実績／構築実績

- 大手製造業の「**CAD 端末の利用状況の見える化**」システム構築
  Oracle や CSV（Notes）、TSV ファイル、Excel からデータを抽出し、SQL Server 2012 上に DWH を構築
  見える化レポートには Reporting Services を利用
- 大手映像制作会社の **BI システム構築**（会計／業務システムにおける予実管理／原価管理など）
  従来 Excel で管理していたシートを Reporting Services のレポートへ完全移行。
  Oracle や勘定奉行からデータを抽出して、SQL Server 上に DWH を構築
- 大手流通系の **DWH／BI システム構築支援**（POS データ／在庫データ分析／ABC 分析／ポイントカード分析）
- 大手アミューズメント企業の **BI システム構築支援**（人事システムにおける人材パフォーマンス管理）
  Reporting Services による勤怠状況の見える化レポートの作成、PostgreSQL／人事システムからのデータ抽出
- 外資系医療メーカーの **BI システム**構築支援（Analysis Services と Excel による販売分析システム）
  OLAP キューブによる売上および顧客データの多次元分析／自由分析（ユーザーによる自由操作が可能）
- 大手流通系の **DWH システム**のパフォーマンス チューニング
  データ量 100 億件の DWH、**総ステップ数 2 万**越えのストアド プロシージャのパフォーマンス チューニング
- ミッション クリティカルな**金融システム**でのトラブル シューティング／定期メンテナンス支援
- SQL Server の下位バージョンからの**移行／アップグレード**支援（32 ビットから x64 への対応も含む）
- 複数台の SQL Server の **Hyper-V 仮想環境**への移行支援（サーバー統合支援）
- ハードウェア リプレース時の**ハードウェア選定**（最適なサーバー、ストレージの選定）、**高可用性環境**の構築
- **2 時間**かかっていた日中バッチ実行時間を、わずか **5 分**へ短縮（**95.8%** の性能向上）
- **Java 環境**（Tomcat、Seasar2、S2Dao）の SQL Server パフォーマンス チューニング etc

コンサルティング時の作業例（パフォーマンス チューニングの場合）

- アプリケーション コード（VB、C#、Java、ASP、VBScript、VBA）の解析／改修支援
- ストアド プロシージャ／ユーザー定義関数／トリガー（Transact-SQL）の解析／改修支援
- インデックス チューニング／SQL チューニング／ロック処理の見直し
- 現状のハードウェアで将来のアクセス増にどこまで耐えられるかを測定する高負荷テストの実施
- IIS ログの解析／アプリケーション ログ（log4net／log4j）の解析
- ボトルネック ハードウェアの発見／ボトルネック SQL の発見／ボトルネック アプリケーションの発見
- SQL Server の構成オプション／データベース設定の分析／使用状況（CPU, メモリ, ディスク, Wait）解析
- 定期メンテナンス支援（インデックスの再構築／断片化解消のタイミングや断片化の事前防止策など）etc

松本美穂（まつもと・みほ）

有限会社エスキューエル・クオリティ 代表取締役 Microsoft MVP for SQL Server（2004 年 4 月～）
経産省認定データベース スペシャリスト／MCDBA／MCSD for .NET／MCITP Database Administrator
SQL Server の日本における最初のバージョンである「SQL Server 4.21a」から SQL Server に携わり、現在、SQL Server を中心とするコンサルティングを行っている。得意分野はパフォーマンス チューニングと Reporting Services。著書の『SQL Server 2000 でいってみよう』と『ASP.NET でいってみよう』（いずれも翔泳社刊）は、トップ セラー（前者は 28,500 部、後者は 16,500 部発行）。近刊に『SQL Server 2016 の教科書』（ソシム刊）がある。

松本崇博（まつもと・たかひろ）

有限会社エスキューエル・クオリティ 取締役 Microsoft MVP for SQL Server（2004 年 4 月～）
経産省認定データベース スペシャリスト／MCDBA／MCSD for .NET／MCITP Database Administrator
SQL Server の BI システムとパフォーマンス チューニングを得意とするコンサルタント。アプリケーション開発（ASP／ASP.NET、C#、VB 6.0、Java、Access VBA など）やシステム管理者（IT Pro）経験もあり、SQL Server だけでなく、アプリケーションや OS、Web サーバーを絡めた、総合的なコンサルティングが行えるのが強み。Analysis Services と Excel による BI システムも得意とする。マイクロソフト認定トレーナー時代の 1998 年度には、Microsoft CPLS トレーナー アワード（Trainer of the Year）を受賞。