Excel & Power BI データ変換マスターガイド

Power Query, Power FX, DAXを極める

目次

第6章: DAX入門

- 6.1 DAXの概念と基本構文
- 6.2 計算列と計算メジャー
- 6.3 基本的な集計関数
- 6.4 日付と時間の計算
- 6.5 ExcelとPower BIでのDAX

第7章: DAXによる高度なデータ分析

- 7.1 コンテキストの理解
- 7.2 フィルター関数の活用
- 7.3 時間インテリジェンス
- 7.4 複雑な集計とKPI
- 7.5 DAXパターンとベストプラクティス

第8章: データモデルの最適化

- 8.1 効率的なデータモデル設計
- 8.2 Power Queryクエリの最適化
- 8.3 DAX式の最適化
- 8.4 メモリ使用量の削減テクニック
- 8.5 大規模データセットでのパフォーマンス向上策

第9章: データ更新と自動化

- 9.1 データ更新の仕組み
- 9.2 Excelでの自動更新設定
- 9.3 Power BIのリフレッシュオプション
- 9.4 増分更新の設定
- 9.5 スケジュールと監視

第10章: 実践的なユースケースとソリューション

- 10.1 財務データ分析シナリオ
- 10.2 販売データのダッシュボード作成
- 10.3 マスターデータ管理
- 10.4 複数ソースの統合レポート
- 10.5 モデルの共有と展開

付録

- A. Power Query関数クイックリファレンス
- B. DAX関数クイックリファレンス
- C. よくあるエラーと解決法
- D. 便利なリソースとコミュニティ

第1章: データ変換技術の基礎知識

1.1 Power Query、Power FX、DAXとは何か

データ分析の世界では、生のデータを意味のある情報に変換するプロセスが極めて重要です。MicrosoftのExcelとPower BIには、このプロセスを強力にサポートする3つの主要技術があります：Power Query、Power FX、DAXです。

Power Query

データの取得と変換を担当するETL（Extract, Transform, Load）エンジン

Power Queryは、データの接続、クリーニング、変形、結合を行うためのデータ準備ツールです。様々なデータソースからデータを取得し、分析に適した形に整えることができます。Power Queryは、直感的なユーザーインターフェイスと、背後で動作するM言語（Power Query式言語）で構成されています。

> 📘 **ベテランの知恵袋**
> Power Queryは「データ前処理の魔術師」とも呼ばれています。データ分析作業の約70〜80%は前処理に費やされるとも言われており、Power Queryはその大部分を自動化できる強力なツールです。画面操作だけでも多くのことができますが、M言語を少し理解するだけで可能性が何倍にも広がります！

Power FX

Excelライクな数式言語でロジックを実装するための計算エンジン

Power FXはExcel関数に似た構文を持つ、低コードプログラミング言語です。Power Appsで広く使われていますが、Power BIのいくつかの機能でも利用されています。Excel関数に慣れた人にとって直感的に使える設計になっています。

DAX (Data Analysis Expressions)

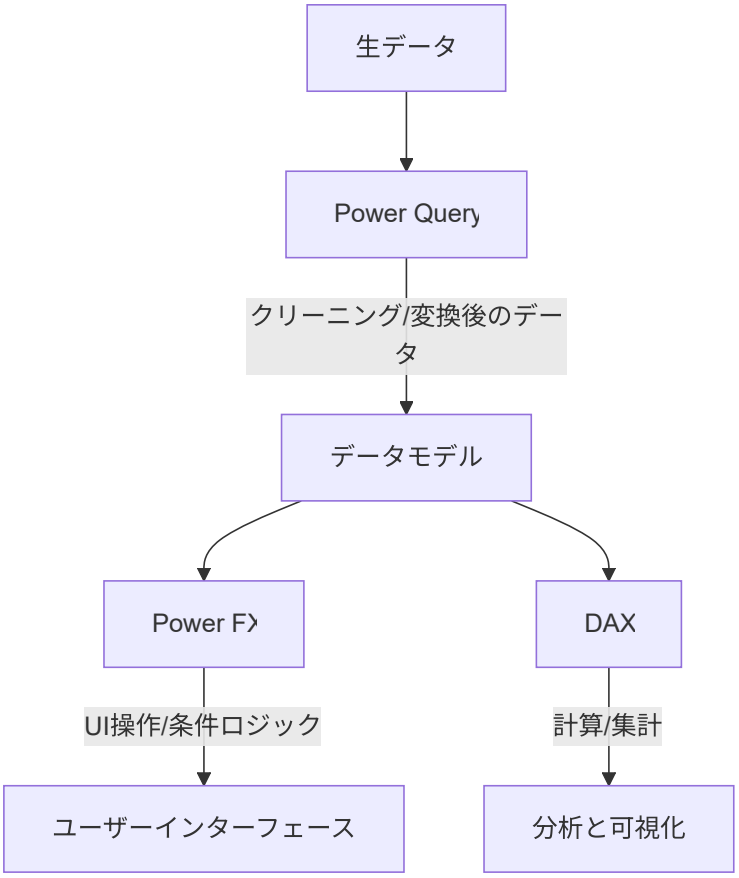
データモデル上での計算と分析のための数式言語

DAXはPower BIとExcelのPower Pivotで使用される数式言語で、計算列や計算メジャー（集計値）を作成できます。リレーショナルデータモデル上で動作し、高度なビジネス計算や複雑な集計を可能にします。

> ❓ **若手の疑問解決**
> **Q**: これら3つの技術はどれも似たように見えますが、どう使い分ければいいのでしょうか？
> **A**: 大まかな役割分担は次のとおりです：
>
> - **Power Query** → データを「準備する」段階（クリーニング、変形、結合など）
> - **Power FX** → アプリのロジックを「制御する」段階（主にPower Appsで使用）
> - **DAX** → 準備したデータを「分析する」段階（集計、KPI計算など）

1.2 それぞれの役割と違い

3つの技術の違いを明確に理解することで、適切なタイミングで適切なツールを選択できるようになります。



処理のタイミング

- **Power Query**: データロード時に一度だけ実行される前処理
- **Power FX**: 主にユーザーのアクション（クリック、選択など）に応じて実行
- **DAX**: レポートの表示・操作時に動的に実行される計算

主な用途

- **Power Query**: データソース接続、データクリーニング、テーブル結合、型変換
- **Power FX**: UI要素の制御、条件付き書式、アプリのロジック
- **DAX**: 集計計算、KPI計算、時系列分析、フィルタリングされたデータの集計

言語の特徴

Power Query（M言語）:

- 関数型プログラミング言語
- ステップバイステップのデータ変換を記録
- クエリ全体が一度に評価される（遅延評価）

Power FX:

- Excel関数に似た構文

- 宣言型言語（何をするかを記述）
- 即時評価される

DAX:

- 列指向の計算言語
- コンテキスト（行、フィルター）に依存
- リレーショナルデータモデル上で動作

> 💡 **ベテランの知恵袋**
> DAXの最も難しい概念は「コンテキスト」です。計算がどのコンテキストで実行されるかを理解すると、DAXの多くの謎が解けます。最初は混乱するかもしれませんが、慣れてくると非常に強力なツールになります。

1.3 Excelと Power BI における実装の違い

ExcelとPower BIはどちらもMicrosoftの製品ですが、これらの技術の実装方法には違いがあります。

Power Query

機能・特徴	Excel	Power BI
呼び名	データの取得と変換 / Power Query	Power Query エディター
アクセス方法	データタブ → 「データの取得と変換」グループ	ホーム → データの変換
ユーザーインターフェース	同じエディター（若干の違いあり）	同じエディター（若干の違いあり）
高度な機能	一部制限あり	すべての機能が利用可能
データの保存	ブックにクエリが保存される	.pbixファイルに埋め込まれる
リフレッシュ	手動または自動（制限あり）	手動、スケジュール、ゲートウェイ連携

DAX

機能・特徴	Excel	Power BI
利用可能な場所	Power Pivot（データモデル）	データビュー、レポートビュー
機能範囲	基本的な関数のみ	すべての関数が利用可能
UI	数式バーとダイアログ	数式バーとダイアログ
メジャーの作成	Power Pivotタブから	データペインから
計算列の作成	データビューから	データビューから

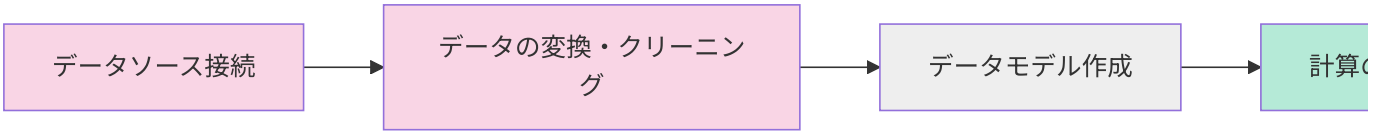
Power FX

- ExcelではPower FXは実装されていない（Excelの通常の関数は異なる）
- Power BIでは、計算列やPower BI Desktopの一部機能で限定的に使用

> 🚫 **失敗から学ぶ**
> よくある誤解として、「ExcelのDAXとPower BIのDAXは同じ」というものがあります。基本的な概念は同じですが、Power BIのDAXの方がより多くの関数を提供しており、より最適化されています。Excelで動作するDAX式がPower BIでも100%同じように動作するとは限らないことを覚えておきましょう。

1.4 データ変換の基本的な流れ

効率的なデータ分析のワークフローは、一般的に以下のステップで構成されます：



1. データソース接続

Power Queryを使用して、様々なデータソース（Excel, CSV, データベース, Web, APIなど）に接続します。

2. データの変換・クリーニング

Power Queryで以下の操作を行います：

- 不要な行・列の削除
- データ型の変更
- 列の分割・結合
- 欠損値の処理
- 重複の除去
- テーブルの結合・マージ

3. データモデル作成

テーブル間のリレーションシップを設定し、スターまたはスノーフレークスキーマを構築します。

4. 計算の追加

DAXを使用して：

- 計算列（特定の行に対する計算）
- メジャー（集計や複雑な計算）
- 計算テーブル（既存のテーブルから派生した新しいテーブル）
  を作成します。

5. レポート・ダッシュボード作成

ビジュアルやレポート要素を配置し、インタラクティブなダッシュボードを作成します。

6. 共有と展開

完成したレポートを共有または展開します。

> 💼 **プロジェクト事例**
> ある製造業のデータ分析プロジェクトでは、以下のように各技術を活用しました：
>
> - **Power Query**：複数の生産ラインからのCSVデータと社内DBからの品質データを統合し、クリーニングしました
> - **DAX**：製品ごとの不良率、ライン稼働率、生産効率のKPIを計算しました
> - **レポート**：経営層向けのダッシュボードを作成し、問題のある生産ラインを素早く特定できるようにしました
>
> この取り組みにより、以前は週次レポート作成に2日かかっていた作業が自動化され、リアルタイムの意思決定が可能になりました。

✅ 第1章 チェックリスト

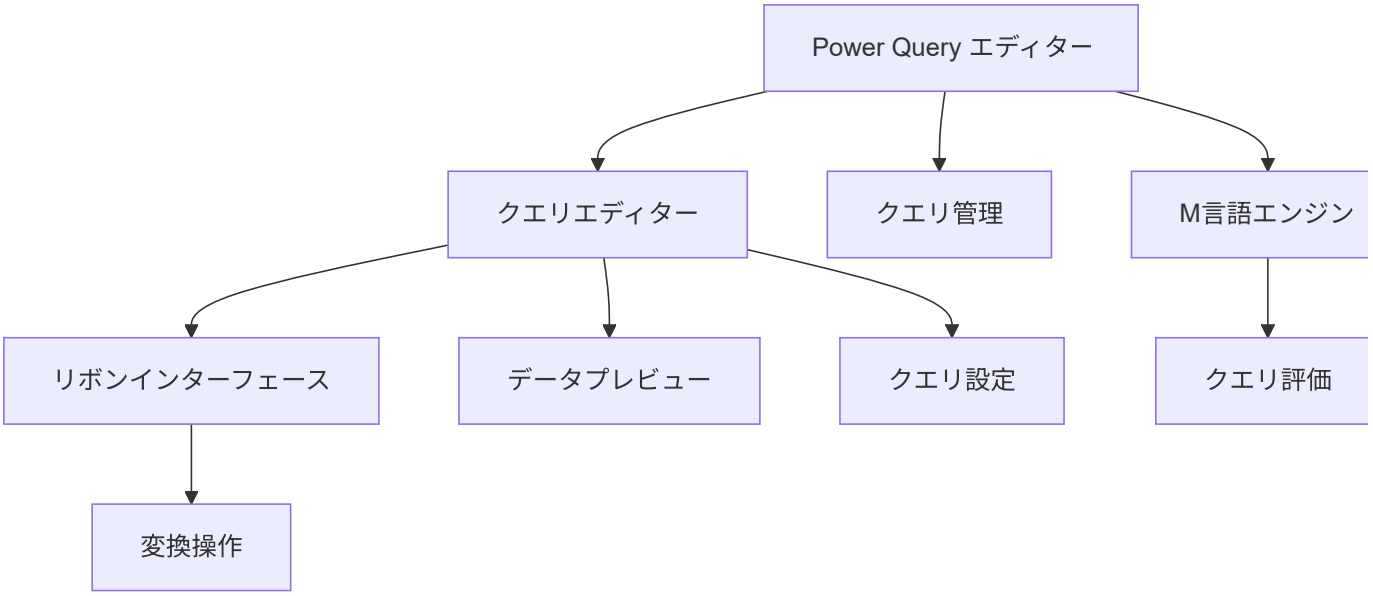
- [ ] Power Query、Power FX、DAXの基本的な役割を理解した
- [ ] 3つの技術の違いを説明できる
- [ ] ExcelとPower BIにおける実装の違いを把握した
- [ ] データ変換の基本的な流れを理解した
- [ ] 適切な状況で適切なツールを選択できる

第2章: Power Query 入門

2.1 Power Queryの基本概念

Power Queryは「データの取得と変換」のためのツールであり、Excelと Power BIに統合されています。データを分析に適した形に整える作業を大幅に効率化してくれます。

Power Queryの主要コンポーネント



1. **Power Query エディタ**: データ変換操作を行うためのグラフィカルインターフェース
2. **クエリ**: 一連のデータ変換ステップを記録したレシピ
3. **M言語**: 背後で動作するPower Query式言語
4. **データソース**: 接続可能な様々なデータ形式やサービス
5. **データ型**: テキスト、数値、日付、真偽値など各列に設定される型情報

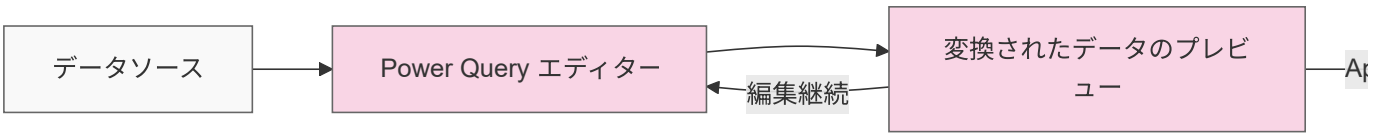
Power Queryの処理モデル

Power Queryの重要な特徴として、その処理モデルがあります：

1. **宣言型処理**: 何をするかを指定し、具体的な実行方法はエンジンに任せる
2. **遅延評価**: 実際にデータがロードされるまで変換は実行されない
3. **クエリの折りたたみ**: 可能な場合、複数の変換ステップをデータソースへの単一のクエリにまとめる
4. **ステップの連鎖**: 各変換操作が前のステップの結果に適用される

> 💡 **ベテランの知恵袋**
> Power Queryの最も強力な特徴の一つは「遅延評価」です。これにより、大量のデータに対しても効率的に処理できます。実際のデータはクエリをロードする時点まで取得されないので、変換ステップを定義している段階では少量のサンプルデータだけを見ながら作業できます。

データの流れ



Power Queryでは、データソースに接続して変換を加え、最終的に結果をテーブルやデータモデルにロードします。一度クエリを作成すれば、データソースが更新されても同じ変換手順が適用されるため、繰り返し作業を自動化できます。

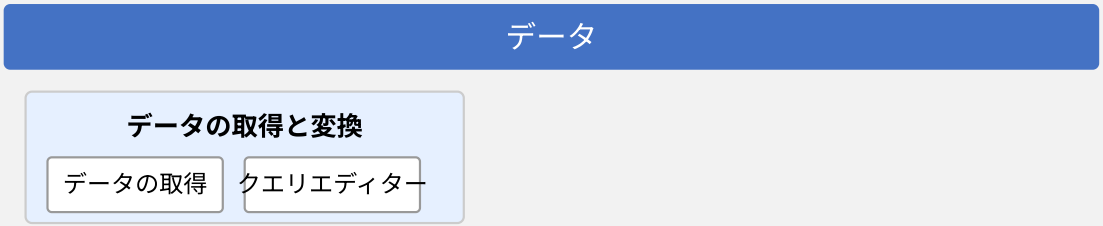
2.2 Excelでの利用方法

Excelの新しいバージョンでは、Power Queryは「データの取得と変換」という名前で提供されています。

Power Queryへのアクセス方法

Excel 2016以降:

1. 「データ」タブをクリック
2. 「データの取得と変換」グループにあるオプションを使用



基本的な使用方法

1. **データを取得**:
   - 「データの取得」→データソースを選択（Webサイト、CSV、Excel、データベースなど）
   - 接続情報を入力し、データをプレビュー
2. **Power Query エディタを使用**:
   - データをプレビューした後、「編集」ボタンをクリックしてPower Query エディタを開く
   - または、既存のクエリを「クエリ エディタ」ボタンで開く
3. **データを変換**:
   - エディタ内のリボンにあるコマンドを使用してデータを変換
   - 適用した変換はステップとして記録される
4. **データをロード**:
   - 「閉じて読み込む」でデータをExcelワークシートにテーブルとして読み込み
   - または「閉じてロード先」で読み込み先オプション（テーブル、ピボットテーブル、データモデルなど）を選択

> ❓ **若手の疑問解決**
> **Q**: Power Queryでデータを変換すると元のデータファイルが変更されてしまいますか？
> **A**: いいえ、Power Queryは元のデータソースを変更せず、読み取り専用でアクセスします。変換はExcel内に保存され、元のデータは変わりません。これはデータのコピーを変換していると考えるとわかりやすいでしょう。

Excelでの便利な機能

- **クエリ管理**:
  - 「データ」タブ→「クエリと接続」で全クエリを表示・管理
  - クエリ名の変更、編集、複製、削除が可能
- **クエリのリンク**:
  - あるクエリの結果を別のクエリの入力として使用
  - 変換手順を分割して管理しやすくする
- **パラメータ**:
  - 「データの取得と変換」→「パラメータ」→「新しいパラメータ」
  - クエリ内で参照可能な変数を定義（ファイルパス、URL、フィルタ値など）

2.3 Power BIでの利用方法

Power BIでも同様のPower Query機能が利用できますが、ユーザーインターフェースや一部の機能に違いがあります。

Power Queryへのアクセス方法

Power BI Desktop:

1. 「ホーム」タブで「データの変換」をクリック
2. または「データの取得」で新しいデータソースに接続する際に自動的に開く

Power BIの主な特徴

- **クエリエディタ名**:「Power Query エディタ」と明示的に呼ばれる
- **モデリングとの統合**: 変換したデータは自動的にデータモデルに読み込まれる
- **高度な機能**: より多くのコネクタやデータ変換機能が利用可能
- **クエリの管理**: 「データの変換」をクリック→サイドバーに全クエリが表示される

Power BI固有の機能

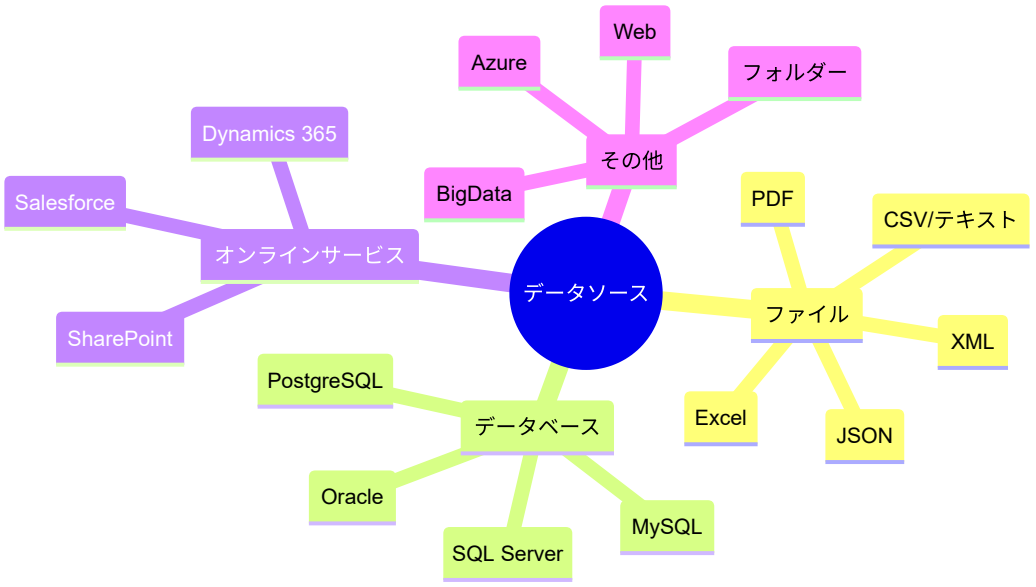
- **DirectQuery モード**: データを読み込まずにデータソースに直接クエリを発行
- **データゲートウェイ連携**: オンプレミスデータへの安全な接続
- **データフロー**: Power BI サービスでの共有可能なPower Queryクエリ
- **増分リフレッシュ**: 常に全データではなく変更部分のみを更新

> 🚫 **失敗から学ぶ**
> ある企業では、大量のデータをPower BIにインポートモードで取り込み、パフォーマンスが著しく低下する問題が発生しました。原因は必要なデータだけをフィルタリングせず、すべてのデータを取り込んでいたことでした。Power Queryで早い段階でフィルタを適用することで、必要なデータのみを取り込むように修正し、パフォーマンスが劇的に改善しました。

2.4 データソースへの接続

Power Queryの大きな強みは、多種多様なデータソースに接続できることです。

主なデータソース



接続の基本手順

1. **データソースの選択**:
   - Excel: 「データ」→「データの取得」→データソースを選択
   - Power BI: 「ホーム」→「データの取得」→データソースを選択
2. **接続情報の入力**:
   - ファイルパス、URL、サーバー名、認証情報など
3. **データのプレビューと選択**:
   - テーブル、シート、クエリなどを選択
   - 「データのロード」または「編集」を選択
4. **Power Query エディタでの編集**:
   - 必要に応じてデータを変換

データソース別の注意点

ファイル接続

- **Excel/CSV**: 列の型が自動推測される（必ず確認が必要）
- **フォルダ**: 同じ構造の複数ファイルを一度に読み込める
- **JSON/XML**: 階層構造を持つため、展開操作が必要なことが多い

データベース接続

- **認証方法**: Windows認証、データベース認証、OAuth認証
- **クエリモード**: インポートかDirectQueryか選択可能（Power BIのみ）

- **ネイティブクエリ**: SQLを直接書ける「詳細オプション」

Web接続

- **動的コンテンツ**: JavaScriptで生成されるコンテンツは取得できないことがある
- **認証**: 一部のWebサイトでは認証が必要
- **Navigation**: ページ内の複数テーブルから選択可能

> 💡 **ベテランの知恵袋**
> データソースを設定する際は、**将来の変更に対して堅牢な設定**を心がけましょう。例えば、ファイルパスを直接ハードコードするのではなく、パラメータを使用すれば、ファイルの場所が変わっても簡単に更新できます。これは特に定期的にデータを更新する場合に役立ちます。

2.5 基本的な変換操作

Power Queryエディタでは、様々なデータ変換操作を行うことができます。ここでは基本的な操作を紹介します。

リボンインターフェース

Power Queryエディタのリボンには、変換操作がカテゴリ別に整理されています：

基本データ変換操作

列の操作

- **列の削除**: 不要な列を選択して右クリック→「削除」または「他の列を削除」
- **列名の変更**: 列ヘッダーをダブルクリック、または右クリック→「名前の変更」
- **列の並べ替え**: 列ヘッダーをドラッグ&ドロップ、または右クリック→「前に移動」/「後ろに移動」
- **データ型の変更**: 列ヘッダーの左側のアイコンをクリック、または「変換」タブ→「データ型」

行の操作

- **フィルター適用**: 列ヘッダーのドロップダウンをクリックしてフィルターオプションを選択
- **上位/下位行の保持**: 「ホーム」タブ→「行の削減」→「上位行の保持」
- **重複の削除**: 「ホーム」タブ→「行の削減」→「重複の削除」
- **エラーのフィルタリング**: 列ヘッダーのフィルター→「エラー」にチェック

値の変換

- **置換**: 「変換」タブ→「値の置換」
- **トリミング**: 「変換」タブ→「トリミング」（前後の空白を削除）
- **大文字/小文字変換**: 「変換」タブ→「大文字に変換」/「小文字に変換」
- **数値の丸め**: 「変換」タブ→「統計」→「四捨五入」など

> ❓ **若手の疑問解決**
> **Q**: 変換操作を間違えた場合、元に戻すにはどうすればよいですか？
> **A**: Power Queryエディタの右側にある「適用した手順」ペインで、間違った手順を右クリックして「削除」または「削除（含む以降のステップ）」を選びます。また、「元に戻す」ボタン（Ctrl+Z）でも最後の操作を取り消せます。

実践的な変換例

日付データの標準化

```
// 様々な形式の日付データを標準形式に変換する例
// 入力データ: "2023/4/1", "2023.4.1", "2023-4-1" など

// 1. 列のデータ型を「テキスト」に変更
// 2. 区切り文字を統一する置換操作
= Table.ReplaceValue(前のステップ, ".", "/", Text.Replace, {"日付"})
= Table.ReplaceValue(前のステップ, "-", "/", Text.Replace, {"日付"})
// 3. 日付型に変換
= Table.TransformColumnTypes(前のステップ, {{"日付", type date}})
```

複数列の結合

```
// 姓と名を結合して氏名列を作成する例
// 入力: "姓"列と"名"列

// 「カスタム列の追加」を使用
= Table.AddColumn(前のステップ, "氏名", each [姓] & " " & [名])
```

変換履歴の理解

Power Queryでの各変換操作は「適用した手順」ペインに記録されます。これはM言語のコードとして保存され、再現可能な変換フローを構成します。

適用した手順

- ソース
- 列型の変更
- フィルター済み行
- 列名の変更
- カスタム列の追加
- 重複の削除

各ステップは：

- 順番に実行される
- 編集可能
- 削除可能
- 無効化可能（右クリック→「無効」）
- 名前変更可能（右クリック→「名前の変更」）

これにより、データ変換プロセスは完全に透明で再現可能になります。

✅ 第2章 チェックリスト

- [ ] Power Queryの基本概念を理解した
- [ ] ExcelとPower BIでのPower Queryへのアクセス方法を把握した
- [ ] さまざまなデータソースへの接続方法を学んだ
- [ ] 基本的なデータ変換操作を実行できる
- [ ] 変換履歴（適用した手順）の仕組みを理解した

第3章: Power Query によるデータ変換の実践

3.1 テーブル操作の基本

テーブル全体を操作する機能は、データ変換の基礎となるものです。ここでは、テーブル全体に対して適用できる主要な操作を学びます。

行の操作

行のフィルタリング

最も基本的な操作の一つは、特定の条件に基づいて行をフィルタリングすることです。

UI操作:

1. フィルタリングしたい列のヘッダーにある下向き矢印をクリック
2. 表示されるメニューからフィルタリングオプションを選択

高度なフィルタリング:

- 「ホーム」タブ→「フィルターの保持」→「詳細フィルター」
- 複数条件の指定が可能（AND/OR条件）

```
// 2020年以降のデータのみをフィルタリングするコード例
= Table.SelectRows(前のステップ, each [年度] >= 2020)
```

> 💡 **ベテランの知恵袋**
> フィルタリングは可能な限り早い段階で適用しましょう。特に大きなデータセットを扱う場合、これによりパフォーマンスが大幅に向上します。不要なデータを最初に除外することで、後続のすべての変換操作が高速化されます。

行の並べ替え

データを特定の列の値に基づいて昇順または降順に並べ替えることができます。

UI操作:

1. 並べ替えたい列のヘッダーにある下向き矢印をクリック
2. 「昇順で並べ替え」または「降順で並べ替え」を選択

複数列による並べ替え:

- 「ホーム」タブ→「並べ替え」→「詳細な並べ替え」
- 複数の列と並べ替え順序を指定可能

```
// 部署で昇順、その後給与で降順に並べ替えるコード例
= Table.Sort(前のステップ, {{"部署", Order.Ascending}, {"給与", Order.Descending}})
```

最初/最後のN行を保持

大きなデータセットの一部のみを扱いたい場合に便利です。

UI操作:

- 「ホーム」タブ→「行の削減」→「上位の行を保持」
- 保持する行数を入力

```
// 上位10行のみを保持するコード例
= Table.FirstN(前のステップ, 10)

// 下位5行のみを保持するコード例
= Table.LastN(前のステップ, 5)
```

重複の処理

同じデータが複数回出現する場合、重複を削除または特定できます。

UI操作:

- 「ホーム」タブ→「行の削減」→「重複の削除」
- 重複を確認する列を選択可能

```
// すべての列を考慮して重複行を削除するコード例
= Table.Distinct(前のステップ)

// 特定の列（顧客ID）のみを考慮して重複を削除するコード例
= Table.Distinct(前のステップ, {"顧客ID"})
```

列の操作

列の選択と削除

データセットから必要な列のみを選択するか、不要な列を削除します。

UI操作:

- 列を選択して右クリック→「削除」
- 複数列を選択（Ctrlキーを押しながら）→右クリック→「削除」
- 保持したい列のみを選択→右クリック→「他の列を削除」

```
// 特定の列のみを選択するコード例
= Table.SelectColumns(前のステップ, {"顧客ID", "氏名", "購入日", "金額"})

// 特定の列を削除するコード例
= Table.RemoveColumns(前のステップ, {"住所", "電話番号", "メモ"})
```

列の並べ替え

列の順序を変更して、より論理的な順序にすることができます。

UI操作:

- 列ヘッダーをドラッグ&ドロップして移動
- 列を選択→右クリック→「前に移動」または「後ろに移動」

```
// 列の順序を指定するコード例
= Table.ReorderColumns(前のステップ, {"顧客ID", "氏名", "購入日", "金額", "商品名"})
```

> ❓ **若手の疑問解決**
> **Q**: 多くの列がある場合、特定のパターンに一致する列だけを選択したり操作したりするにはどうすればよいですか？
> **A**: Power Queryでは、列名のパターンに基づいて列を選択する機能があります。「表示」タブの「列の選択」ダイアログボックスで「列の選択」を選び、フィルタリングオプションを使用できます。また、M言語では `Table.SelectColumns(source, each Text.Contains([Name], "sales"))` のように、列名にパターンマッチングを適用できます。

3.2 列の追加と変更

データ変換では、既存の列を変更したり、新しい列を追加したりすることが多く必要になります。

カスタム列の追加

既存の列のデータを使用して新しい列を作成できます。計算、テキスト操作、条件ロジックなどを適用できます。

UI操作:

1. 「列の追加」タブ→「カスタム列」
2. 数式を入力して新しい列を定義

一般的な使用例:

1. **単純な計算**:

```
// 金額と数量から合計金額を計算する
= [単価] * [数量]
```

2. **テキスト連結**:

```
// 姓と名を連結して完全な名前を作成する
= [姓] & " " & [名]
```

3. **条件付きロジック**:

```
// 点数に基づいて評価を割り当てる
= if [点数] >= 90 then "A" else if [点数] >= 80 then "B" else if [点数] >= 70 then "C"
else "D"
```

4. **日付操作**:

```
// 日付から年を抽出する
= Date.Year([購入日])
```

> 💡 **ベテランの知恵袋**
> カスタム列を作成する際は、まずシンプルなケースで式をテストし、その後徐々に複雑にしていくのがおすすめです。また、大量のデータがある場合、カスタム列の計算は処理時間がかかることがあるので、パフォーマンスに注意しましょう。

インデックス列の追加

行番号を追加することで、データの順序を保持したり、一意の識別子を作成したりできます。

UI操作:

- 「列の追加」タブ→「インデックス列」
- 開始番号と増分値を指定可能

```
// 0から始まるインデックス列を追加するコード例
= Table.AddIndexColumn(前のステップ, "インデックス", 0, 1)
```

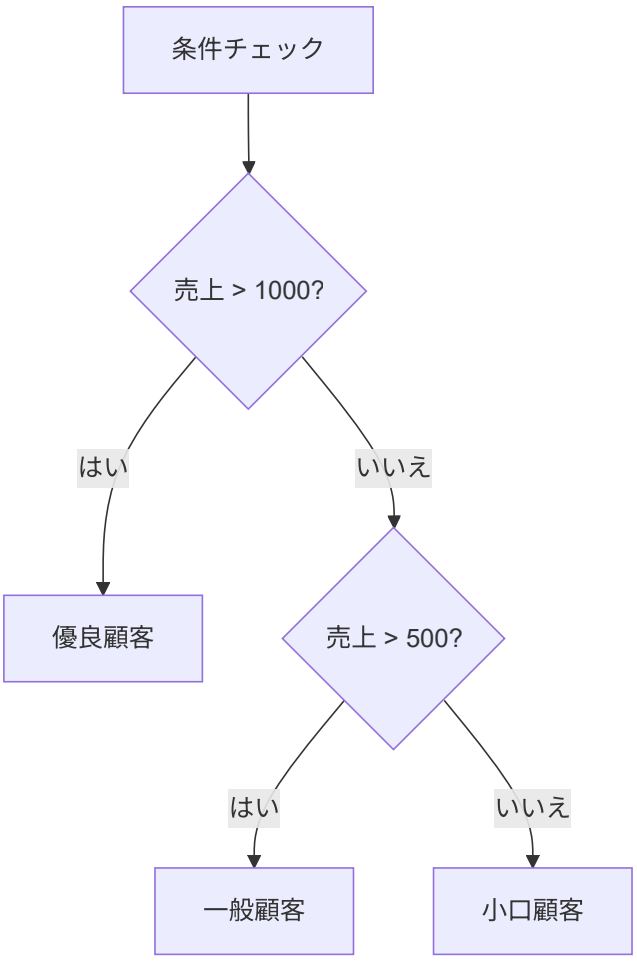
条件付き列

複数の条件に基づいて値を割り当てる新しい列を作成します。

UI操作:

1. 「列の追加」タブ→「条件付き列」
2. 条件と結果を指定

これは特に分類や評価などに役立ちます：



列のデータ型の変更

正確な分析のために、各列に適切なデータ型を設定することが重要です。

UI操作:

- 列ヘッダーの左側のデータ型アイコンをクリック
- 「変換」タブ→「データ型」→適切な型を選択

主要なデータ型:

- テキスト
- 整数/小数点数
- 日付/時刻
- 真偽値（はい/いいえ）
- 通貨

```
// 複数列のデータ型を一度に変更するコード例
= Table.TransformColumnTypes(
    前のステップ,
```

```
    {
        {"売上日", type date},
        {"金額", type number},
        {"顧客ID", type text},
        {"配送済み", type logical}
    }
)
```

> 🚫 **失敗から学ぶ**
> ある会社では、CSVから取り込んだデータの日付列が自動的にテキスト型として認識され、日付順の並べ替えが正しく機能しませんでした。明示的に日付型に変換することで問題が解決しました。データ型の自動検出に頼りすぎず、重要な列は常に明示的にデータ型を設定するのがベストプラクティスです。

列の分割

1つの列を複数の列に分割することで、データをより構造化できます。

UI操作:

1. 分割したい列を選択
2. 「変換」タブ→「列の分割」
3. 区切り文字またはその他の分割オプションを選択

一般的な使用例:

- 「氏名」列を「姓」と「名」に分割
- 「住所」列を「都道府県」「市区町村」などに分割
- 「日時」列を「日付」と「時刻」に分割

```
// コンマで区切られた住所を分割するコード例
= Table.SplitColumn(
    前のステップ,
    "住所",
    Splitter.SplitTextByDelimiter(",", QuoteStyle.Csv),
    {"都道府県", "市区町村", "町名番地"}
)
```

3.3 行のフィルタリングと並べ替え

効率的なデータ分析のためには、必要なデータだけを取り出し、適切に並べ替えることが重要です。

高度なフィルタリング

単純な値フィルターだけでなく、複雑な条件でデータをフィルタリングすることができます。

テキストフィルター

テキスト列に対してさまざまな条件でフィルタリングできます：

UI操作:

- 列ヘッダーのフィルターボタン→「テキストフィルター」
- 「次で始まる」「次を含む」「次で終わる」などのオプション

```
// 東京で始まる住所をフィルタリングするコード例
= Table.SelectRows(前のステップ, each Text.StartsWith([住所], "東京"))

// 「株式会社」を含む会社名をフィルタリングするコード例
= Table.SelectRows(前のステップ, each Text.Contains([会社名], "株式会社"))
```

数値フィルター

数値データに対して範囲や比較演算子を使用してフィルタリングできます：

UI操作:

- 列ヘッダーのフィルターボタン→「数値フィルター」
- 「より大きい」「以下」「間」などのオプション

```
// 1000以上5000以下の金額をフィルタリングするコード例
= Table.SelectRows(前のステップ, each [金額] >= 1000 and [金額] <= 5000)
```

日付フィルター

日付データに特化したフィルターオプションが用意されています：

UI操作:

- 列ヘッダーのフィルターボタン→「日付フィルター」
- 「過去」「今日」「今年」「期間」などのオプション

```
// 今年のデータのみをフィルタリングするコード例
= Table.SelectRows(前のステップ, each Date.Year([日付]) = Date.Year(DateTime.LocalNow()))

// 過去30日間のデータをフィルタリングするコード例
= Table.SelectRows(前のステップ, each [日付] >= Date.AddDays(DateTime.LocalNow(), -30))
```

複合フィルタリング

複数の条件を組み合わせて高度なフィルタリングを行うことができます。

AND条件（すべての条件を満たす）:

```
= Table.SelectRows(
    前のステップ,
    each [部署] = "営業" and [売上] > 1000000 and Date.Year([日付]) = 2023
)
```

OR条件（いずれかの条件を満たす）:

```
= Table.SelectRows(
    前のステップ,
    each [部署] = "営業" or [部署] = "マーケティング"
)
```

複雑な条件の組み合わせ:

```
= Table.SelectRows(
    前のステップ,
    each ([部署] = "営業" or [部署] = "マーケティング") and [売上] > 500000
)
```

> ❓ **若手の疑問解決**
> **Q**: 複数の列に同じフィルター条件を適用したい場合、一つずつ設定する必要がありますか？
> **A**: 残念ながらUIでは各列に個別にフィルターを設定する必要がありますが、M言語を使うと複数列に対して同じパターンのフィルターを一度に適用できます。例えば `Table.SelectRows(前のステップ, each Text.Contains([商品名], "特選") or Text.Contains([説明], "特選"))` のようにできます。

動的フィルタリング（パラメータの活用）

Power Queryのパラメータを使用すると、フィルター条件を動的に変更できます。

パラメータの作成:

1. 「ホーム」タブ→「パラメータの管理」→「新しいパラメータ」
2. 名前、型、現在の値などを設定

パラメータを使用したフィルタリング:

```
= Table.SelectRows(前のステップ, each [部署] = 部署パラメータ)
```

このアプローチは、同じクエリをさまざまな条件で再利用する場合に特に役立ちます。

複数条件による並べ替え

データを複数の列に基づいて並べ替えることができます。

UI操作:

1. 「ホーム」タブ→「並べ替え」→「詳細な並べ替え」
2. 複数の列と並べ替え順序を指定

```
// 部署で昇順に並べ替え、同じ部署内では売上で降順に並べ替えるコード例
= Table.Sort(
    前のステップ,
    {
        {"部署", Order.Ascending},
        {"売上", Order.Descending}
```

```
    }
)
```

3.4 グループ化と集計

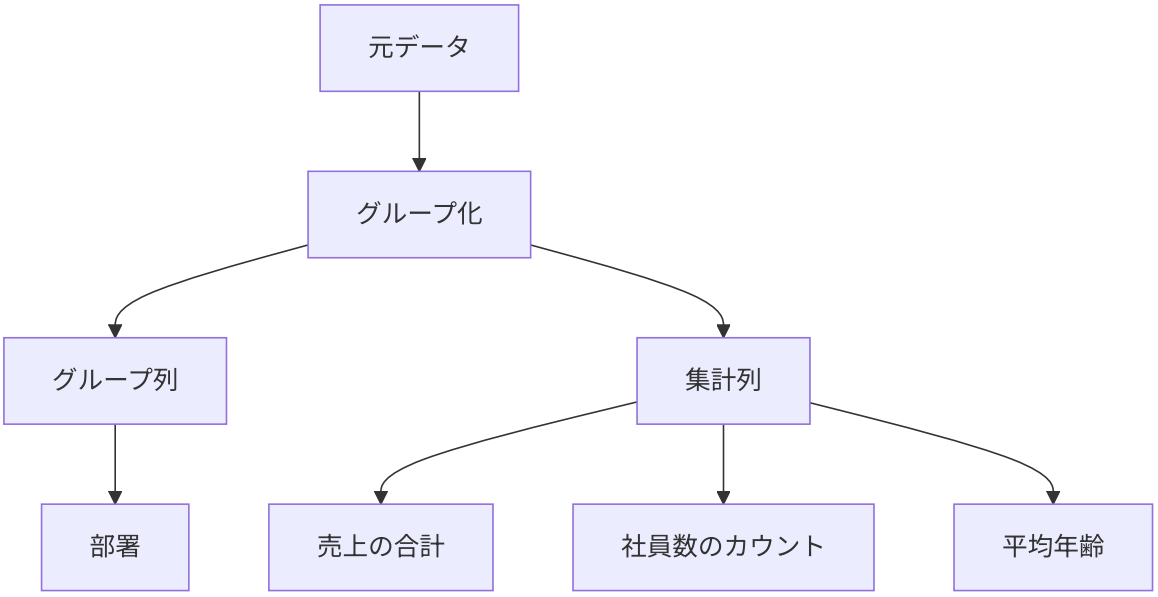
データをグループ化することで、要約情報を抽出し、より高度な分析が可能になります。

基本的なグループ化

グループ化では、一つまたは複数の列の値に基づいてデータをまとめ、各グループに対して集計計算を行います。

UI操作:

1. 「変換」タブ→「グループ化」
2. グループ化する列と集計方法を選択



単一列でのグループ化

最も基本的なグループ化は、1つの列の値でデータをグループ化することです。

```
// 部署ごとに売上合計と社員数を集計するコード例
= Table.Group(
    前のステップ,
    {"部署"},  // グループ化する列
    {
        {"売上合計", each List.Sum([売上]), type number},
        {"社員数", each Table.RowCount(_), type number}
    }
)
```

複数列でのグループ化

より詳細な分析のために、複数の列でグループ化することもできます。

```
// 部署と年度でグループ化し、四半期ごとの売上を集計するコード例
= Table.Group(
    前のステップ,
    {"部署", "年度"},  // グループ化する列
    {
        {"Q1売上", each List.Sum(Table.SelectRows(_, each [四半期] = "Q1")[売上]), type
number},
        {"Q2売上", each List.Sum(Table.SelectRows(_, each [四半期] = "Q2")[売上]), type
number},
        {"Q3売上", each List.Sum(Table.SelectRows(_, each [四半期] = "Q3")[売上]), type
number},
        {"Q4売上", each List.Sum(Table.SelectRows(_, each [四半期] = "Q4")[売上]), type
number},
        {"年間売上", each List.Sum([売上]), type number}
    }
)
```

主な集計関数

グループ化時に使用できる主な集計関数は以下の通りです：

集計関数	説明	例
Sum	合計を計算	`each List.Sum([売上])`
Average	平均を計算	`each List.Average([売上])`
Min	最小値を取得	`each List.Min([売上])`
Max	最大値を取得	`each List.Max([売上])`
Count	行数をカウント	`each Table.RowCount(_)`
CountDistinct	一意の値の数をカウント	`each List.Count(List.Distinct([顧客ID]))`
First	最初の値を取得	`each List.First([取引日])`
Last	最後の値を取得	`each List.Last([取引日])`

> 💡 **ベテランの知恵袋**
> グループ化操作は非常に強力ですが、大量のデータに対して実行すると処理時間がかかることがあります。可能であれば、グループ化の前に必要な列だけを選択し、フィルタリングしておくとパフォーマンスが向上します。

カスタム集計の作成

標準の集計関数以外にも、独自のロジックで集計値を計算できます。

例: 売上上位3件の平均を計算

```
// 部署ごとに売上上位3件の平均を計算するコード例
= Table.Group(
    前のステップ,
    {"部署"},
    {
        {"トップ3平均売上", each
            List.Average(
                List.FirstN(
                    List.Sort([売上], Order.Descending),
                    3
                )
            ),
            type number
        }
    }
)
```

例: 中央値の計算

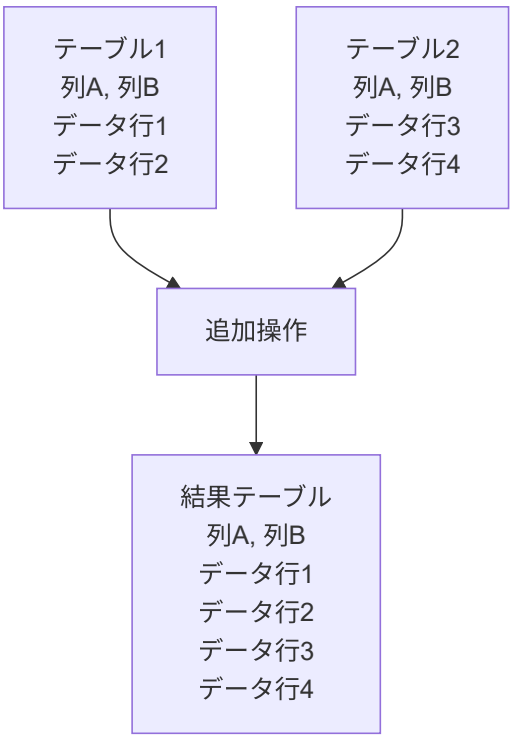
```
// 給与の中央値を計算するコード例
= Table.Group(
    前のステップ,
    {"部署"},
    {
        {"給与中央値", each
            let
                SortedList = List.Sort([給与]),
                Count = List.Count(SortedList),
                Median = if Number.IsOdd(Count) then
                            List.Nth(SortedList, Number.IntegerDivide(Count, 2))
                         else
                            (List.Nth(SortedList, Number.IntegerDivide(Count, 2) - 1) +
List.Nth(SortedList, Number.IntegerDivide(Count, 2))) / 2
            in
                Median,
            type number
        }
    }
)
```

3.5 テーブルの結合とマージ

複数のデータソースからのデータを組み合わせることは、データ分析における重要なタスクです。Power Queryでは、テーブルを結合する方法として主に「追加」と「マージ」の2つの操作があります。

テーブルの追加（行の結合）

「追加」操作は、2つ以上のテーブルを縦方向に結合します。これは、同じ構造（同じ列）を持つテーブルを単一のテーブルにまとめる場合に使用します。



UI操作:

1. 「ホーム」タブ→「テーブルの追加」
2. 追加するテーブルを選択し、列の対応を確認

```
// 2つのテーブルを追加するコード例
= Table.Combine({テーブル1, テーブル2})
```

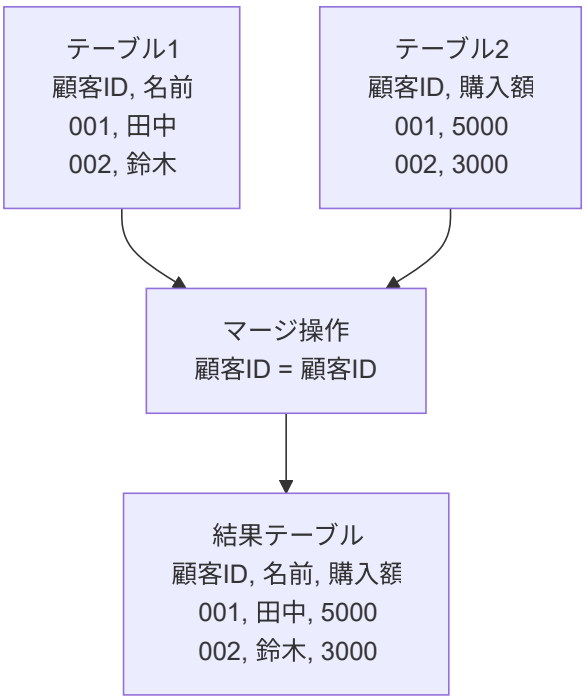
異なる構造のテーブルを追加:
列名や数が異なるテーブルを追加する場合、列の対応付けを行う必要があります。

```
// 列構造が異なるテーブルを追加するコード例
= Table.Combine({
    テーブル1,
    Table.RenameColumns(テーブル2, {{"古い列名", "新しい列名"}})
})
```

> ❓ **若手の疑問解決**
> **Q**: テーブル追加で列が足りない場合、どうなりますか？
> **A**: 一方のテーブルにしか存在しない列がある場合、その列は存在しないテーブルの行に対してnull値が入ります。これにより、結合後のテーブルは常にすべての列を持ちます。

テーブルのマージ（列の結合）

「マージ」操作は、2つのテーブルを特定の列の値に基づいて水平方向に結合します。これはSQLのJOIN操作に相当し、異なるテーブルからの関連情報を組み合わせる場合に使用します。



UI操作:

1. 「ホーム」タブ→「テーブルのマージ」
2. 結合するテーブルと結合キー（列）を選択
3. 結合の種類を選択

結合の種類

Power Queryでは、以下の結合タイプが利用可能です：

1. **内部結合（Inner Join）**: 両方のテーブルで一致する行のみを返す
2. **左外部結合（Left Outer Join）**: 左テーブルのすべての行と、一致する右テーブルの行を返す
3. **右外部結合（Right Outer Join）**: 右テーブルのすべての行と、一致する左テーブルの行を返す
4. **完全外部結合（Full Outer Join）**: 両方のテーブルのすべての行を返す
5. **左反結合（Left Anti Join）**: 左テーブルの行のうち、右テーブルに一致しない行のみを返す
6. **右反結合（Right Anti Join）**: 右テーブルの行のうち、左テーブルに一致しない行のみを返す

```
// 顧客テーブルと注文テーブルを顧客IDで内部結合するコード例
= Table.NestedJoin(
    顧客テーブル,        // 左テーブル
    {"顧客ID"},          // 左結合キー
    注文テーブル,        // 右テーブル
    {"顧客ID"},          // 右結合キー
    "結合結果",          // 結果列名
    JoinKind.Inner       // 結合の種類
)
```

結合後のデータの展開:
マージ操作後、結果は新しい列にテーブル形式で格納されます。これを展開して実際の列にアクセスする必要があります。

UI操作:

1. 結合結果列の展開ボタン（矢印アイコン）をクリック
2. 展開する列を選択

```
// 結合結果を展開するコード例
= Table.ExpandTableColumn(
    前のステップ,
    "結合結果",           // 展開する列名
    {"購入日", "金額"},  // 展開する列
    {"購入日", "金額"}   // 新しい列名
)
```

> 🚫 **失敗から学ぶ**
> マージ操作でよくある問題は、結合キーのデータ型が異なることです。例えば、一方のテーブルでは顧客IDが数値型、もう一方では文字列型という場合、結合が正しく機能しません。マージ前に、結合キーの列のデータ型が一致していることを確認しましょう。

複数条件でのマージ

複数の列を結合キーとして使用することもできます。

```
// 顧客IDと日付の両方で注文テーブルと在庫テーブルを結合するコード例
= Table.NestedJoin(
    注文テーブル,
    {"顧客ID", "注文日"},
    在庫テーブル,
    {"顧客ID", "日付"},
    "結合結果",
    JoinKind.Inner
)
```

高度なマージテクニック

自己結合

同じテーブルを自分自身とマージすることで、階層関係などを分析できます。

```
// 社員テーブルを自己結合して上司の情報を取得するコード例
= Table.NestedJoin(
    社員テーブル,
    {"上司ID"},          // 社員の上司ID
    社員テーブル,        // 同じテーブル
    {"社員ID"},          // 社員ID
    "上司情報",
```

```
    JoinKind.LeftOuter
)
```

ファジー結合

Power BIのPower Queryでは、完全一致ではなく類似したテキスト値に基づく「ファジー結合」も可能です。

UI操作:

1. 「ホーム」タブ→「テーブルのマージ」
2. 「結合の種類」で「ファジー」を選択
3. 類似度などのオプションを設定

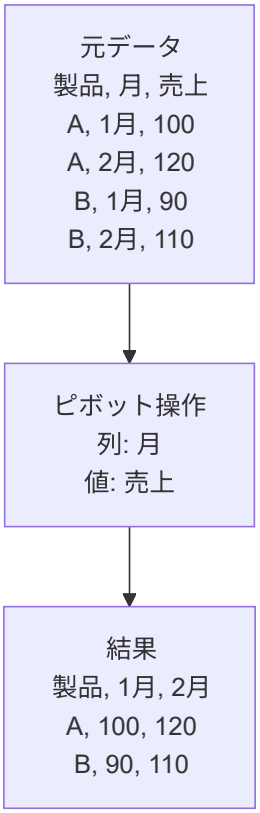
これは、表記揺れのある顧客名や製品名でテーブルを結合する場合に非常に役立ちます。

3.6 ピボットとアンピボット

データの形状変換は、分析に適した形式にデータを整理するために重要な操作です。Power Queryでは、ピボットとアンピボットという操作でデータの向きを変換できます。

ピボット操作

ピボット操作は、行として格納されている値を列に変換します。これはクロス集計表を作成する場合に役立ちます。



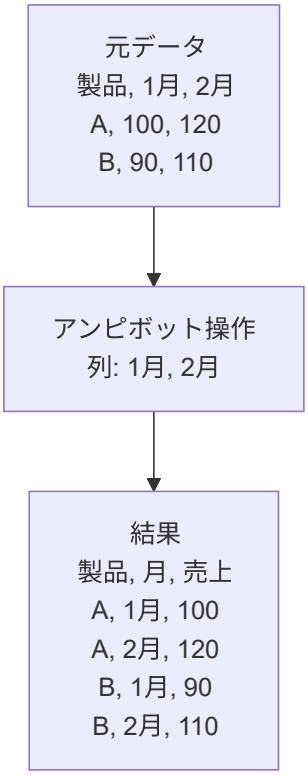
UI操作:

1. ピボット対象の列を選択（例：「月」列）
2. 右クリック→「ピボット列」
3. 値列を選択（例：「売上」列）
4. 必要に応じて集計方法を選択

```
// 月ごとの売上をピボットするコード例
= Table.Pivot(
    前のステップ,
    List.Distinct([月]),  // 列ヘッダーになる値
    "月",                 // ピボットする列
    "売上",               // 値列
    List.Sum              // 集計関数（オプション）
)
```

アンピボット操作

アンピボット操作は、ピボットの逆で、列として格納されている値を行に変換します。これは「整然データ（Tidy Data）」形式に変換する場合に便利です。



UI操作:

1. アンピボットしたい列を選択（通常、複数列を選択）
2. 右クリック→「列のアンピボット」

3. 必要に応じて属性列名と値列名を設定

```
// 月別売上列をアンピボットするコード例
= Table.UnpivotOtherColumns(
    前のステップ,
    {"製品"},          // そのまま保持する列
    "月",              // 属性列名（元の列名が格納される）
    "売上"             // 値列名（元の列の値が格納される）
)
```

アンピボットの応用: 特定の列のみをアンピボット

すべての列ではなく、特定のパターンに一致する列だけをアンピボットしたい場合は、以下のアプローチが使用できます。

UI操作:

1. アンピボットしたい列のみを選択
2. 右クリック→「選択した列のアンピボット」

```
// Q1からQ4までの四半期列のみをアンピボットするコード例
= Table.UnpivotOtherColumns(
    前のステップ,
    {"年度", "部署", "担当者"},  // そのまま保持する列
    "四半期",                    // 属性列名
    "売上"                       // 値列名
)
```

> 💡 **ベテランの知恵袋**
> データ分析では、ピボットテーブルやビジュアルで使用する前に、まずデータを「整然データ」形式（各観測値が行、各変数が列、各種類の観測単位が表）に整理することが推奨されます。Power Queryのアンピボットはこのような変換に最適です。

ピボット/アンピボットの実践例

例1: 月次売上データの四半期集計

月ごとの売上データを四半期ごとにピボットして集計する例：

```
// まず月を四半期に変換
= Table.AddColumn(前のステップ, "四半期", each
    if [月] <= 3 then "Q1" else if [月] <= 6 then "Q2" else if [月] <= 9 then "Q3" else
"Q4",
    type text)

// 四半期でピボット
= Table.Pivot(
    前のステップ,
    List.Distinct([四半期]),
```

```
    "四半期",
    "売上",
    List.Sum
)
```

例2: 調査データの整形

アンケート結果の各質問（Q1,Q2,Q3...）が列になっているデータを、分析しやすいように変換する例：

```
// 質問列をアンピボット
= Table.UnpivotOtherColumns(
    前のステップ,
    {"回答者ID", "年齢", "性別"},  // 基本情報は保持
    "質問",                        // 質問ID
    "回答"                         // 回答内容
)

// 質問IDと質問内容のマッピングテーブルとマージ
= Table.NestedJoin(
    前のステップ,
    {"質問"},
    質問マスタ,
    {"質問ID"},
    "質問詳細",
    JoinKind.LeftOuter
)
```

✅ 第3章 チェックリスト

- [ ] テーブル操作の基本（行操作、列操作）を理解した
- [ ] カスタム列の追加と列の変更方法を学んだ
- [ ] 複合条件によるフィルタリングと並べ替えができる
- [ ] グループ化と集計操作を活用できる
- [ ] テーブルの結合（追加とマージ）の違いを理解した
- [ ] ピボットとアンピボット操作を使いこなせる

第4章: M言語によるPower Query の拡張

4.1 M言語の基本

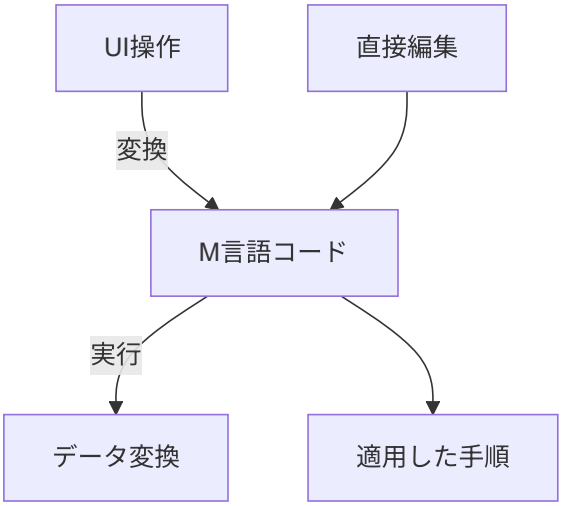
これまで、Power Queryの操作はほとんどユーザーインターフェース（UI）を通じて行ってきました。しかし、Power Queryの真の力を引き出すには、背後で動作するM言語（Power Query式言語）を理解することが重要です。

M言語とは

M言語は、Power Queryのために特別に設計された関数型プログラミング言語です。Power Queryエディタで行う操作は、バックグラウンドでM言語のコードに変換されます。

特徴：

- 関数型プログラミング言語
- 強い型付け
- 遅延評価（必要になるまで実行されない）
- ステップバイステップの変換操作を記述



M言語の基本構文

M言語の基本的な構文要素を理解しましょう。

値と型

M言語では、さまざまなデータ型を扱うことができます：

```
// 数値
let
    整数 = 42,
    小数 = 3.14
in
    小数
```

```
// テキスト
let
    メッセージ = "こんにちは、M言語"
in
    メッセージ

// 論理値
let
    真 = true,
    偽 = false
in
    真

// リスト（配列）
let
    数字リスト = {1, 2, 3, 4, 5},
    文字リスト = {"a", "b", "c"}
in
    数字リスト

// レコード（オブジェクト）
let
    人物 = [
        名前 = "田中太郎",
        年齢 = 30,
        部署 = "営業"
    ]
in
    人物

// テーブル
let
    社員テーブル = #table(
        {"社員ID", "名前", "部署"},
        {
            {1, "山田一郎", "営業"},
            {2, "鈴木二郎", "開発"},
            {3, "佐藤三郎", "管理"}
        }
    )
in
    社員テーブル
```

let式

M言語では、`let...in`構文が基本的なコードブロックとなります：

```
let
    // 変数定義
    名前1 = 値1,
    名前2 = 値2,
    // 最後の変数
    名前n = 値n
in
    // 結果として返す値
    最終結果
```

各ステップで定義した値は、それ以降のステップで参照できます。

関数

M言語では関数は第一級オブジェクトで、変数に代入したり、他の関数に渡したりできます：

```
// 簡単な関数定義
let
    二乗 = (x) => x * x,
    結果 = 二乗(5)
in
    結果  // 結果は25

// 複数パラメータを持つ関数
let
    加算 = (a, b) => a + b,
    結果 = 加算(10, 20)
in
    結果  // 結果は30
```

if式による条件分岐

条件によって異なる値を返す場合は、if式を使用します：

```
let
    点数 = 85,
    結果 = if 点数 >= 90 then "A"
           else if 点数 >= 80 then "B"
           else if 点数 >= 70 then "C"
           else "D"
in
    結果  // 結果は"B"
```

リストとリスト関数

リスト（配列）は、M言語でよく使われるデータ構造です。List名前空間の関数を使用してリストを操作できます：

```
let
    数字 = {1, 2, 3, 4, 5},

    // リストの合計
    合計 = List.Sum(数字),  // 15

    // リストの平均
    平均 = List.Average(数字),  // 3

    // リストのフィルタリング（偶数のみ）
    偶数 = List.Select(数字, each _ mod 2 = 0),  // {2, 4}

    // リストの変換（各要素を2倍）
    倍数 = List.Transform(数字, each _ * 2)  // {2, 4, 6, 8, 10}
in
    倍数
```

> 💡 **ベテランの知恵袋**
> M言語での `each` キーワードは、リスト処理でよく使用される省略記法です。`each x` は `(x) => x` の省略形です。リスト内の各要素に対する操作を簡潔に書けるため、List関数では頻繁に使用されます。

レコードとフィールドアクセス

レコード（オブジェクト）のフィールドには、ドット記法または括弧記法でアクセスできます：

```
let
    人物 = [
        名前 = "田中太郎",
        年齢 = 30,
        部署 = "営業"
    ],

    // ドット記法でアクセス
    名前 = 人物[名前],  // "田中太郎"

    // 括弧記法でアクセス（動的フィールド名に便利）
    フィールド名 = "部署",
    部署情報 = 人物[フィールド名]  // "営業"
in
    部署情報
```

4.2 UIで生成されたコードの理解

Power Queryのユーザーインターフェースで行った操作は、すべてM言語のコードとして記録されます。この生成コードを理解することで、より高度なカスタマイズが可能になります。

高度なエディタの開き方

Power Queryエディタでは、各ステップのM言語コードを確認できます：

1. 「適用した手順」ペインでステップを選択
2. 数式バーにそのステップのM言語コードが表示される
3. より広い表示で編集したい場合は、数式バー右側の下向き矢印をクリック

または、「表示」タブ→「高度なエディター」で、クエリ全体のM言語コードを確認できます。

一般的なUIステップとM言語コードの対応

Power QueryのUI操作がどのようなM言語コードに変換されるかを理解しましょう。

データソースへの接続

```
// Excelファイルへの接続
Source = Excel.Workbook(File.Contents("C:\Data\Sales.xlsx"), null, true),
Data = Source{[Item="売上データ",Kind="Table"]}[Data]
```

列の型変換

```
// 日付と数値の型変換
型変換済み = Table.TransformColumnTypes(
        前のステップ,
        {
            {"売上日", type date},
            {"売上金額", Currency.Type},
            {"数量", Int64.Type}
        }
    )
```

列のフィルタリング

```
// 東京支店のデータのみを抽出
フィルタ済み = Table.SelectRows(前のステップ, each [支店] = "東京")

// 売上金額が10万円以上のデータを抽出
フィルタ済み = Table.SelectRows(前のステップ, each [売上金額] >= 100000)
```

列の追加

```
// カスタム列の追加（単価を計算）
単価追加 = Table.AddColumn(前のステップ, "単価", each [売上金額] / [数量], type number)

// 条件付き列の追加
ランク追加 = Table.AddColumn(前のステップ, "ランク", each
    if [売上金額] >= 500000 then "A"
    else if [売上金額] >= 300000 then "B"
    else if [売上金額] >= 100000 then "C"
```

```
    else "D",
    type text)
```

グループ化と集計

```
// 支店と商品カテゴリでグループ化し、売上合計を計算
グループ化 = Table.Group(
    前のステップ,
    {"支店", "商品カテゴリ"},
    {
        {"売上合計", each List.Sum([売上金額]), type number},
        {"平均単価", each List.Average([単価]), type number},
        {"取引回数", each Table.RowCount(_), Int64.Type}
    }
)
```

テーブルの結合

```
// テーブルの追加（縦方向の結合）
追加結果 = Table.Combine({テーブル1, テーブル2})

// テーブルのマージ（横方向の結合）
マージ結果 = Table.NestedJoin(
    顧客テーブル,
    {"顧客ID"},
    注文テーブル,
    {"顧客コード"},
    "注文詳細",
    JoinKind.LeftOuter
)
```

> ❓ **若手の疑問解決**
> **Q**: 自動生成されたM言語コードを修正しても安全ですか？
> **A**: はい、安全です。ただし、コードを変更すると、その後のUIでの操作が制限される場合があります。また、変更後のコードにエラーがないかを確認することが重要です。基本的な変更から始めて、徐々に複雑な変更に挑戦するのがおすすめです。

コードの読み方と修正方法

生成されたM言語コードを読み、修正するためのポイントを紹介します。

コードの基本構造

Power Queryのすべてのクエリは、以下の基本構造を持っています：

```
let
    Source = ..., // 最初のステップ（データソース）
    ステップ1 = ..., // 1つ目の変換
    ステップ2 = ..., // 2つ目の変換
```

```
    ...
    最終ステップ = ... // 最後の変換
in
    最終ステップ // 結果として返されるステップ
```

各ステップは前のステップの結果を受け取り、変換を適用して次のステップに渡します。

変数名の理解

生成されるコードでは、各ステップに名前が付けられます：

- 「Source」: 通常、最初のデータソース接続ステップ
- 日本語の変数名: ステップの名前変更を反映（例：「列型の変更」「フィルタ済み」）
- 自動生成名: ステップ名を変更していない場合（例：「#"Changed Type"」「Filtered Rows」）

特殊文字を含む変数名

M言語では、特殊文字やスペースを含む変数名をハッシュ記号とダブルクォートで囲みます：

```
#"列型の変更" = Table.TransformColumnTypes(...)
```

この構文は自動生成されるコードでよく見られます。ステップ名を単純な英数字にすると、この特殊な記法は不要になります。

コード修正の一般的なパターン

1. **値の変更**: 固定値（ファイルパス、フィルタ条件など）を修正

```
// 修正前
フィルタ済み = Table.SelectRows(前のステップ, each [支店] = "東京")

// 修正後（複数の支店を対象に）
フィルタ済み = Table.SelectRows(前のステップ, each [支店] = "東京" or [支店] = "大阪")
```

2. **関数の変更**: 使用する関数や引数を修正

```
// 修正前（四捨五入）
丸め処理 = Table.TransformColumns(前のステップ, {{"金額", each Number.Round(_, 0), type
number}})

// 修正後（切り捨て）
丸め処理 = Table.TransformColumns(前のステップ, {{"金額", each Number.RoundDown(_, 0),
type number}})
```

3. **ステップの追加**: 既存のステップ間に新しいステップを挿入

```
// 既存のコード
型変換 = Table.TransformColumnTypes(...),
フィルタ済み = Table.SelectRows(型変換, ...),
```

```
// 新しいステップを挿入
型変換 = Table.TransformColumnTypes(...),
空白削除 = Table.TransformColumns(型変換, {{"顧客名", Text.Trim, type text}}),
フィルタ済み = Table.SelectRows(空白削除, ...),
```

4.3 関数の作成と利用

M言語では関数を作成することで、繰り返し使用する処理をカプセル化し、再利用できます。これにより、コードの可読性向上と保守性の向上が図れます。

基本的な関数の作成

M言語での関数定義の基本構文は次のとおりです：

```
(param1, param2, ...) => 計算式またはlet式
```

シンプルな関数の例：

```
// 消費税を計算する関数
(価格) => 価格 * 1.10

// 複数のパラメータを持つ関数
(価格, 税率) => 価格 * (1 + 税率)
```

より複雑な処理を行う関数は、let式を使用できます：

```
// 商品の割引後価格を計算する関数
(価格, 数量) =>
    let
        // 数量に応じた割引率を決定
        割引率 = if 数量 >= 100 then 0.20
                 else if 数量 >= 50 then 0.15
                 else if 数量 >= 10 then 0.10
                 else 0,

        // 割引後の単価
        割引後単価 = 価格 * (1 - 割引率),

        // 合計金額
        合計 = 割引後単価 * 数量
    in
        合計
```

Power Queryでの関数の作成方法

Power Queryでは、以下の方法で関数を作成できます：

1. パラメータからの関数化

既存のクエリをパラメータ化して関数に変換する方法です：

1. 通常のクエリを作成し、動作を確認
2. 可変要素（ファイル名、フィルタ条件など）をパラメータに置き換え
3. 「ホーム」タブ→「クエリの管理」→クエリを選択→右クリック→「関数に変換」

2. 空の関数から作成

新しい関数を最初から作成する方法です：

1. 「ホーム」タブ→「新しいソース」→「空のクエリ」
2. 「表示」タブ→「高度なエディター」
3. 関数コードを直接入力

```
// ファイル名をパラメータとして受け取り、そのファイルから特定のシートを読み込む関数
(ファイル名 as text) =>
let
    ソース = Excel.Workbook(File.Contents(ファイル名), null, true),
    売上データ = ソース{[Item="売上", Kind="Sheet"]}[Data],
    ヘッダー昇格 = Table.PromoteHeaders(売上データ, [PromoteAllScalars=true]),
    型変換 = Table.TransformColumnTypes(
        ヘッダー昇格,
        {
            {"日付", type date},
            {"売上", type number},
            {"数量", Int64.Type}
        }
    )
in
    型変換
```

関数のテストと使用

関数を作成したら、テストして使用できます：

1. 関数クエリが選択された状態で、パラメータ入力フィールドが表示される
2. パラメータ値を入力し、「呼び出し」ボタンをクリック
3. 関数の出力結果が表示される

作成した関数は、他のクエリから呼び出すことができます：

```
// 月次売上ファイルを読み込む関数を使用
一月データ = 売上ファイル読込("C:\Data\Sales_2023_01.xlsx")
二月データ = 売上ファイル読込("C:\Data\Sales_2023_02.xlsx")
三月データ = 売上ファイル読込("C:\Data\Sales_2023_03.xlsx")
```

```
// 3ヶ月分のデータを結合
第一四半期 = Table.Combine({一月データ, 二月データ, 三月データ})
```

> 💡 **ベテランの知恵袋**
> 関数を作成する際は、エラー処理を考慮することが重要です。例えば、ファイルが存在しない場合や、期待する形式でない場合のエラーハンドリングを含めると、より堅牢な関数になります。try/otherwise構文を使用したエラーハンドリングを検討してください。

関数ライブラリの構築

よく使う関数をライブラリとして構築すると、複数のプロジェクトで再利用できます：

1. 共通関数を含むPower Queryファイル（.pqfl）を作成
2. 他のファイルからクエリのインポートで利用

```
// 郵便番号から住所を検索する関数
(郵便番号 as text) =>
let
    // 郵便番号のフォーマットをチェック（3桁-4桁形式）
    有効な形式 = Text.Length(Text.Remove(郵便番号, "-")) = 7,

    // 郵便番号マスタテーブルへの接続（例としての実装）
    郵便番号DB = Excel.Workbook(File.Contents("C:\Data\PostalCode.xlsx"), null, true),
    郵便番号テーブル = 郵便番号DB{[Item="郵便番号", Kind="Table"]}[Data],

    // 郵便番号で検索
    検索結果 = Table.SelectRows(郵便番号テーブル, each [郵便番号] = 郵便番号),

    // 結果の処理
    住所 = if Table.IsEmpty(検索結果) then
              "該当する住所がありません"
           else
              検索結果{0}[都道府県] & 検索結果{0}[市区町村] & 検索結果{0}[町名]
in
    住所
```

4.4 条件分岐とループ

M言語ではデータ変換のための条件分岐とループ処理が可能です。ただし、通常のプログラミング言語とは少し異なる方法で実装します。

条件分岐

if 式による条件分岐

M言語の基本的な条件分岐は if 式を使用します：

```
if 条件 then 真の場合の値 else 偽の場合の値
```

複数条件の例：

```
// 成績評価の例
評価 = if 点数 >= 90 then "A"
       else if 点数 >= 80 then "B"
       else if 点数 >= 70 then "C"
       else if 点数 >= 60 then "D"
       else "F"
```

テーブル内での条件分岐：

```
// 条件に基づいて値を設定するカスタム列
Table.AddColumn(前のステップ, "区分", each
    if [売上] >= 1000000 then "大口"
    else if [売上] >= 500000 then "中口"
    else "小口",
    type text
)
```

条件分岐関数

より複雑な条件分岐には、M言語の以下の関数が役立ちます：

- **if then else**: 基本的な条件分岐

  ```
  if 条件 then 値1 else 値2
  ```

- **logical 演算子**: 複数条件の結合

  ```
  // AND条件
  if 条件1 and 条件2 then 値1 else 値2

  // OR条件
  if 条件1 or 条件2 then 値1 else 値2

  // NOT条件
  if not 条件 then 値1 else 値2
  ```

- **検証関数**: 値の種類やパターンをチェック

  ```
  // 数値かどうか
  if Value.Is(値, type number) then "数値です" else "数値ではありません"

  // nullかどうか
  if 値 = null then "値がありません" else "値があります"
  ```

> 🚫 **失敗から学ぶ**
> M言語での条件分岐で注意すべき点は、すべての条件分岐パスで返される値の型が一致している必要

> があることです。例えば、ある条件では数値、別の条件ではテキストを返すようなコードはエラーになります。このようなケースでは、明示的な型変換が必要になります。

ループ処理

M言語は関数型言語であるため、伝統的な命令型のループ（for, while）は存在しません。代わりに、以下の方法でループに相当する処理を実装します：

1. List関数によるイテレーション

リスト（配列）の各要素に対する操作は、List名前空間の関数を使用します：

```
// リストの各要素に関数を適用
List.Transform({1, 2, 3, 4, 5}, each _ * 2)  // 結果: {2, 4, 6, 8, 10}

// 条件に一致する要素のみをフィルタリング
List.Select({1, 2, 3, 4, 5}, each _ mod 2 = 0)  // 結果: {2, 4}

// 累積値の計算
List.Accumulate({1, 2, 3, 4, 5}, 0, (state, current) => state + current)  // 結果: 15
```

2. Table関数によるイテレーション

テーブルの各行に対する操作は、Table名前空間の関数を使用します：

```
// 各行に対して変換を適用
Table.TransformRows(
    テーブル,
    each [
        顧客ID = [顧客ID],
        氏名 = [姓] & " " & [名],
        年齢 = [年齢],
        区分 = if [年齢] >= 60 then "シニア" else "一般"
    ]
)

// 列の値を変換
Table.TransformColumns(
    テーブル,
    {
        {"金額", each _ * 1.1, type number},  // 10%増額
        {"商品名", Text.Proper, type text}    // 先頭文字を大文字に
    }
)
```

3. 再帰関数による反復処理

より複雑なループには、再帰関数を使用できます：

```
// 1からnまでの合計を計算する再帰関数
(n as number) =>
let
    // 再帰のヘルパー関数
    合計計算 = (現在値, 累積) =>
        if 現在値 > n then
            累積
        else
            @合計計算(現在値 + 1, 累積 + 現在値)
in
    合計計算(1, 0)
```

再帰関数を使った例：指定回数データを処理する関数

```
// 指定回数、指定関数を繰り返し適用する関数
(値, 関数, 回数) =>
let
    // 再帰ヘルパー
    反復適用 = (現在値, 残り回数) =>
        if 残り回数 <= 0 then
            現在値
        else
            @反復適用(関数(現在値), 残り回数 - 1)
in
    反復適用(値, 回数)

// 使用例: 2を3回2倍する（2→4→8→16）
反復(2, each _ * 2, 3)  // 結果: 16
```

条件付きテーブル操作

条件に基づいて異なるテーブル操作を適用する例を見てみましょう：

```
// データの性質に応じて異なる変換を適用
let
    データ = テーブル,

    // データの性質を確認
    列数 = Table.ColumnCount(データ),
    行数 = Table.RowCount(データ),

    // 条件に応じた処理
    結果 = if 列数 > 10 and 行数 > 1000 then
               // 大きなデータセットの場合、一部の列のみを選択
               Table.SelectColumns(データ, {"ID", "名前", "売上", "日付"})
           else if 行数 = 0 then
               // 空のテーブルの場合、スキーマだけを保持
               データ
```

```
            else
                // 標準的なケース、通常の変換を適用
                データ |> Table.TransformColumnTypes(...) |> Table.SelectRows(...)
in
    結果
```

4.5 UIでは難しい処理のカスタマイズ

Power QueryのUIは多くの一般的なデータ変換操作をカバーしていますが、より高度なニーズに対応するためには、M言語を直接編集する必要があるケースがあります。

UIで対応が難しい処理

1. 動的な列名の処理

列名が動的に変化するテーブルを処理する場合、M言語を使うと柔軟に対応できます：

```
// テーブル内のすべての数値列の合計行を追加
let
    データ = テーブル,

    // 数値列を識別
    列名リスト = Table.ColumnNames(データ),
    数値列リスト = List.Select(
        列名リスト,
        each Type.Is(Table.Column(データ, _), type number)
    ),

    // 各数値列の合計を計算
    合計行 = Record.FromList(
        List.Transform(
            数値列リスト,
            each List.Sum(Table.Column(データ, _))
        ),
        数値列リスト
    ),

    // 数値以外の列にはnullを設定
    完全合計行 = Record.Combine({
        Record.FromList(
            List.Transform(
                List.Difference(列名リスト, 数値列リスト),
                each null
            ),
            List.Difference(列名リスト, 数値列リスト)
        ),
        合計行
    }),

    // 合計行を含む新しいテーブル
```

```
    結果 = Table.InsertRows(データ, Table.RowCount(データ), {完全合計行})
in
    結果
```

2. 複雑なフォルダー処理

複数のフォルダやサブフォルダ内のファイルを再帰的に処理する場合：

```
// フォルダとサブフォルダ内のすべてのCSVファイルを再帰的に読み込む関数
(フォルダパス as text) as table =>
let
    // フォルダ内のアイテムを取得
    フォルダ内容 = Folder.Contents(フォルダパス),

    // フォルダとファイルを分離
    フォルダ = Table.SelectRows(フォルダ内容, each [Attributes][Folder] = true),
    ファイル = Table.SelectRows(フォルダ内容, each [Attributes][Folder] = false),

    // CSVファイルのみをフィルタリング
    CSVファイル = Table.SelectRows(ファイル, each Text.EndsWith(Text.Lower([Extension]),
".csv")),

    // 各CSVファイルを読み込み
    CSVデータ = Table.AddColumn(CSVファイル, "データ", each
Csv.Document(File.Contents([Folder Path] & [Name]))),

    // サブフォルダの処理（再帰呼び出し）
    サブフォルダデータ = List.Combine(
        List.Transform(
            Table.ToRecords(フォルダ),
            each Table.ToList(Table.SelectColumns(
                @フォルダ処理([Folder Path]),
                {"データ"}
            ))
        )
    ),

    // すべてのデータを結合
    全データ = List.Combine({
        Table.ToList(Table.SelectColumns(CSVデータ, {"データ"})),
        サブフォルダデータ
    })
in
    #table(
        {"データ"},
        List.Transform(全データ, each {_})
    )
```

3. APIとの連携

REST APIからデータを取得し、複雑なJSON応答を処理する例：

```
// JSONデータを含むWeb APIからデータを取得する関数
(APIエンドポイント as text, パラメータ as record) =>
let
    // HTTPリクエストを構築
    クエリパラメータ = Uri.BuildQueryString(パラメータ),
    URL = APIエンドポイント & "?" & クエリパラメータ,

    // APIを呼び出し
    応答 = Web.Contents(URL),

    // JSONを解析
    JSONデータ = Json.Document(応答),

    // 例：JSONから特定のデータ構造を抽出
    アイテム = JSONデータ[items],

    // レコードリストをテーブルに変換
    テーブル = Table.FromRecords(アイテム)
in
    テーブル
```

> 💼 **プロジェクト事例**
> ある企業では、数百の部署別レポートをExcelで作成していました。各部署のデータはSharePointに保存され、毎週手動で更新されていました。Power Queryを使って、部署コードをパラメータとする関数を作成し、M言語で動的にフォルダパスを構築することで、すべてのレポートを自動生成できるようになりました。これにより、週に2日かかっていた作業が数分に短縮されました。

カスタムM言語関数の例

1. 日本の祝日を判定する関数

```
// 日付が日本の祝日かどうかを判定する関数
(日付 as date) as logical =>
let
    // 祝日マスタ（簡易版）
    祝日リスト = #table(
        {"日付", "名称"},
        {
            {#date(2023, 1, 1), "元日"},
            {#date(2023, 1, 2), "振替休日"},
            {#date(2023, 1, 9), "成人の日"},
            {#date(2023, 2, 11), "建国記念の日"},
            {#date(2023, 2, 23), "天皇誕生日"},
            {#date(2023, 3, 21), "春分の日"}
            // 他の祝日も同様に追加
        }
    ),
```

```
    // 日付で検索
    結果 = Table.SelectRows(祝日リスト, each [日付] = 日付),

    // 祝日かどうか
    祝日か = not Table.IsEmpty(結果)
in
    祝日か
```

2. テーブルの差分を抽出する関数

```
// 2つのテーブルの差分を抽出する関数
(古いテーブル as table, 新しいテーブル as table, キー列名 as text) as table =>
let
    // キー列の値を取得
    古いキー = Table.Column(古いテーブル, キー列名),
    新しいキー = Table.Column(新しいテーブル, キー列名),

    // 削除された行（古いテーブルにはあるが新しいテーブルにはない）
    削除キー = List.Difference(古いキー, 新しいキー),
    削除行 = Table.SelectRows(古いテーブル, each List.Contains(削除キー, Record.Field(_, キー列名))),

    // 追加された行（新しいテーブルにはあるが古いテーブルにはない）
    追加キー = List.Difference(新しいキー, 古いキー),
    追加行 = Table.SelectRows(新しいテーブル, each List.Contains(追加キー, Record.Field(_, キー列名))),

    // 変更された行（両方にあるが内容が異なる）
    共通キー = List.Intersect({古いキー, 新しいキー}),

    // 各共通キーについて行を比較
    変更行 = List.Accumulate(
        共通キー,
        #table({"キー", "変更前", "変更後"}, {}),
        (状態, 現在キー) =>
            let
                古い行 = Table.SelectRows(古いテーブル, each Record.Field(_, キー列名) = 現在キー),
                新しい行 = Table.SelectRows(新しいテーブル, each Record.Field(_, キー列名) = 現在キー),
                行の比較 = Record.ToList(古い行{0}) <> Record.ToList(新しい行{0})
            in
                if 行の比較 then
                    Table.InsertRows(
                        状態,
                        Table.RowCount(状態),
                        {[キー = 現在キー, 変更前 = 古い行{0}, 変更後 = 新しい行{0}]}
                    )
```

```
                else
                    状態
    )
in
    [
        削除 = 削除行,
        追加 = 追加行,
        変更 = 変更行
    ]
```

3. 日付のカレンダーテーブルを生成する関数

```
// 指定期間のカレンダーテーブルを生成する関数
(開始日 as date, 終了日 as date) as table =>
let
    // 日数を計算
    日数 = Duration.Days(終了日 - 開始日) + 1,

    // 日付のリストを生成
    日付リスト = List.Dates(開始日, 日数, #duration(1, 0, 0, 0)),

    // リストをテーブルに変換
    初期テーブル = Table.FromList(日付リスト, Splitter.SplitByNothing(), {"日付"}, null,
ExtraValues.Error),
    型変換 = Table.TransformColumnTypes(初期テーブル, {{"日付", type date}}),

    // カレンダー属性を追加
    年追加 = Table.AddColumn(型変換, "年", each Date.Year([日付]), Int64.Type),
    月追加 = Table.AddColumn(年追加, "月", each Date.Month([日付]), Int64.Type),
    日追加 = Table.AddColumn(月追加, "日", each Date.Day([日付]), Int64.Type),

    曜日番号追加 = Table.AddColumn(日追加, "曜日番号", each Date.DayOfWeek([日付],
Day.Monday) + 1, Int64.Type),
    曜日追加 = Table.AddColumn(曜日番号追加, "曜日", each
        let
            曜日リスト = {"月", "火", "水", "木", "金", "土", "日"}
        in
            曜日リスト{[曜日番号] - 1},
        type text
    ),

    四半期追加 = Table.AddColumn(曜日追加, "四半期", each "Q" &
Text.From(Date.QuarterOfYear([日付])), type text),

    週末フラグ追加 = Table.AddColumn(四半期追加, "週末", each [曜日番号] >= 6, type
logical),

    月初フラグ追加 = Table.AddColumn(週末フラグ追加, "月初日", each Date.Day([日付]) = 1,
type logical),
```

```
    月末フラグ追加 = Table.AddColumn(月初フラグ追加, "月末日", each Date.Day([日付]) =
Date.DaysInMonth([日付]), type logical),

    日本年度追加 = Table.AddColumn(月末フラグ追加, "年度", each
        if Date.Month([日付]) >= 4 then
            Date.Year([日付])
        else
            Date.Year([日付]) - 1,
        Int64.Type
    ),

    結果 = 日本年度追加
in
    結果
```

✅ 第4章 チェックリスト

- [ ] M言語の基本構文を理解した
- [ ] UIで生成されたコードを読み、必要に応じて修正できる
- [ ] パラメータを使用した関数の作成と利用ができる
- [ ] M言語での条件分岐と繰り返し処理を理解した
- [ ] UIでは難しい処理をM言語でカスタマイズできる

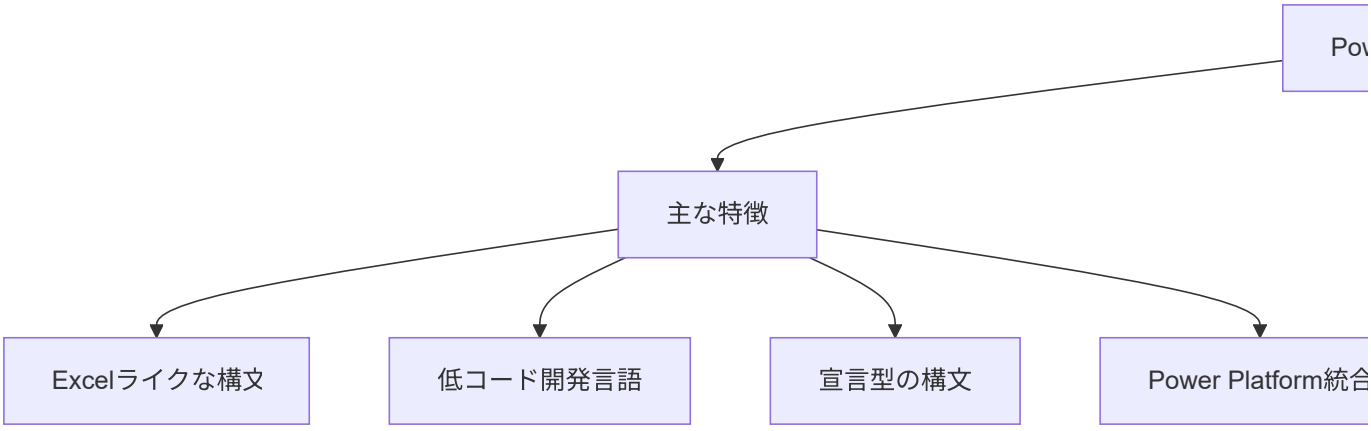
第5章: Power FX の基礎

5.1 Power FXとは

Power FXは、Microsoftが開発した低コードプログラミング言語で、主にPower Platformのアプリケーション内でユーザーインターフェイスやロジックを制御するために使用されます。Power BIでも一部の機能でPower FXの考え方が採用されています。

Power FXの概要

Power FXはExcel関数に似た構文を持ち、Excelに慣れたユーザーが比較的容易に習得できるよう設計されています。



Power FXの位置づけ

Power FXは、Power Platformのさまざまな製品で使用されますが、本書ではPower BIとの関連性に焦点を当てます。

Power BIにおけるPower FXの役割：

- 計算列の作成
- ビジュアルのフィルタリング
- 条件付き書式
- DAXの代替（一部のシナリオで）

> 💡 **ベテランの知恵袋**
> Power FXはPower AppsとPower Automateで広く使用されていますが、Power BIでは限定的な実装にとどまっています。しかし、MicrosoftはPower Platform全体でのPower FXの採用を拡大する方向に進んでいるため、これを学ぶことは将来的に価値があります。特にPower BI以外のPower Platform製品も使う予定がある場合は、Power FXの基礎を理解しておくと応用がきくでしょう。

5.2 基本的な構文とExcel関数との違い

Power FXはExcel関数に基づいていますが、いくつかの重要な違いがあります。

基本的な構文

Power FXの基本的な構文例：

```
// 単純な計算
Price * Quantity

// 条件式
If(Revenue > 1000000, "High", "Low")

// テキスト連結
FirstName & " " & LastName

// 日付計算
DateAdd(Today(), 30, Days)
```

ExcelとPower FXの違い

機能	Excel	Power FX
セル参照	A1, B2	列名やコントロール名を直接参照
列範囲	A1:A10	単一の列名（自動的に列全体）
数式の場所	セル内	プロパティまたは計算列
更新のタイミング	計算時（即時）	イベント駆動または定期更新
大文字/小文字の区別	区別なし	区別なし（大部分）
コレクション操作	限定的	強力なコレクション関数

主要な演算子

Power FXでは、以下の演算子が使用できます：

カテゴリ	演算子	説明	例
算術	+	加算	5 + 3
	-	減算	5 - 3
	*	乗算	5 * 3
	/	除算	5 / 3
	^	べき乗	5 ^ 2
比較	=	等しい	Price = 100
	<>	等しくない	Price <> 100
	>	より大きい	Price > 100
	>=	以上	Price >= 100
	<	より小さい	Price < 100
	<=	以下	Price <= 100
論理	&&	AND（かつ）	Price > 50 && Quantity > 10
	||	OR（または）	Category = "A" || Category = "B"

カテゴリ	演算子	説明	例
	!	NOT（否定）	!IsExpired
テキスト	&	連結	"Hello" & " " & "World"

データ型

Power FXでは以下の主要なデータ型が使用されます：

- **テキスト**: "Hello World"
- **数値**: 42, 3.14
- **真偽値**: true, false
- **日付と時刻**: Date(2023, 4, 1), Time(13, 30, 0)
- **レコード**: {Name: "John", Age: 30}
- **テーブル**: Table({ID: 1, Name: "John"}, {ID: 2, Name: "Jane"})
- **コレクション**: [1, 2, 3, 4, 5]

> ❓ **若手の疑問解決**
> **Q**: Power FXはExcelとほぼ同じように見えますが、同じExcel関数を使えばいいのですか？
> **A**: 基本的な構文はExcelと似ていますが、すべての関数がそのまま使えるわけではありません。また、Power FX固有の関数や機能も多くあります。特にテーブル操作やコレクション操作など、Excelにはない強力な機能がPower FXにはあります。最初は似ている部分から始めて、徐々にPower FX固有の機能を学ぶとよいでしょう。

5.3 Power BIでの活用シーン

Power BIではPower FXの完全な実装ではなく、一部の機能のみが利用可能です。以下の主要な活用シーンを見ていきましょう。

計算列での利用

Power BIのデータビューでは、Power FXに似た構文で計算列を作成できます：

```
// 単価を計算する計算列
売上金額 / 数量

// 顧客名を連結する計算列
[姓] & " " & [名]

// 条件分類を行う計算列
IF([売上金額] > 1000000, "大口",
  IF([売上金額] > 500000, "中口", "小口"))
```

条件付き書式での利用

ビジュアルの条件付き書式にもPower FX風の式が使用されます：

```
// 売上が予算を超えた場合に緑色にする
[売上] > [予算]
```

```
// 前年比で10%以上増加した場合に緑色にする
[当年売上] > [前年売上] * 1.1
```

フィルタリングと分析での利用

ビジュアルのフィルターやクイック測定でもPower FX的な式が使用されます：

```
// 東京支店のデータのみを表示
[支店] = "東京"

// 直近3ヶ月のデータのみを表示
[日付] >= TODAY() - 90
```

Power BIとPower Appsの統合

Power BIレポートにPower Appsビジュアルを埋め込むことで、より高度な対話性を実現できます。この場合、Power AppsでのPower FXが全面的に利用可能になります：

```
// Power BI視覚エフェクトから選択されたデータを取得
Selected = PowerBIIntegration.SelectedRow()

// 選択されたデータに基づいて表示するテキスト
TextBox.Text = "選択された商品: " & Selected.ProductName
```

> 💼 **プロジェクト事例**
> ある小売企業では、Power BIダッシュボードにPower Appsビジュアルを統合することで、分析と操作を一つのインターフェースで実現しました。ユーザーはPower BIで売上データを分析し、問題のある商品を特定したら、同じ画面内のPower Appsコンポーネントを使って価格調整や在庫発注を行えます。Power FXを使った条件付きロジックにより、売上減少商品には自動的に割引提案が表示されるようになっています。

5.4 計算列の作成

Power BIでPower FX的な構文を使って計算列を作成する方法を見ていきましょう。

基本的な計算列

Power BIのデータビューで計算列を追加する手順：

1. 「データ」ビューに切り替え
2. 対象のテーブルを選択
3. 「列ツール」タブ → 「新しい列」をクリック
4. 式を入力

以下は、基本的な計算列の例です：

単純な算術計算

```
// 売上総利益を計算する列
粗利益 = [売上金額] - [原価]
```

```
// 利益率を計算する列
利益率 = DIVIDE([売上金額] - [原価], [売上金額], 0)
```

テキスト操作

```
// 氏名を連結する列
氏名 = [姓] & " " & [名]

// 頭文字を抽出する列
頭文字 = LEFT([氏名], 1)
```

日付操作

```
// 年月を抽出する列
年月 = FORMAT([取引日], "yyyy/MM")

// 四半期を計算する列
四半期 = "Q" & QUARTER([取引日])
```

条件分岐を含む計算列

IF関数やスイッチ関数を使用した条件分岐：

```
// 売上規模による区分
売上区分 =
IF(
    [売上金額] >= 1000000, "大口",
    IF(
        [売上金額] >= 500000, "中口",
        IF(
            [売上金額] >= 100000, "小口",
            "極小"
        )
    )
)

// 商品カテゴリによる区分
価格帯 =
SWITCH(
    [商品カテゴリ],
    "パソコン", IF([単価] > 200000, "高級", "標準"),
    "スマートフォン", IF([単価] > 100000, "高級", "標準"),
    "アクセサリ", IF([単価] > 10000, "高級", "標準"),
    "その他"  // デフォルト値
)
```

計算列とmMジャーの違い

Power BIでは、計算列とメジャー（DAX）の使い分けが重要です：

特徴	計算列	メジャー
計算タイミング	データ読み込み時に計算	レポート表示時に計算
データの保存	結果が保存される	結果は一時的
フィルターコンテキスト	行コンテキストのみ	行とフィルターコンテキスト
適した用途	行ごとの計算、分類	集計、KPI、動的計算
例	顧客名の連結、カテゴリ分類	売上合計、平均単価、成長率

> 🚫 **失敗から学ぶ**
> あるプロジェクトでは、集計値（売上合計など）を計算列として実装してしまい、データモデルのサイズが不必要に大きくなり、パフォーマンスも低下しました。各行に同じ値が繰り返し保存されたためです。集計を行う場合は必ずメジャー（DAX）を使用しましょう。計算列は行レベルの変換や分類に限定すべきです。

高度な計算列のパターン

より複雑な計算を実現するパターンを見てみましょう：

条件付き集計

```
// 商品別の売上比率（その商品カテゴリ内での割合）
カテゴリ内シェア =
DIVIDE(
    [売上金額],
    CALCULATE(
        SUM([売上金額]),
        ALLEXCEPT(テーブル, [商品カテゴリ])
    ),
    0
)
```

前の行との比較

```
// 連続する日付間の売上変化
前日比 =
VAR 前日売上 = CALCULATE(
    SUM([売上金額]),
    FILTER(
        ALL(カレンダー),
        カレンダー[日付] = PREVIOUSDAY(カレンダー[日付])
    )
)
```

```
RETURN
    DIVIDE([売上金額] - 前日売上, 前日売上, 0)
```

5.5 フィルタリングと条件付き書式

Power BIでPower FX的な構文を使用してデータをフィルタリングしたり、条件付き書式を適用したりする方法を見ていきましょう。

ビジュアルのフィルタリング

Power BIでビジュアルにフィルターを適用する方法はいくつかあります：

基本的なフィルター

ビジュアルのフィルターペインを使った基本的なフィルター：

1. ビジュアルを選択
2. 「視覚化」ペインの「フィルター」セクションで条件を設定

基本的なフィルター条件の例：

- 特定の値（例：[支店] is "東京"）
- 値の範囲（例：[売上] is between 100000 and 500000）
- 上位N項目（例：Top 10 by [売上]）

相対日付フィルター

日付に関する相対的なフィルターも設定可能：

- 直近N日間（例：Last 30 days）
- 当月（例：This month）
- 前年同期（例：Same period last year）

```
// 直近90日間のデータを表示
[日付] >= TODAY() - 90

// 当四半期のデータを表示
[日付] >= STARTOFQUARTER(TODAY()) && [日付] <= ENDOFQUARTER(TODAY())
```

高度なフィルター

より複雑なフィルタリングには、「高度なフィルター」オプションを使用：

```
// 東京または大阪支店で、売上が100万円以上の取引
([支店] = "東京" || [支店] = "大阪") && [売上] >= 1000000

// 前年比で売上が増加している商品のみ
[当年売上] > [前年売上]
```

条件付き書式

Power BIでは、データの視覚的なハイライトに条件付き書式を使用できます：

値に基づく書式設定

テーブルや行列ビジュアルの条件付き書式：

1. ビジュアルを選択
2. 書式設定したいフィールドを右クリック
3. 「条件付き書式」を選択

条件付き書式の一般的な例：

```
// 売上目標の達成度に応じて色分け
[売上実績] / [売上目標] >= 1  // 緑色
[売上実績] / [売上目標] < 0.8  // 赤色
```

バックグラウンドとフォントカラー

条件付き書式では、背景色とフォント色を別々に設定できます：

```
// 利益率による背景色の設定
[利益率] >= 0.3  // 濃い緑
[利益率] >= 0.2  // 薄い緑
[利益率] < 0.1  // 赤

// 売上金額によるフォント色の設定
[売上金額] >= 1000000  // 青
[売上金額] < 100000  // グレー
```

データバー、カラースケール、アイコンセット

より視覚的な条件付き書式オプション：

- **データバー**: 値の大きさを横棒で表現
- **カラースケール**: 値の範囲をグラデーションで表現
- **アイコンセット**: 値に応じたアイコンを表示

```
// 売上データバーの設定
最小値: 0
最大値: MAX([売上金額])

// 前年比のカラースケール
下限: -0.1（赤）
中間: 0（黄）
上限: 0.1（緑）

// 目標達成度のアイコンセット
≥ 1（チェックマーク）
```

```
≥ 0.8（警告）
< 0.8（バツ印）
```

> 💡 **ベテランの知恵袋**
> 条件付き書式は強力なツールですが、使いすぎると視覚的なノイズが増えて情報が伝わりにくくなります。本当に強調したい例外的なデータポイントだけをハイライトするように心がけましょう。また、色覚多様性に配慮し、赤と緑だけでなく、形状や濃淡の違いも併用するとよいでしょう。

複数の条件による書式設定

複数の指標を組み合わせた条件付き書式も可能です：

```
// 売上と利益率の両方に基づく条件付き書式
IF(
    [売上金額] >= 1000000 && [利益率] >= 0.3,
    "優秀",  // 緑色
    IF(
        [売上金額] >= 500000 || [利益率] >= 0.25,
        "良好",  // 青色
        IF(
            [売上金額] < 100000 && [利益率] < 0.15,
            "要注意",  // 赤色
            "普通"  // 灰色
        )
    )
)
```

ビジュアルレベルでの表示制御

Power BIでは、ビジュアル全体の表示/非表示を条件によって制御することもできます：

1. ビジュアルを選択
2. 「書式」ペイン→「プロパティ」→「条件付き表示」をオン
3. 条件を設定

```
// 東京支店が選択された場合のみビジュアルを表示
SELECTEDVALUE(支店[支店名]) = "東京"

// 売上が予算を下回っている場合のみアラートビジュアルを表示
SUM(売上[金額]) < SUM(予算[金額])
```

✅ 第5章 チェックリスト

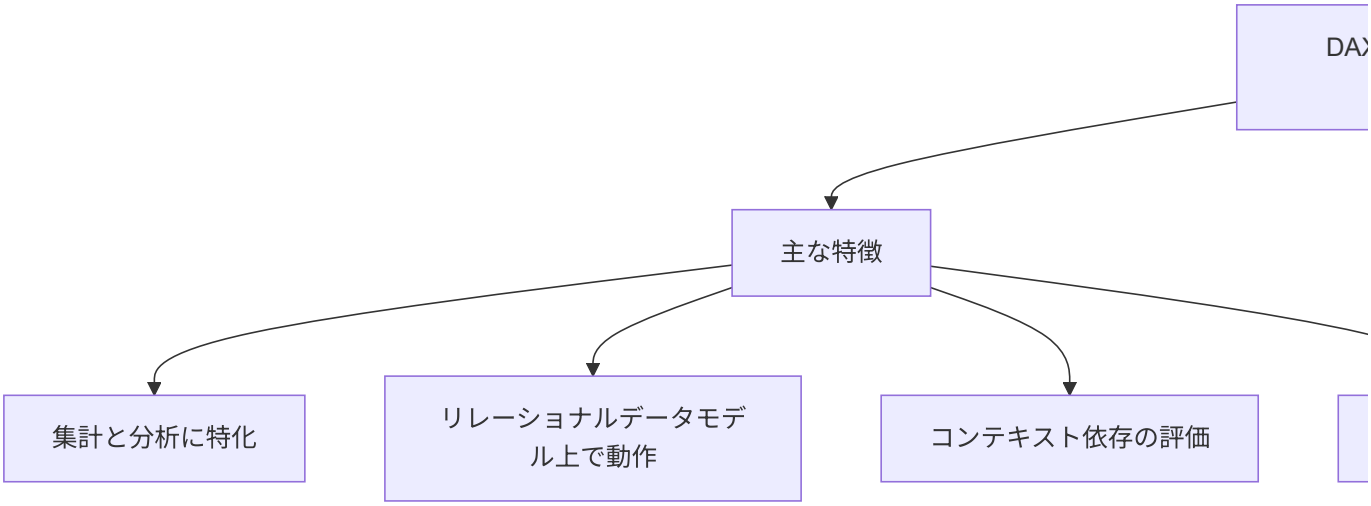
- [ ] Power FXの基本概念と用途を理解した
- [ ] Excel関数との類似点と相違点を把握した
- [ ] Power BIでのPower FX活用シーンを知った
- [ ] 計算列の作成と適切な利用法を学んだ
- [ ] フィルタリングと条件付き書式の適用方法を習得した

第6章: DAX入門

6.1 DAXの概念と基本構文

DAX（Data Analysis Expressions）は、Power BIとExcel Power Pivotで使用される数式言語です。データモデルに対して計算を行い、集計や分析を可能にします。

DAXとは何か



DAXは以下の主要な特徴を持っています：

- **列志向**: 列全体に対する操作を記述
- **コンテキスト依存**: 実行される状況（コンテキスト）によって結果が変わる
- **リレーショナル**: テーブル間の関係を活用した計算が可能
- **豊富な関数**: 集計、時間インテリジェンス、フィルタリングなど様々な関数を提供

DAXの構文基礎

DAX式の基本的な構造：

```
// 基本的なDAX式の構造
[計算結果] = 関数名(引数1, 引数2, ...)

// 例: 売上合計を計算するメジャー
[売上合計] = SUM(売上[金額])

// 例: 前年同期比を計算するメジャー
[前年同期比] =
DIVIDE(
    [当期売上],
    CALCULATE(
        [当期売上],
        SAMEPERIODLASTYEAR('カレンダー'[日付])
```

```
    ),
    0
)
```

> 💡 **ベテランの知恵袋**
> DAXを学ぶ際の最大の壁は「コンテキスト」の概念です。単純な構文自体はそれほど複雑ではありませんが、式がどのようなコンテキストで評価されるかを理解することが重要です。最初は単純なメジャーから始めて、徐々に複雑なケースに挑戦していくことをお勧めします。

DAXと他の言語との違い

DAXは独自の特性を持つ言語です。他の一般的なプログラミング言語とは異なる点があります：

特徴	DAX	一般的なプログラミング言語
評価モデル	コンテキスト依存	トップダウン評価
変数スコープ	グローバル（コンテキスト内）	ローカル/グローバル
制御構造	限定的（IF, SWITCH）	豊富（if-else, for, while, など）
関数	多数の組み込み関数	ユーザー定義関数が一般的
データ型	限定的（数値, テキスト, 日付, 真偽値, テーブル）	多様な型システム

DAXの基本的な演算子

DAXでは以下の演算子が使用できます：

カテゴリ	演算子	説明	例
算術	+	加算	5 + 3
	-	減算	5 - 3
	*	乗算	5 * 3
	/	除算	5 / 3
	^	べき乗	5 ^ 2
比較	=	等しい	[価格] = 100
	<>	等しくない	[価格] <> 100
	>	より大きい	[価格] > 100
	>=	以上	[価格] >= 100
	<	より小さい	[価格] < 100
	<=	以下	[価格] <= 100
論理	&&	AND（かつ）	[価格] > 50 && [数量] > 10
	||	OR（または）	[カテゴリ] = "A" || [カテゴリ] = "B"

DAXのデータ型

DAXでは以下の主要なデータ型が使用されます：

- **整数**: 1, 42, -10
- **小数**: 3.14, 2.5
- **通貨**: 金額値（内部的には数値として扱われる）
- **テキスト**: "Hello World"
- **真偽値**: TRUE, FALSE
- **日付/時刻**: 日付と時刻の値
- **テーブル**: 他のDAX式から返されるテーブル（表形式の結果）

6.2 計算列と計算メジャー

DAXには主に「計算列」と「計算メジャー（メジャー）」という2つの重要な計算タイプがあります。これらの違いを理解することがDAXを効果的に使うための鍵となります。

計算列とは

計算列は、データモデル内のテーブルに新しい列を追加し、各行に対して値を計算します。

特徴:

- テーブルの各行に対して計算される
- データの読み込み時に計算され、結果は保存される
- 行コンテキストで評価される
- 特定のテーブルに属する

作成方法:

1. Power BIデスクトップの「データビュー」でテーブルを選択
2. 「列ツール」タブ→「新しい列」をクリック
3. DAX式を入力

基本的な例:

```
// 単価を計算する計算列
単価 = 売上[売上金額] / 売上[数量]

// フルネームを作成する計算列
氏名 = 顧客[姓] & " " & 顧客[名]

// 年齢区分を設定する計算列
年齢区分 =
IF(
    顧客[年齢] < 20, "10代",
    IF(
        顧客[年齢] < 30, "20代",
        IF(
            顧客[年齢] < 40, "30代",
            IF(
                顧客[年齢] < 50, "40代",
                "50代以上"
            )
```

```
        )
    )
)
```

メジャーとは

メジャーは、ビジュアルやレポートのコンテキストに応じて動的に計算される集計です。

特徴:

- ビジュアル表示時に動的に計算される（結果は保存されない）
- フィルターコンテキストに応じて結果が変わる
- モデル全体にアクセスできる
- 特定のテーブルに所属するが、モデル全体で使用可能

作成方法:

1. Power BIデスクトップの「レポートビュー」でフィールドリストを表示
2. テーブルを右クリック→「新しいメジャー」を選択
3. DAX式を入力

基本的な例:

```
// 売上合計を計算するメジャー
売上合計 = SUM(売上[売上金額])

// 平均単価を計算するメジャー
平均単価 = AVERAGE(売上[単価])

// 前年同期比を計算するメジャー
前年同期比 =
DIVIDE(
    [当期売上],
    CALCULATE(
        [当期売上],
        DATEADD('カレンダー'[日付], -1, YEAR)
    ),
    0
)
```

計算列とメジャーの使い分け

計算列とメジャーの適切な使い分けは、パフォーマンスとモデルの設計に大きな影響を与えます。

用途	計算列	メジャー
分類や属性の追加	✓	✗
行レベルの計算	✓	✗
集計（合計、平均など）	✗	✓

用途	計算列	メジャー
フィルター依存の計算	✗	✓
比率や割合	✗	✓
時間インテリジェンス	✗	✓

> 🚫 **失敗から学ぶ**
> あるプロジェクトでは、すべての計算をメジャーではなく計算列として実装していました。例えば「売上合計」を各行に計算列として持たせていたため、データセットのサイズが不必要に大きくなり、更新時間も長くなっていました。集計値は必ずメジャーとして実装し、計算列は分類や属性の追加など、行レベルの計算に限定すべきです。

VAR構文の活用

複雑なDAX式では、中間結果を変数に格納するVAR構文が役立ちます：

```
// VAR構文を使用しない場合
利益率 =
DIVIDE(
    SUM(売上[売上金額]) - SUM(売上[原価]),
    SUM(売上[売上金額]),
    0
)

// VAR構文を使用する場合
利益率 =
VAR 売上合計 = SUM(売上[売上金額])
VAR 原価合計 = SUM(売上[原価])
VAR 利益 = 売上合計 - 原価合計
RETURN
    DIVIDE(利益, 売上合計, 0)
```

VAR構文の利点：

- コードの可読性向上
- 複雑な式の整理
- 同じ計算を複数回使用する場合の効率化
- デバッグのしやすさ

6.3 基本的な集計関数

DAXには様々な集計関数があり、データの分析に利用できます。

標準的な集計関数

最も基本的な集計関数は以下の通りです：

関数	説明	例
SUM	数値の合計を計算	`SUM(売上[金額])`

関数	説明	例
AVERAGE	数値の平均を計算	`AVERAGE(売上[単価])`
MIN	最小値を取得	`MIN(売上[日付])`
MAX	最大値を取得	`MAX(売上[金額])`
COUNT	値の数をカウント	`COUNT(売上[注文ID])`
COUNTROWS	行数をカウント	`COUNTROWS(売上)`
DISTINCTCOUNT	一意の値の数をカウント	`DISTINCTCOUNT(売上[顧客ID])`

使用例:

```
// 売上金額の合計
売上合計 = SUM(売上[金額])

// 平均単価
平均単価 = AVERAGE(売上[単価])

// 最も古い注文日
最古注文日 = MIN(売上[注文日])

// 最高額の注文
最大注文額 = MAX(売上[金額])

// 注文件数
注文件数 = COUNTROWS(売上)

// 一意の顧客数
ユニーク顧客数 = DISTINCTCOUNT(売上[顧客ID])
```

条件付き集計

特定の条件に基づいて集計を行う例：

```
// 東京支店の売上合計
東京支店売上 =
CALCULATE(
    SUM(売上[金額]),
    売上[支店] = "東京"
)

// 高額注文の件数（10万円以上）
高額注文件数 =
COUNTROWS(
    FILTER(
        売上,
        売上[金額] >= 100000
```

```
    )
)
```

CALCULATE関数の基本

CALCULATE関数は、DAXの最も重要な関数の一つで、フィルターコンテキストを修正できます：

```
CALCULATE(式, フィルター1, フィルター2, ...)
```

使用例:

```
// 前年の売上合計
前年売上 =
CALCULATE(
    SUM(売上[金額]),
    DATEADD('カレンダー'[日付], -1, YEAR)
)

// 特定商品カテゴリの売上合計
食品カテゴリ売上 =
CALCULATE(
    SUM(売上[金額]),
    商品[カテゴリ] = "食品"
)
```

> ❓ **若手の疑問解決**
> **Q**: CALCULATEとFILTERの違いは何ですか？
> **A**: CALCULATEはフィルターコンテキスト全体を修正し、その中で式を評価します。FILTERはテーブルに対してフィルターを適用し、結果のテーブルを返します。通常、FILTERはCALCULATE内で使用されることが多いですが、CALCULATE単独でシンプルなフィルターを適用することもできます。

累計と移動平均

累計や移動平均を計算する例：

```
// 年累計売上
年累計売上 =
CALCULATE(
    SUM(売上[金額]),
    DATESYTD('カレンダー'[日付])
)

// 四半期累計売上
四半期累計売上 =
CALCULATE(
    SUM(売上[金額]),
    DATESQTD('カレンダー'[日付])
)
```

```
// 過去3ヶ月移動平均
三ヶ月移動平均 =
AVERAGEX(
    DATESINPERIOD(
        'カレンダー'[日付],
        MAX('カレンダー'[日付]),
        -3,
        MONTH
    ),
    [日次売上]
)
```

ランキングと集合関数

ランキングや相対評価を行う関数：

```
// 製品ごとの売上ランキング
製品ランキング =
RANKX(
    ALL(商品[製品名]),
    [製品別売上],
    ,
    DESC
)

// 上位10%の顧客の売上合計
上位顧客売上 =
CALCULATE(
    SUM(売上[金額]),
    FILTER(
        VALUES(顧客[顧客ID]),
        [顧客ランキング] <= [顧客総数] * 0.1
    )
)
```

6.4 日付と時間の計算

日付と時間に関する計算は、ビジネス分析において非常に重要です。DAXには時間インテリジェンス関数と呼ばれる、日付と時間の分析に特化した関数群があります。

日付テーブルの重要性

DAXの時間インテリジェンス関数を使用するには、日付テーブルが必要です：

1. 連続した日付を含むテーブルを用意
2. Power BIで日付テーブルとして指定（「モデリング」タブ→「日付テーブルとしてマーク」）
3. 取引テーブルの日付列と日付テーブルをリレーションシップで接続

```
// DAXで日付テーブルを作成する例
カレンダー =
CALENDAR(
    DATE(2020, 1, 1),
    DATE(2025, 12, 31)
)
```

基本的な日付計算

```
// 今日の日付を取得
Today = TODAY()

// 現在の日時を取得
Now = NOW()

// 日付から年を抽出
Year = YEAR('カレンダー'[日付])

// 日付から四半期を抽出
Quarter = QUARTER('カレンダー'[日付])

// 日付から月を抽出
Month = MONTH('カレンダー'[日付])

// 日付から日を抽出
Day = DAY('カレンダー'[日付])

// 日付から曜日を取得（1=日曜日, 7=土曜日）
Weekday = WEEKDAY('カレンダー'[日付])
```

期間計算

期間（YTD, QTD, MTD）の集計を計算する関数：

```
// 年累計売上（Year-to-Date）
年累計売上 =
CALCULATE(
    SUM(売上[金額]),
    DATESYTD('カレンダー'[日付])
)

// 四半期累計売上（Quarter-to-Date）
四半期累計売上 =
CALCULATE(
    SUM(売上[金額]),
    DATESQTD('カレンダー'[日付])
)
```

```
// 月累計売上（Month-to-Date）
月累計売上 =
CALCULATE(
    SUM(売上[金額]),
    DATESMTD('カレンダー'[日付])
)
```

期間比較

異なる期間を比較する計算：

```
// 前年同月比較
前年同月比 =
DIVIDE(
    [当月売上],
    CALCULATE(
        [当月売上],
        SAMEPERIODLASTYEAR('カレンダー'[日付])
    ),
    0
)

// 前月比較
前月比 =
DIVIDE(
    [当月売上],
    CALCULATE(
        [当月売上],
        DATEADD('カレンダー'[日付], -1, MONTH)
    ),
    0
)

// 前年同期間比較（任意の期間）
前年同期間比 =
VAR 現在のフィルター = ALLSELECTED('カレンダー'[日付])
RETURN
DIVIDE(
    CALCULATE(
        SUM(売上[金額]),
        現在のフィルター
    ),
    CALCULATE(
        SUM(売上[金額]),
        DATEADD(現在のフィルター, -1, YEAR)
    ),
    0
)
```

> 💡 **ベテランの知恵袋**
> 日付計算のDAXコードは非常に複雑になりがちです。特に会計年度や特殊な期間が必要な場合は、事前に十分なカレンダーテーブルを設計することが重要です。年、四半期、月、週、会計年度などの属性をカレンダーテーブルに計算列として追加しておくと、メジャーの作成が格段に簡単になります。

期間の指定とフィルタリング

特定の期間にデータをフィルタリングする方法：

```
// 過去3ヶ月間のデータ
過去3ヶ月売上 =
CALCULATE(
    SUM(売上[金額]),
    DATESINPERIOD(
        'カレンダー'[日付],
        MAX('カレンダー'[日付]),
        -3,
        MONTH
    )
)

// 特定期間のデータ（例：2023年上半期）
2023年上半期売上 =
CALCULATE(
    SUM(売上[金額]),
    'カレンダー'[年] = 2023,
    'カレンダー'[月] <= 6
)

// 週末（土日）の売上のみ
週末売上 =
CALCULATE(
    SUM(売上[金額]),
    'カレンダー'[曜日名] IN {"土曜日", "日曜日"}
)
```

移動集計と移動平均

移動期間での集計や平均を計算：

```
// 直近3ヶ月の移動合計
移動3ヶ月売上 =
CALCULATE(
    SUM(売上[金額]),
    DATESINPERIOD(
        'カレンダー'[日付],
        MAX('カレンダー'[日付]),
        -3,
        MONTH
```

```
    )
)

// 直近12週の移動平均
移動12週平均 =
AVERAGEX(
    DATESINPERIOD(
        'カレンダー'[日付],
        MAX('カレンダー'[日付]),
        -12,
        WEEK
    ),
    [週次売上]
)
```

6.5 ExcelとPower BIでのDAX

DAXはExcel（Power Pivot）とPower BI両方で使用できますが、いくつかの違いがあります。

ExcelでのDAX

Excel Power Pivotでは、DAXを使用して計算列とメジャーを作成できます：

1. **アクセス方法**:
   - Power Pivotタブ→「Power Pivotウィンドウを管理」
   - データモデルビューで計算列やメジャーを追加
2. **利用可能な機能**:
   - 基本的なDAX構文と関数
   - 計算列とメジャーの作成
   - 一部の高度なDAX機能は制限あり
3. **表示方法**:
   - ピボットテーブルを使ってメジャーを表示
   - ピボットテーブルのフィルターコンテキストでメジャーを評価

ExcelでのDAXの例:

```
// Excel Power Pivotでの売上合計メジャー
[売上合計] = SUM([売上金額])

// Excel Power Pivotでの売上成長率メジャー
[売上成長率] =
DIVIDE(
    [当期売上] - [前期売上],
    [前期売上],
    0
)
```

Power BIでのDAX

Power BIでは、DAXの機能がより充実しています：

1. **アクセス方法**:
   - 「レポートビュー」または「データビュー」からメジャーや計算列を追加
   - DAXクエリエディタでクエリを直接記述可能
2. **利用可能な機能**:
   - すべてのDAX関数とパターン
   - 計算列、メジャー、計算テーブル
   - 高度なDAX関数も完全にサポート
3. **表示方法**:
   - 様々なビジュアルでメジャーを表示
   - Power BIのフィルターコンテキストでメジャーを評価

Power BIでのDAXの例:

```
// Power BIでの売上合計メジャー
売上合計 = SUM(売上[金額])

// Power BIでの売上成長率メジャー
売上成長率 =
VAR 当期売上 = [当期売上]
VAR 前期売上 = [前期売上]
RETURN
    DIVIDE(
        当期売上 - 前期売上,
        前期売上,
        0
    )
```

主な違いと注意点

機能	Excel	Power BI
DAX関数の範囲	一部制限あり	完全にサポート
パフォーマンス	中〜低（大規模データの場合）	高（Power BIエンジンで最適化）
計算テーブル	制限あり	完全にサポート
DAXクエリ	非対応	DAXクエリエディタで対応
ビジュアル化	ピボットテーブル中心	多様なビジュアルをサポート
データモデル	単一ファイル内	複数ソースから構築可能

> 💼 **プロジェクト事例**
> ある企業では、最初にExcel Power Pivotでダッシュボードを構築していましたが、データ量の増加とともにパフォーマンスの問題が発生しました。Power BIに移行することで、同じDAXコードを使いながらも、処理速度が大幅に改善し、より豊富なビジュアルでデータを表現できるようになりました。移行の際にはほとんどのDAXコードをそのまま再利用できたため、スムーズな移行が可能でした。

互換性を保つためのベストプラクティス

Excel Power PivotとPower BIの両方でDAXを使用する場合の推奨事項：

1. **共通関数の使用**: 両方のプラットフォームでサポートされている関数を優先して使用
2. **シンプルな式**: 複雑なネストを避け、VAR構文で式を整理
3. **命名規則の統一**: わかりやすく一貫した命名規則を使用
4. **モデル設計の一貫性**: 同様のテーブル構造とリレーションシップを維持
5. **互換性テスト**: 両方のプラットフォームで定期的にテスト

✅ 第6章 チェックリスト

- [ ] DAXの基本概念と構文を理解した
- [ ] 計算列とメジャーの違いと適切な使い分けを学んだ
- [ ] 基本的な集計関数の使い方を習得した
- [ ] 日付と時間の計算方法を理解した
- [ ] ExcelとPower BIでのDAXの違いと共通点を把握した

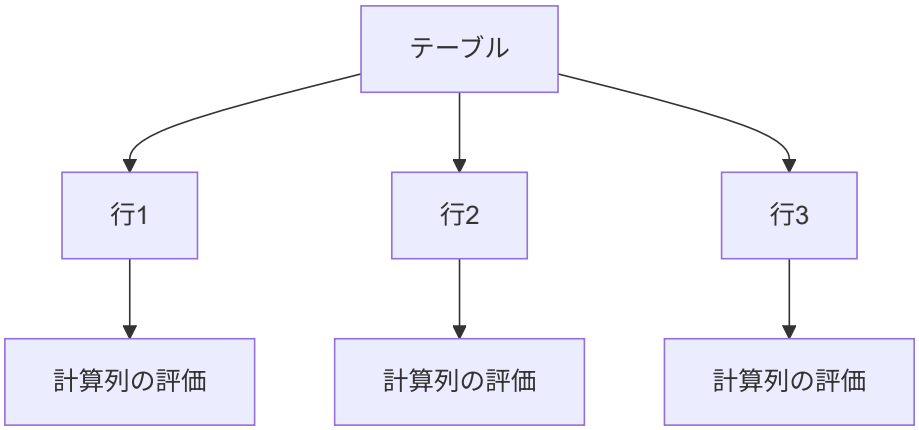
第7章: DAXによる高度なデータ分析

7.1 コンテキストの理解

DAXを真に理解するには「コンテキスト」の概念を把握することが不可欠です。DAXには主に「行コンテキスト」と「フィルターコンテキスト」の2種類のコンテキストがあります。

行コンテキストとは

行コンテキストは、テーブルの各行ごとに評価される環境です。主に計算列で使用されます。



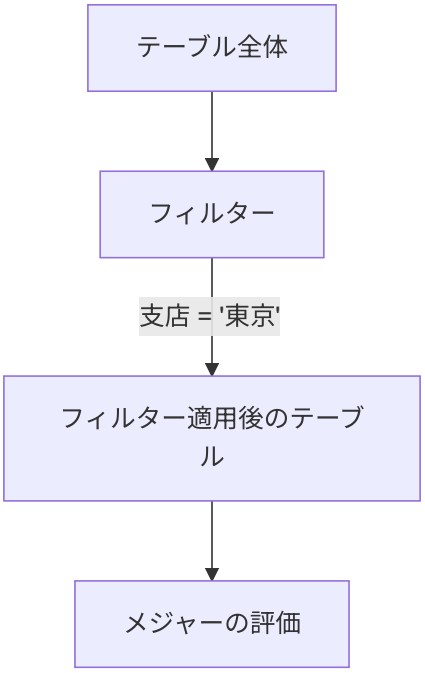
具体例:

```
// 売上テーブル内の計算列（行コンテキスト）
利益 = 売上[売上金額] - 売上[原価]
```

この計算では、各行の「売上金額」と「原価」の値が使用されます。テーブル内の各行に対して独立して計算が行われます。

フィルターコンテキストとは

フィルターコンテキストは、データのサブセット（一部）に対して評価される環境です。主にメジャーとビジュアルで使用されます。



具体例:

```
// 売上合計メジャー（フィルターコンテキスト）
売上合計 = SUM(売上[売上金額])
```

この計算では、ビジュアルやレポートのフィルターに応じて、対象となる行が変わります。例えば、「東京支店」のみが選択されている場合、東京支店のデータのみが集計されます。

> ❓ **若手の疑問解決**
> **Q**: 行コンテキストとフィルターコンテキストはどう使い分ければよいですか？
> **A**: 基本的に、各行で独立した計算を行いたい場合（例：利益率、カテゴリ分類など）は計算列（行コンテキスト）を使います。一方、集計値や視覚化するデータに応じて変わる計算（例：売上合計、成長率など）はメジャー（フィルターコンテキスト）を使います。

コンテキストの変換

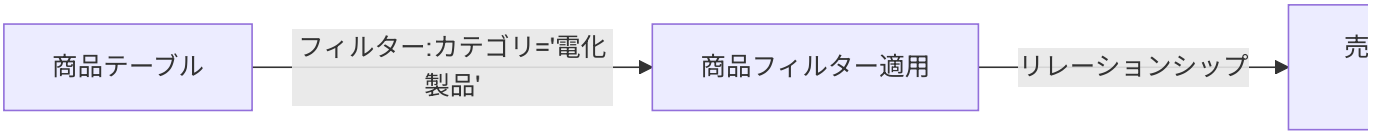
CALCULATE関数を使用すると、行コンテキストをフィルターコンテキストに変換できます：

```
// 行コンテキストでの計算例（計算列内）
商品別売上シェア =
DIVIDE(
    売上[売上金額],
    CALCULATE(
        SUM(売上[売上金額]),
        ALL(売上)
    ),
    0
)
```

この例では、CALCULATE関数とALL関数を使用して、行コンテキストを一時的にフィルターコンテキストに変換し、全体の売上合計を計算しています。

コンテキストの伝播

リレーションシップを通じて、フィルターコンテキストは関連テーブルに伝播します：



具体例:

```
// 電化製品カテゴリの売上合計
電化製品売上 =
CALCULATE(
    SUM(売上[売上金額]),
    商品[カテゴリ] = "電化製品"
)
```

この例では、商品テーブルに適用されたフィルターが、リレーションシップを通じて売上テーブルにも適用されます。

コンテキスト修正

CALCULATE関数を使用して、既存のフィルターコンテキストを修正できます：

```
// すべての支店の合計（ビジュアルのフィルターを無視）
全支店売上 =
CALCULATE(
    SUM(売上[売上金額]),
    ALL(支店)
)

// 前年同期の売上
前年同期売上 =
CALCULATE(
    SUM(売上[売上金額]),
    SAMEPERIODLASTYEAR('カレンダー'[日付])
)
```

7.2 フィルター関数の活用

DAXのフィルター関数は、計算のコンテキストを制御するための強力なツールです。適切に使用することで、複雑な分析を実現できます。

主要なフィルター関数

関数	説明	用途
ALL	すべてのフィルターを削除	全体値の計算
ALLEXCEPT	指定した列以外のフィルターを削除	部分的なフィルター解除
ALLSELECTED	ビジュアルレベルのフィルターのみ保持	選択コンテキストの保持
FILTER	条件に基づいてテーブルをフィルタリング	詳細なフィルタリング
CALCULATETABLE	フィルターコンテキスト内でテーブル式を評価	フィルターを適用したテーブル取得

ALL関数の使用例

ALL関数は、フィルターコンテキストを削除し、テーブル全体または列全体にアクセスできるようにします：

```
// 商品ごとの全社売上に対する比率
売上比率 =
DIVIDE(
    SUM(売上[売上金額]),
    CALCULATE(
        SUM(売上[売上金額]),
        ALL(売上)
    ),
    0
)

// 各商品カテゴリ内での商品別売上シェア
カテゴリ内シェア =
DIVIDE(
    SUM(売上[売上金額]),
    CALCULATE(
        SUM(売上[売上金額]),
        ALL(商品[商品名]),
        VALUES(商品[カテゴリ])
    ),
    0
)
```

ALLEXCEPT関数の使用例

ALLEXCEPT関数は、指定した列以外のフィルターを削除します：

```
// 商品カテゴリのフィルターを維持しつつ、他のフィルターを削除
カテゴリ内合計 =
CALCULATE(
    SUM(売上[売上金額]),
    ALLEXCEPT(商品, 商品[カテゴリ])
)
```

ALLSELECTED関数の使用例

ALLSELECTED関数は、ユーザーが選択したフィルターを維持しつつ、その他のフィルターを削除します：

```
// 現在のビジュアルで選択されている期間・地域内での売上比率
選択内比率 =
DIVIDE(
    SUM(売上[売上金額]),
    CALCULATE(
        SUM(売上[売上金額]),
        ALLSELECTED(売上)
    ),
    0
)
```

> 💡 **ベテランの知恵袋**
> ALL、ALLEXCEPT、ALLSELECTEDの違いを理解することは、DAXマスターへの重要なステップです。ALL関数は完全に「全部」を意味し、ALLEXCEPTは「これだけ残して全部クリア」、ALLSELECTEDは「ユーザーが選んだフィルターだけ尊重」と覚えるとわかりやすいでしょう。割合や比率を計算する際によく使います。

FILTER関数の使用例

FILTER関数は、条件に基づいてテーブルをフィルタリングします：

```
// 高額取引（10万円以上）の合計
高額取引合計 =
CALCULATE(
    SUM(売上[売上金額]),
    FILTER(
        売上,
        売上[売上金額] >= 100000
    )
)

// 前年比10%以上成長した商品の売上合計
成長商品売上 =
CALCULATE(
    SUM(売上[売上金額]),
    FILTER(
        VALUES(商品[商品名]),
        [前年比] >= 1.1
    )
)
```

複数のフィルター組み合わせ

複数のフィルター条件を組み合わせることで、より複雑な分析が可能になります：

```
// 東京支店の電化製品カテゴリの前年売上
東京支店電化製品前年売上 =
CALCULATE(
    SUM(売上[売上金額]),
    売上[支店] = "東京",
    商品[カテゴリ] = "電化製品",
    DATEADD('カレンダー'[日付], -1, YEAR)
)

// 高利益率（20%以上）かつ高額（10万円以上）の取引件数
優良取引件数 =
COUNTROWS(
    FILTER(
        売上,
        売上[利益率] >= 0.2 && 売上[売上金額] >= 100000
    )
)
```

フィルターの制御とリダイレクト

フィルターコンテキストの流れを制御するテクニック：

CROSSFILTER関数

リレーションシップのフィルター方向を一時的に変更します：

```
// 販売された商品の在庫金額（逆方向フィルター）
販売商品在庫額 =
CALCULATE(
    SUM(在庫[金額]),
    CROSSFILTER(
        売上[商品ID],
        在庫[商品ID],
        BOTH
    )
)
```

USERELATIONSHIP関数

非アクティブなリレーションシップを一時的に使用します：

```
// 出荷日ベースの売上（既定は注文日）
出荷日売上 =
CALCULATE(
    SUM(売上[売上金額]),
    USERELATIONSHIP(
        売上[出荷日],
        'カレンダー'[日付]
```

```
    )
)
```

7.3 時間インテリジェンス

DAXの時間インテリジェンス関数は、時系列データの分析に特化した関数群です。これらを活用することで、期間比較や累計計算などを効果的に行うことができます。

期間関数

特定の期間を定義する関数：

```
// 年の最初から現在までの期間（Year to Date）
年累計売上 =
CALCULATE(
    SUM(売上[売上金額]),
    DATESYTD('カレンダー'[日付])
)

// 四半期の最初から現在までの期間（Quarter to Date）
四半期累計売上 =
CALCULATE(
    SUM(売上[売上金額]),
    DATESQTD('カレンダー'[日付])
)

// 月の最初から現在までの期間（Month to Date）
月累計売上 =
CALCULATE(
    SUM(売上[売上金額]),
    DATESMTD('カレンダー'[日付])
)
```

DATEADD関数

指定した期間を加算または減算します：

```
// 前年同期間の売上
前年同期売上 =
CALCULATE(
    SUM(売上[売上金額]),
    DATEADD('カレンダー'[日付], -1, YEAR)
)

// 前月の売上
前月売上 =
CALCULATE(
    SUM(売上[売上金額]),
    DATEADD('カレンダー'[日付], -1, MONTH)
```

```
)

// 翌四半期の売上予測
翌四半期予測 =
CALCULATE(
    SUM(売上[売上金額]) * 1.1,  // 10%成長と仮定
    DATEADD('カレンダー'[日付], 1, QUARTER)
)
```

PARALLELPERIOD関数

並行期間を計算します：

```
// 前年同月の売上
前年同月売上 =
CALCULATE(
    SUM(売上[売上金額]),
    PARALLELPERIOD(
        'カレンダー'[日付],
        -12,
        MONTH
    )
)
```

> 🚫 **失敗から学ぶ**
> あるプロジェクトでは、会計年度（4月始まり）での分析が必要だったにもかかわらず、標準の時間インテリジェンス関数をそのまま使用していました。これにより、年度累計などの計算が誤っていました。会計年度や特殊なカレンダーを使用する場合は、カスタムカレンダーテーブルを作成し、会計年度の開始日を基準にした時間インテリジェンス計算を行うことが重要です。

SAMEPERIODLASTYEAR関数

前年の同じ期間を取得します：

```
// 前年同期間の売上
前年同期売上 =
CALCULATE(
    SUM(売上[売上金額]),
    SAMEPERIODLASTYEAR('カレンダー'[日付])
)
```

DATESINPERIOD関数

指定した期間内の日付を取得します：

```
// 過去3ヶ月間の売上
過去3ヶ月売上 =
CALCULATE(
    SUM(売上[売上金額]),
```

```
    DATESINPERIOD(
        'カレンダー'[日付],
        MAX('カレンダー'[日付]),
        -3,
        MONTH
    )
)
```

会計年度の処理

会計年度が暦年と異なる場合の処理：

```
// 4月始まりの会計年度の累計
会計年度累計 =
VAR 現在日付 = MAX('カレンダー'[日付])
VAR 会計年度開始日 =
    DATE(
        YEAR(現在日付) - IF(MONTH(現在日付) < 4, 1, 0),
        4,
        1
    )
RETURN
    CALCULATE(
        SUM(売上[売上金額]),
        FILTER(
            ALL('カレンダー'),
            'カレンダー'[日付] >= 会計年度開始日 &&
            'カレンダー'[日付] <= 現在日付
        )
    )
```

複合期間計算

より複雑な期間計算の例：

```
// 前年同期間との成長率
売上成長率 =
VAR 当期売上 = SUM(売上[売上金額])
VAR 前期売上 =
    CALCULATE(
        SUM(売上[売上金額]),
        SAMEPERIODLASTYEAR('カレンダー'[日付])
    )
RETURN
    DIVIDE(
        当期売上 - 前期売上,
        前期売上,
        0
    )
```

```
// 直近3ヶ月と前年同期間の比較
3ヶ月成長率 =
VAR 現在日付 = MAX('カレンダー'[日付])
VAR 直近3ヶ月 =
    DATESINPERIOD(
        'カレンダー'[日付],
        現在日付,
        -3,
        MONTH
    )
VAR 当期売上 =
    CALCULATE(
        SUM(売上[売上金額]),
        直近3ヶ月
    )
VAR 前期売上 =
    CALCULATE(
        SUM(売上[売上金額]),
        DATEADD(直近3ヶ月, -1, YEAR)
    )
RETURN
    DIVIDE(
        当期売上 - 前期売上,
        前期売上,
        0
    )
```

7.4 複雑な集計とKPI

ビジネスインテリジェンスの重要な部分は、複雑な集計と主要業績評価指標（KPI）の計算です。DAXを使用して、様々な高度な分析指標を作成できます。

複雑な集計の例

累積比率

売上の累積比率を計算する例：

```
// 売上の累積比率
累積比率 =
VAR 現在の日付 = MAX('カレンダー'[日付])
VAR 年始日 = STARTOFYEAR(現在の日付)
VAR 期間累計 =
    CALCULATE(
        SUM(売上[売上金額]),
        FILTER(
            ALL('カレンダー'),
            'カレンダー'[日付] >= 年始日 &&
            'カレンダー'[日付] <= 現在の日付
```

```
        )
    )
VAR 年間合計 =
    CALCULATE(
        SUM(売上[売上金額]),
        FILTER(
            ALL('カレンダー'),
            'カレンダー'[日付] >= 年始日 &&
            'カレンダー'[日付] <= ENDOFYEAR(現在の日付)
        )
    )
RETURN
    DIVIDE(期間累計, 年間合計, 0)
```

移動平均

売上の3ヶ月移動平均を計算する例：

```
// 3ヶ月移動平均
移動平均 =
VAR 現在日付 = MAX('カレンダー'[日付])
RETURN
    AVERAGEX(
        DATESINPERIOD(
            'カレンダー'[日付],
            現在日付,
            -3,
            MONTH
        ),
        [月次売上]
    )
```

ランキングと百分位数

売上ランキングと百分位数を計算する例：

```
// 製品別売上ランキング
売上ランキング =
RANKX(
    ALL(商品[商品名]),
    [商品別売上],
    ,
    DESC
)

// 売上百分位数（トップ何%に位置するか）
売上パーセンタイル =
VAR 総製品数 = COUNTROWS(VALUES(商品[商品名]))
VAR 現在ランク = [売上ランキング]
```

```
RETURN
    DIVIDE(現在ランク, 総製品数, 0)
```

一般的なKPIの計算例

売上関連KPI

```
// 売上達成率（目標に対する実績の割合）
売上達成率 =
DIVIDE(
    SUM(売上[売上金額]),
    SUM(目標[目標金額]),
    0
)

// 平均注文金額
平均注文金額 =
DIVIDE(
    SUM(売上[売上金額]),
    DISTINCTCOUNT(売上[注文ID]),
    0
)

// 顧客単価（LTV：顧客生涯価値の簡易版）
顧客単価 =
DIVIDE(
    SUM(売上[売上金額]),
    DISTINCTCOUNT(売上[顧客ID]),
    0
)
```

収益性KPI

```
// 粗利益
粗利益 =
SUM(売上[売上金額]) - SUM(売上[原価])

// 粗利益率
粗利益率 =
DIVIDE(
    [粗利益],
    SUM(売上[売上金額]),
    0
)

// 商品別貢献利益
貢献利益 =
VAR 売上 = SUM(売上[売上金額])
```

```
VAR 原価 = SUM(売上[原価])
VAR 変動費 = SUM(売上[変動費])
RETURN
    売上 - 原価 - 変動費
```

> ❓ **若手の疑問解決**
> **Q**: 多くのKPIを作成していますが、どのように整理すればよいですか？
> **A**: KPIは目的別にフォルダに整理するとよいでしょう。Power BIでは「表示フォルダ」機能を使って、関連するメジャーをグループ化できます。例えば「売上分析」「収益性」「顧客分析」などのフォルダに分類すると、必要なメジャーを見つけやすくなります。また、命名規則も重要で、例えば「KPI: 売上達成率」のようにプレフィックスをつけるとわかりやすくなります。

在庫関連KPI

```
// 在庫回転率（年間ベース）
在庫回転率 =
DIVIDE(
    SUM(売上[原価]),
    AVERAGEX(
        VALUES('カレンダー'[月]),
        [月末在庫金額]
    ),
    0
)

// 在庫日数
在庫日数 =
DIVIDE(
    [月末在庫金額],
    [日次平均原価],
    0
)
```

顧客関連KPI

```
// 顧客獲得コスト
顧客獲得コスト =
DIVIDE(
    SUM(マーケティング[費用]),
    COUNTROWS(
        FILTER(
            顧客,
            顧客[獲得日] >= MIN('カレンダー'[日付]) &&
            顧客[獲得日] <= MAX('カレンダー'[日付])
        )
    ),
    0
)
```

```
// 顧客継続率
顧客継続率 =
VAR 前期顧客 =
    CALCULATETABLE(
        VALUES(売上[顧客ID]),
        DATEADD('カレンダー'[日付], -1, YEAR)
    )
VAR 当期も購入した顧客 =
    CALCULATETABLE(
        INTERSECT(
            VALUES(売上[顧客ID]),
            前期顧客
        ),
        ALL('カレンダー')
    )
RETURN
    DIVIDE(
        COUNTROWS(当期も購入した顧客),
        COUNTROWS(前期顧客),
        0
    )
```

複合KPIと指標

複数の要素を組み合わせたより複雑なKPI：

```
// バランススコアカード指標（例：複合的な業績指標）
総合業績スコア =
VAR 売上スコア = [売上達成率] * 0.4  // 40%のウェイト
VAR 利益スコア = [利益達成率] * 0.3   // 30%のウェイト
VAR 顧客スコア = [顧客満足度] * 0.2   // 20%のウェイト
VAR 内部スコア = [業務効率化指標] * 0.1 // 10%のウェイト
RETURN
    売上スコア + 利益スコア + 顧客スコア + 内部スコア

// RFM分析スコア（顧客セグメンテーションのための指標）
RFMスコア =
VAR 最新購入 = [Recency_Score] * 0.35  // 最新購入日スコア（35%）
VAR 購入頻度 = [Frequency_Score] * 0.35 // 購入頻度スコア（35%）
VAR 購入金額 = [Monetary_Score] * 0.3   // 購入金額スコア（30%）
RETURN
    最新購入 + 購入頻度 + 購入金額
```

7.5 DAXパターンとベストプラクティス

DAXを効果的に活用するためには、一般的なパターンとベストプラクティスを理解することが重要です。ここでは、よく使われるDAXパターンとその実装方法を紹介します。

親子階層の集計

組織階層や製品カテゴリなどの親子関係を扱うパターン：

```
// 部門の売上（部門自体と所属する子部門の合計）
部門階層売上 =
VAR 現在部門 = VALUES(部門[部門ID])
VAR 子部門リスト =
    TREATAS(
        PATH(
            現在部門,
            部門[部門ID],
            部門[親部門ID]
        ),
        部門[部門ID]
    )
RETURN
    CALCULATE(
        SUM(売上[売上金額]),
        子部門リスト
    )
```

セミアディティブ集計

残高や在庫などの「セミアディティブ」値（単純に合計できない値）を集計するパターン：

```
// 月末在庫残高
月末在庫 =
VAR 現在期間 = MAX('カレンダー'[日付])
RETURN
    CALCULATE(
        SUM(在庫[残高]),
        FILTER(
            ALL('カレンダー'),
            'カレンダー'[日付] = ENDOFMONTH(現在期間)
        )
    )
```

相対位置の参照

テーブル内の相対位置（前の行、次の行など）を参照するパターン：

```
// 前月からの増減
前月比増減 =
VAR 現在月 = VALUES('カレンダー'[月])
VAR 当月売上 = SUM(売上[売上金額])
VAR 前月売上 =
    CALCULATE(
        SUM(売上[売上金額]),
        DATEADD(現在月, -1, MONTH)
    )
```

```
RETURN
    当月売上 - 前月売上
```

条件付き集計

特定の条件を満たすデータのみを集計するパターン：

```
// 大口顧客（購入金額100万円以上）の売上比率
大口顧客売上比率 =
VAR 大口顧客売上 =
    CALCULATE(
        SUM(売上[売上金額]),
        FILTER(
            VALUES(顧客[顧客ID]),
            [顧客生涯価値] >= 1000000
        )
    )
VAR 全体売上 = SUM(売上[売上金額])
RETURN
    DIVIDE(大口顧客売上, 全体売上, 0)
```

> 💡 **ベテランの知恵袋**
> DAXにはよく使われるパターンがたくさんあります。すべてを一から考えるのではなく、これらの既存パターンを参考にしてカスタマイズするのが効率的です。DAX Patterns（daxpatterns.com）のようなウェブサイトや、DAXのパターン集を扱った書籍を参考にすると、複雑な計算も簡単に実装できるようになります。

異なる単位での集計

異なる粒度や単位でデータを集計するパターン：

```
// 日次売上に対する月平均売上の比率
月平均に対する比率 =
VAR 日次売上 = SUM(売上[売上金額])
VAR 現在の月 = VALUES('カレンダー'[月])
VAR 月の日数 =
    CALCULATE(
        COUNTROWS(VALUES('カレンダー'[日付])),
        ALL('カレンダー'[日]),
        現在の月
    )
VAR 月間総売上 =
    CALCULATE(
        SUM(売上[売上金額]),
        ALL('カレンダー'[日]),
        現在の月
    )
VAR 月平均売上 = DIVIDE(月間総売上, 月の日数, 0)
```

```
RETURN
    DIVIDE(日次売上, 月平均売上, 0)
```

パフォーマンスのベストプラクティス

DAX計算のパフォーマンスを向上させるためのベストプラクティス：

1. **VAR変数の活用**: 複数回使用する式を変数に格納
2. **フィルターの最適化**: 可能な限り早い段階でデータをフィルタリング
3. **不要な計算の回避**: ISBLANK、ISFILTERED関数を使用して不要な計算をスキップ
4. **メモリの効率化**: 特に大規模データセットでは不要な列を排除
5. **計算列とメジャーの適切な使い分け**: 行レベル計算は計算列、集計はメジャーを使用

```
// パフォーマンスを考慮したDAXの例
最適化されたメジャー =
// 不要な計算をスキップ
IF(
    ISBLANK([ベースメジャー]),
    BLANK(),
    // 必要な場合のみ以下の計算を実行
    VAR ベース値 = [ベースメジャー]  // 複数回使う値を変数に格納
    VAR フィルター済み =
        // 早い段階でフィルタリング
        CALCULATETABLE(
            VALUES(商品[商品ID]),
            商品[状態] = "有効"
        )
    RETURN
        // メイン計算
        CALCULATE(
            ベース値,
            フィルター済み
        )
)
```

命名規則とドキュメント

効果的なDAX開発のための命名規則とドキュメント：

1. **命名規則**:
   - メジャー名には頭文字を大文字（例：「売上合計」）
   - 関連する測定値にはプレフィックスを使用（例：「KPI:」「財務:」「計算:」）
   - テーブル名と列名を区別するために角括弧を使用（例：`[売上合計]`）
2. **ドキュメント**:
   - 複雑なメジャーには説明（ディスクリプション）を追加
   - VAR変数に意味のある名前を付ける
   - 計算の目的と使用方法をコメントで記録

```
// 良い例：適切にドキュメント化されたDAX
売上シェア =
// 目的: 全体売上に対する現在フィルターの売上シェアを計算
// 使用方法: 製品や地域ごとのシェア分析に使用
VAR 現在の売上 = SUM(売上[金額])  // 現在のフィルターでの売上
VAR 全体売上 =
    CALCULATE(
        SUM(売上[金額]),
        ALL(商品),
        ALL(地域)
    )  // フィルターを除去した全体売上
RETURN
    DIVIDE(現在の売上, 全体売上, 0)  // シェアの計算（0除算防止）
```

✅ 第7章 チェックリスト

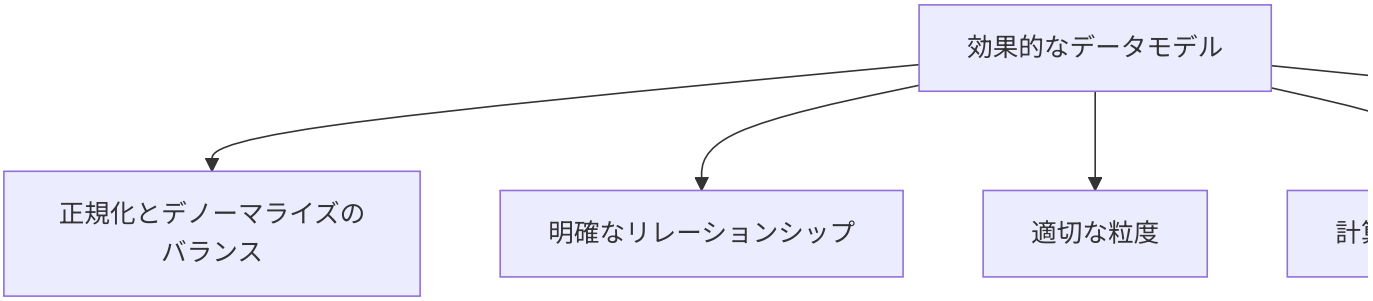
- [ ] 行コンテキストとフィルターコンテキストの違いを理解した
- [ ] 主要なフィルター関数（ALL, ALLEXCEPT, FILTER）の使い方を習得した
- [ ] 時間インテリジェンス関数を使って期間比較や累計計算ができる
- [ ] 複雑な集計とKPIの作成方法を学んだ
- [ ] 一般的なDAXパターンとベストプラクティスを把握した

第8章: データモデルの最適化

8.1 効率的なデータモデル設計

データモデルの設計は、Power QueryやDAXの効率的な活用の基盤となります。適切なデータモデル設計により、分析のパフォーマンスと柔軟性が大幅に向上します。

データモデリングの基本原則



スターとスノーフレークスキーマ

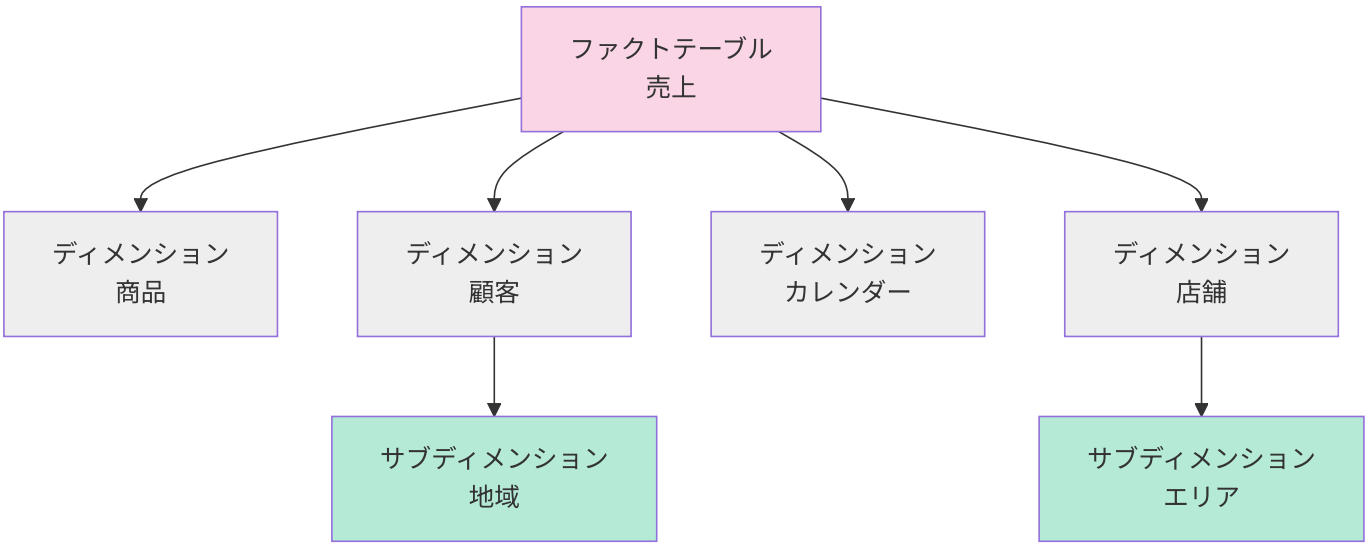
ビジネスインテリジェンスでは、スターとスノーフレークスキーマが一般的です。

スタースキーマ:

- 中央に「ファクトテーブル」（数値データ）
- 周囲に「ディメンションテーブル」（属性データ）
- シンプルで理解しやすい

スノーフレークスキーマ:

- スタースキーマの拡張
- ディメンションテーブルがさらに正規化される
- より複雑だが、データの重複が少ない



> 💡 **ベテランの知恵袋**
> 私の経験では、ほとんどの分析シナリオでスタースキーマが最適です。理解しやすく、パフォーマンスも良好です。ディメンションテーブルにいくつかの正規化（スノーフレーク的要素）を加えることはありますが、あまり複雑にしすぎないことがポイントです。まずシンプルなスタースキーマから始めて、必要に応じて拡張するアプローチをお勧めします。

効果的なモデル設計のポイント

1. 適切な粒度の選択

ファクトテーブルの各行が表す「詳細レベル」（粒度）を適切に決定することが重要です：

```
// 適切な粒度の例:
// 「売上」ファクトテーブル
売上ID（主キー）
日付（外部キー）
製品ID（外部キー）
顧客ID（外部キー）
店舗ID（外部キー）
販売数量
売上金額
原価
利益
```

粒度が細かすぎると不必要にテーブルが大きくなり、粗すぎると必要な分析ができなくなります。

2. 正規化とデノーマライズのバランス

- **正規化**: データの重複を減らし、整合性を高める
- **デノーマライズ**: 一部のテーブルを結合して、クエリのパフォーマンスを向上

Power BIでは、一般的に：

- ディメンションテーブルは適度に正規化する
- ファクトテーブルはなるべくシンプルに保つ
- 頻繁に一緒に使われる小さなテーブルは結合を検討

3. 計算のプッシュダウン

可能な限り、計算を早い段階（Power Queryレベル）で行うことでパフォーマンスが向上します：

- **Power Queryで行う計算**:
  - 単純な加算、乗算
  - 文字列連結
  - 日付の加工
  - グループ化による集計
- **DAXで行う計算**:
  - 動的な集計
  - コンテキスト依存の計算
  - 時間インテリジェンス
  - 比率や割合

データモデルの主要コンポーネント

1. ファクトテーブル

数値データ（メジャー）を含むテーブルで、通常は大量の行を持ちます：

- **トランザクション型**: 個々の取引（例：販売、購入）
- **スナップショット型**: 特定時点の状態（例：在庫、残高）
- **累積型**: 変化する状態（例：プロジェクト進捗）

2. ディメンションテーブル

属性情報を含むテーブルで、ファクトテーブルより小さいことが一般的：

- **日付/時間**: 暦、会計年度、休日情報など
- **製品**: 製品階層、属性、カテゴリなど
- **顧客**: 顧客情報、セグメント、地域など
- **組織**: 部門、チーム、従業員情報など

3. リレーションシップ

テーブル間の関連を定義し、フィルターの伝播を制御します：

- **1対多**: 最も一般的（例：1製品対多売上）
- **多対多**: 橋渡しテーブルを使用（例：多製品対多カテゴリ）
- **方向**: 単方向または双方向
- **アクティブまたは非アクティブ**

一般的なデータモデルのパターン

1. 集計テーブル

頻繁に使用される集計を事前計算してパフォーマンスを向上：

```
// 月次売上集計テーブルの例
月次売上集計 =
SUMMARIZECOLUMNS(
    'カレンダー'[年],
    'カレンダー'[月],
    商品[カテゴリ],
    "売上金額", SUM(売上[金額]),
    "販売数量", SUM(売上[数量]),
    "顧客数", DISTINCTCOUNT(売上[顧客ID])
)
```

2. 役割分割ディメンション

同じディメンションテーブルを異なる役割で使用：

```
// 日付テーブルを注文日と出荷日の両方として使用
// リレーションシップの設定:
// 1. 売上[注文日] → カレンダー[日付] (アクティブ)
// 2. 売上[出荷日] → カレンダー[日付] (非アクティブ)

// 出荷日ベースの売上（非アクティブリレーションシップを使用）
出荷日売上 =
CALCULATE(
    SUM(売上[金額]),
    USERELATIONSHIP(売上[出荷日], 'カレンダー'[日付])
)
```

3. 橋渡しテーブル

多対多関係を実現するための中間テーブル：

```
// 製品とカテゴリの多対多関係
// テーブル構成:
// 製品テーブル (製品ID, 製品名, etc.)
// カテゴリテーブル (カテゴリID, カテゴリ名, etc.)
// 製品カテゴリ橋渡しテーブル (製品ID, カテゴリID)

// リレーションシップ:
// 製品[製品ID] → 製品カテゴリ[製品ID]
// カテゴリ[カテゴリID] → 製品カテゴリ[カテゴリID]

// カテゴリごとの売上（橋渡しテーブル経由）
カテゴリ別売上 =
CALCULATE(
    SUM(売上[金額]),
    USERELATIONSHIP(売上[製品ID], 製品[製品ID]),
    TREATAS(
```

```
        VALUES(カテゴリ[カテゴリ名]),
        製品カテゴリ[カテゴリID]
    )
)
```

> 🚫 **失敗から学ぶ**
> あるプロジェクトでは、データモデルを設計する前にレポートの作成を始めてしまい、後からモデル変更が必要になり多くの時間を浪費しました。最初にユーザーの要件を理解し、必要な分析に適したデータモデルを設計することが重要です。「急がば回れ」でモデル設計に時間をかけることで、後工程が格段にスムーズになります。

8.2 Power Queryクエリの最適化

Power Queryは強力ですが、適切に使用しないとパフォーマンスの問題が発生する可能性があります。ここでは、Power Queryクエリを最適化するためのテクニックを紹介します。

クエリの最適化の基本原則

効率的なPower Queryのための基本原則：

1. **早期フィルタリング**: 必要なデータだけをロード
2. **変換順序の最適化**: 最もデータを減らす操作を先に実行
3. **折りたたみの活用**: データソースで処理を実行
4. **メモリ使用の最適化**: 大きなデータセットでの対策

早期フィルタリング

データソースからのデータ取得時に、必要なデータのみをフィルタリングすることが重要です：

```
// 非効率な例: すべてのデータを取得してから絞り込み
let
    ソース = Sql.Database("server", "database"),
    売上テーブル = ソース{[Schema="dbo",Item="売上"]}[Data],
    日付フィルタ = Table.SelectRows(売上テーブル, each [注文日] >= #date(2023, 1, 1))
in
    日付フィルタ

// 効率的な例: クエリでフィルタリング
let
    ソース = Sql.Database("server", "database"),
    クエリ = "SELECT * FROM dbo.売上 WHERE 注文日 >= '2023-01-01'",
    フィルタ済み = Sql.Database("server", "database", [Query=クエリ])
in
    フィルタ済み
```

SQLデータソースでは、ネイティブクエリを使用することで、データベースエンジンの処理能力を活用できます。

列の選択と削除

不要な列を早い段階で削除することでメモリ使用量と処理時間を削減できます：

```
// 非効率な例: すべての列をロードしてから不要な列を削除
let
    ソース = Excel.Workbook(File.Contents("data.xlsx"), null, true),
    売上シート = ソース{[Name="売上データ"]}[Data],
    列削除 = Table.RemoveColumns(売上シート, {"備考", "内部コード", "作成者"})
in
    列削除

// 効率的な例: 必要な列のみを選択
let
    ソース = Excel.Workbook(File.Contents("data.xlsx"), null, true),
    売上シート = ソース{[Name="売上データ"]}[Data],
    列選択 = Table.SelectColumns(売上シート, {"日付", "顧客ID", "製品ID", "金額", "数量"})
in
    列選択
```

クエリの折りたたみ

クエリの折りたたみとは、Power Queryの一連の操作をデータソースに送信される単一のクエリに変換する機能です。これにより、処理がデータソース側で実行され、パフォーマンスが向上します。

折りたたみが適用されているかを確認するには：

1. Power Queryエディタで「表示」タブ→「クエリ設定」を展開
2. ステップの横に「折りたたみ可能なステップ」アイコンがあるか確認

折りたたみを妨げる操作を避ける：

- カスタムM言語関数
- 複雑な条件式
- 特定のテーブル変換
- データ型の複雑な変更

```
// 折りたたみに適した例
let
    ソース = Sql.Database("server", "database"),
    売上テーブル = ソース{[Schema="dbo",Item="売上"]}[Data],
    日付フィルタ = Table.SelectRows(売上テーブル, each [注文日] >= #date(2023, 1, 1)),
    列選択 = Table.SelectColumns(日付フィルタ, {"注文ID", "顧客ID", "金額"}),
    金額ソート = Table.Sort(列選択,{{"金額", Order.Descending}})
in
    金額ソート

// 折りたたみが難しい例（カスタム関数使用）
let
    ソース = Sql.Database("server", "database"),
    売上テーブル = ソース{[Schema="dbo",Item="売上"]}[Data],
    カスタム列 = Table.AddColumn(売上テーブル, "消費税額", each
```

```
        if [税率区分] = "軽減" then [金額] * 0.08 else [金額] * 0.1
    )
in
    カスタム列
```

> ❓ **若手の疑問解決**
> **Q**: クエリの折りたたみが適用されているかどうかを確認する簡単な方法はありますか？
> **A**: Power Queryエディタの「表示」タブにある「ネイティブクエリ」ボタンをクリックすると、データソースに送信される実際のクエリを確認できます。また、各ステップを右クリックして「ネイティブクエリの表示」を選択すると、そのステップまでのクエリを確認できます。SQLのSELECT文が表示されれば、折りたたみが適用されている証拠です。

計算処理のプッシュダウン

可能な限り計算処理をデータソース側で実行することで、パフォーマンスを向上させることができます：

```
// 非効率な例: Power Queryでグループ化と集計
let
    ソース = Sql.Database("server", "database"),
    売上テーブル = ソース{[Schema="dbo",Item="売上"]}[Data],
    グループ化 = Table.Group(売上テーブル, {"製品カテゴリ", "年月"}, {
        {"売上合計", each List.Sum([金額]), type number},
        {"販売数量", each List.Sum([数量]), type number}
    })
in
    グループ化

// 効率的な例: SQLでグループ化と集計
let
    ソース = Sql.Database("server", "database"),
    クエリ = "
        SELECT
            製品カテゴリ,
            FORMAT(注文日, 'yyyy-MM') AS 年月,
            SUM(金額) AS 売上合計,
            SUM(数量) AS 販売数量
        FROM
            dbo.売上
            JOIN dbo.製品 ON 売上.製品ID = 製品.製品ID
        GROUP BY
            製品カテゴリ,
            FORMAT(注文日, 'yyyy-MM')
    ",
    集計済み = Sql.Database("server", "database", [Query=クエリ])
in
    集計済み
```

参照クエリと重複の削減

類似した処理を含む複数のクエリがある場合、共通部分を参照クエリとして抽出することで、処理の重複を避けることができます：

```
// 共通の基本クエリ
基本売上データ = let
    ソース = Sql.Database("server", "database"),
    売上テーブル = ソース{[Schema="dbo",Item="売上"]}[Data],
    型変換 = Table.TransformColumnTypes(売上テーブル, {
        {"注文日", type date},
        {"金額", type number},
        {"数量", Int64.Type}
    }),
    フィルタ = Table.SelectRows(型変換, each [注文日] >= #date(2023, 1, 1))
in
    フィルタ

// 参照クエリ1: 日時売上
日次売上 = let
    ソース = 基本売上データ,
    日次集計 = Table.Group(ソース, {"注文日"}, {
        {"売上合計", each List.Sum([金額]), type number},
        {"注文件数", each Table.RowCount(_), Int64.Type}
    })
in
    日次集計

// 参照クエリ2: カテゴリ別売上
カテゴリ別売上 = let
    ソース = 基本売上データ,
    カテゴリ集計 = Table.Group(ソース, {"製品カテゴリ"}, {
        {"売上合計", each List.Sum([金額]), type number},
        {"販売数量", each List.Sum([数量]), type number}
    })
in
    カテゴリ集計
```

パラメータの活用

クエリにパラメータを使用することで、柔軟性を高めつつ最適化することができます：

```
// 日付範囲パラメータの作成
// 1. 「ホーム」タブ → 「パラメータの管理」 → 「新しいパラメータ」
// 2. 名前: 開始日付, 型: 日付, 現在の値: 2023-01-01
// 3. 名前: 終了日付, 型: 日付, 現在の値: 2023-12-31

// パラメータを使用したクエリ
パラメータ付き売上データ = let
    ソース = Sql.Database("server", "database"),
    クエリ = "
```

```
        SELECT *
        FROM dbo.売上
        WHERE 注文日 BETWEEN @開始日付 AND @終了日付
    ",
    パラメータ = [
        開始日付 = 開始日付,
        終了日付 = 終了日付
    ],
    フィルタ済み = Sql.Database("server", "database", [Query=クエリ, Parameters=パラメータ])
in
    フィルタ済み
```

大規模データセットでの最適化

特に大規模なデータセットを扱う場合の最適化テクニック：

1. **行数の制限**: 開発中は行数を制限して高速化

```
制限付きデータ = Table.FirstN(売上データ, 1000)
```

2. **データタイプの最適化**: 適切なデータ型を使用してメモリ使用量を削減

```
型最適化 = Table.TransformColumnTypes(
    売上データ,
    {
        {"注文ID", Int64.Type},  // 整数IDには64ビット整数
        {"金額", Currency.Type},  // 金額には通貨型
        {"フラグ", Logical.Type}  // YesNoにはブール型
    }
)
```

3. **増分更新**: すべてのデータではなく変更部分のみを更新

```
// 最終更新日パラメータを使用
増分更新クエリ = "
    SELECT *
    FROM dbo.売上
    WHERE 最終更新日 > @前回更新日
"
```

8.3 DAX式の最適化

DAX式は、適切に記述することでパフォーマンスを大幅に向上させることができます。ここでは、DAX式を最適化するための主要なテクニックを紹介します。

DAX最適化の基本原則

効率的なDAX式を記述するための基本原則：

1. **フィルターコンテキストの最適化**: 早期にフィルタリングする
2. **計算の分割**: 複雑な式を分解する
3. **中間結果のキャッシュ**: VAR変数を活用する
4. **列計算と行計算の適切な使用**: SUMX vs SUM など
5. **適切なインデックスの活用**: 高カーディナリティの列を使わない

フィルターコンテキストの最適化

DAXのパフォーマンスは、適切なフィルターコンテキストによって大きく向上します：

```
// 非効率な例: 大きなテーブルをフィルタリングしてから集計
非効率な式 =
SUMX(
    FILTER(
        売上,
        売上[注文日] >= DATE(2023, 1, 1) &&
        売上[注文日] <= DATE(2023, 12, 31)
    ),
    売上[金額]
)

// 効率的な例: CALCULATEでフィルターコンテキストを設定してから集計
効率的な式 =
CALCULATE(
    SUM(売上[金額]),
    売上[注文日] >= DATE(2023, 1, 1),
    売上[注文日] <= DATE(2023, 12, 31)
)
```

VAR変数の活用

複雑な式では、中間結果をVAR変数に格納することで、重複計算を避け、可読性も向上します：

```
// 非効率な例: 同じ計算を複数回実行
非効率な式 =
DIVIDE(
    SUM(売上[金額]) - SUM(売上[原価]),
    SUM(売上[金額]),
    0
)

// 効率的な例: VAR変数を使用して中間結果を保存
効率的な式 =
VAR 売上合計 = SUM(売上[金額])
VAR 原価合計 = SUM(売上[原価])
VAR 利益 = 売上合計 - 原価合計
```

```
RETURN
    DIVIDE(利益, 売上合計, 0)
```

X関数の適切な使用

DAXには、標準の集計関数（SUM, AVERAGE など）とX関数（SUMX, AVERAGEX など）があります。適切に使い分けることが重要です：

```
// 標準関数: 単一列の集計に最適
売上合計 = SUM(売上[金額])

// X関数: 行ごとに計算が必要な場合に使用
利益合計 =
SUMX(
    売上,
    売上[金額] - 売上[原価]
)
```

X関数は強力ですが、パフォーマンスコストが高くなる可能性があります。不必要なX関数の使用は避けましょう。

CALCULATE関数の最適化

CALCULATE関数は強力ですが、正しく使用することが重要です：

```
// 非効率な例: 不要なCALCULATEのネスト
非効率な式 =
CALCULATE(
    CALCULATE(
        SUM(売上[金額]),
        ALL(製品)
    ),
    顧客[地域] = "東京"
)

// 効率的な例: 単一のCALCULATEにフィルターをまとめる
効率的な式 =
CALCULATE(
    SUM(売上[金額]),
    ALL(製品),
    顧客[地域] = "東京"
)
```

> 💡 **ベテランの知恵袋**
> DAXの最適化で最も重要なのは、メモリ内で処理されるデータ量を減らすことです。適切なフィルターを早期に適用し、不要なテーブルスキャンを避けることで、大幅なパフォーマンス向上が見込めます。特に大規模なファクトテーブルを扱う場合、この原則が重要になります。

高度なDAX最適化テクニック

1. SWITCH関数の活用

複数のIF文の代わりにSWITCH関数を使用すると、パフォーマンスが向上することがあります：

```
// 非効率な例: 多数のIF文
非効率な式 =
IF([カテゴリ] = "食品", [食品売上],
    IF([カテゴリ] = "衣料", [衣料売上],
        IF([カテゴリ] = "家電", [家電売上],
            [その他売上]
        )
    )
)

// 効率的な例: SWITCH関数
効率的な式 =
SWITCH(
    [カテゴリ],
    "食品", [食品売上],
    "衣料", [衣料売上],
    "家電", [家電売上],
    [その他売上]  // デフォルト値
)
```

2. SELECTEDVALUE関数の活用

単一値の取得にはSELECTEDVALUEを使用すると、エラーチェックが簡略化されます：

```
// 非効率な例: VALUESとIF文を使用
非効率な式 =
VAR カテゴリ値 = VALUES(製品[カテゴリ])
RETURN
    IF(
        COUNTROWS(カテゴリ値) = 1,
        FIRSTNONBLANK(カテゴリ値, 1),
        "複数または未選択"
    )

// 効率的な例: SELECTEDVALUE関数
効率的な式 =
SELECTEDVALUE(
    製品[カテゴリ],
    "複数または未選択"
)
```

3. 適切なイテレーター関数の選択

処理の目的に応じて、最適なイテレーター関数を選択します：

```
// 全行の合計が必要な場合はSUMX
利益合計 =
SUMX(
    売上,
    売上[金額] - 売上[原価]
)

// 条件を満たす行数を数える場合はCOUNTX
高額取引数 =
COUNTX(
    FILTER(
        売上,
        売上[金額] >= 100000
    ),
    売上[注文ID]
)

// 一意の値のみが必要な場合はVALUES
カテゴリリスト =
COUNTROWS(
    VALUES(製品[カテゴリ])
)
```

4. ISBLANK関数を使用した不要な計算の回避

条件に応じて計算をスキップすることで、パフォーマンスを向上させることができます：

```
条件付き計算 =
IF(
    ISBLANK([基本メジャー]),
    BLANK(),
    // 基本メジャーがある場合のみ実行される複雑な計算
    CALCULATE(
        [複雑な計算],
        FILTER(
            ALL(製品),
            製品[状態] = "有効"
        )
    )
)
```

8.4 メモリ使用量の削減テクニック

Power BIやExcelで大規模なデータセットを扱う場合、メモリ使用量の最適化が重要になります。ここでは、メモリ使用量を削減するための実践的なテクニックを紹介します。

データソースレベルでの最適化

データを読み込む前に、ソースレベルで最適化することでメモリ使用量を大幅に削減できます：

1. 不要な列の除外

```
// Power Queryでの不要列の除外
必要な列のみ = Table.SelectColumns(
    ソーステーブル,
    {"日付", "顧客ID", "製品ID", "金額", "数量"}
)
```

2. 適切なフィルタリング

```
// 必要なデータのみをフィルタリング
フィルタ済み = Table.SelectRows(
    テーブル,
    each [日付] >= #date(2023, 1, 1) and [日付] <= #date(2023, 12, 31)
)
```

3. 事前集計

```
// 詳細データではなく集計データをロード
月次集計 = Table.Group(
    売上データ,
    {"年", "月", "顧客ID", "製品カテゴリ"},
    {
        {"売上合計", each List.Sum([金額]), type number},
        {"販売数量", each List.Sum([数量]), type number}
    }
)
```

データ型の最適化

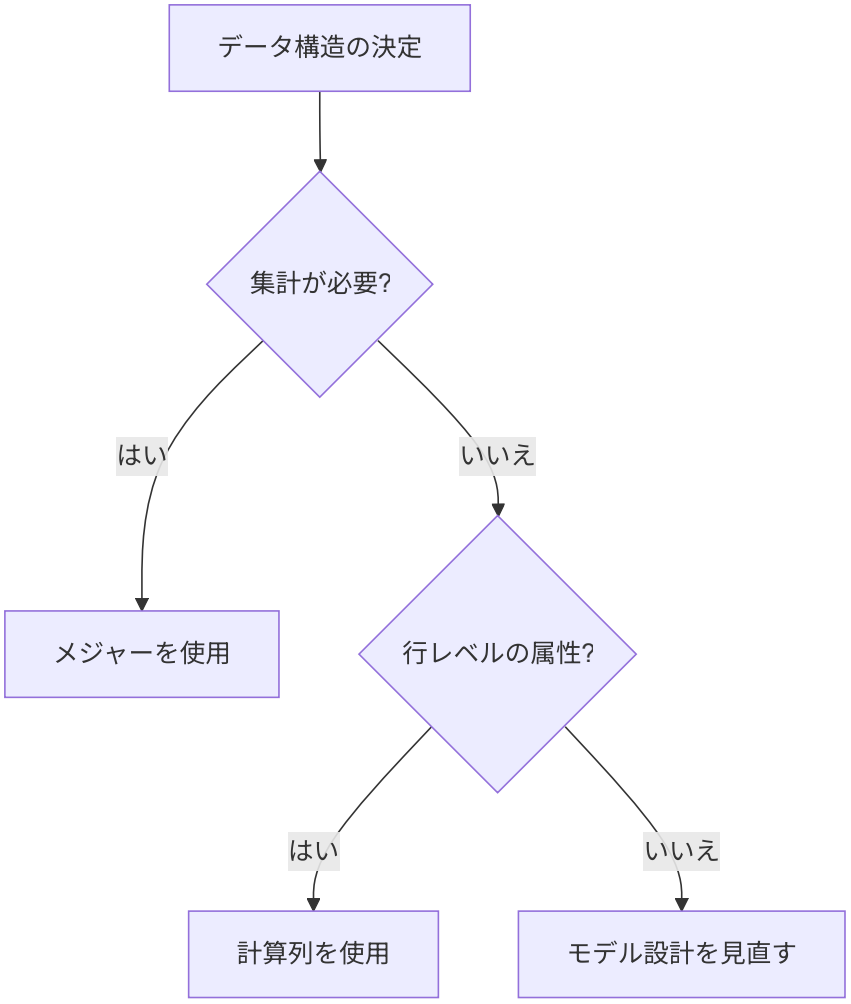
適切なデータ型を使用することで、メモリ使用量を削減できます：

```
// 最適なデータ型を指定
データ型最適化 = Table.TransformColumnTypes(
    テーブル,
    {
        {"ID", Int64.Type},           // 大きな整数
        {"小数", Decimal.Type},       // 精度が必要な小数
        {"コード", type text},        // テキスト
        {"日付", type date},          // 日付のみ（時刻なし）
        {"金額", Currency.Type},      // 通貨
        {"フラグ", Logical.Type}      // ブール値
    }
)
```

特に注意すべきデータ型の選択：

- 日付と時刻: 時刻が不要な場合は `type date` を使用
- 整数: 小さい値なら `Int16.Type`、大きい値なら `Int64.Type`
- 小数: 一般的な小数は `type number`、精度が重要なら `Decimal.Type`
- ブール値: `type text` の代わりに `Logical.Type` を使用

計算列とメジャーの適切な使用



計算列はデータモデルに保存され、メモリを消費します。一方、メジャーは必要なときだけ計算されます。

```
// 計算列として不適切な例（メモリ消費大）
売上合計列 = CALCULATE(SUM(売上[金額]))

// メジャーとして適切な例（メモリ消費小）
売上合計メジャー = SUM(売上[金額])
```

> 🚫 **失敗から学ぶ**
> あるプロジェクトでは、集計値をすべて計算列として実装していたため、データモデルのサイズが10

> 倍以上に膨れ上がってしまいました。レポートの読み込みに数分かかる状態になり、最終的にすべての集計をメジャーに変換し直す大規模なリファクタリングが必要になりました。集計値は常にメジャーとして実装することを徹底しましょう。

テーブルの正規化と非正規化のバランス

```
// 非正規化テーブル（メモリ使用量大）
非正規化売上 = Table.NestedJoin(
    売上, {"顧客ID"}, 顧客, {"顧客ID"}, "顧客情報", JoinKind.LeftOuter
)

// 正規化されたモデル（メモリ使用量小）
// 1. 売上テーブルと顧客テーブルを別々にロード
// 2. データモデルでリレーションシップを設定
```

一般的には、以下のアプローチが有効です：

- ファクトテーブル（売上など）は非正規化せず、高い粒度を維持
- ディメンションテーブル（顧客、製品など）は適度に正規化
- 頻繁に使用されるルックアップ値は非正規化を検討

計算グループの活用

Power BI（または表形式モデル）では、計算グループを使用して類似したメジャーを効率的に作成できます：

```
// 計算グループの例（時間比較の場合）
// 1. 「時間比較」という計算グループを作成
// 2. 「当期」「前年同期」「前期」などのアイテムを追加
// 3. 計算アイテムの式を定義

// 基本メジャー
売上金額 = SUM(売上[金額])

// 計算アイテムの定義
当期 = [売上金額]
前年同期 = CALCULATE([売上金額], SAMEPERIODLASTYEAR('カレンダー'[日付]))
前期 = CALCULATE([売上金額], DATEADD('カレンダー'[日付], -1, QUARTER))
対前年比 = DIVIDE([売上金額], CALCULATE([売上金額], SAMEPERIODLASTYEAR('カレンダー'[日付])),
0)
```

計算グループを使用することで、同様のパターンを持つ多数のメジャーをより効率的に管理できます。

キャッシュと集計テーブルの活用

頻繁に使用される複雑な計算結果を保存するために、集計テーブルを使用することができます：

```
// DAXで作成する集計テーブル
日次売上集計 =
```

```
SUMMARIZECOLUMNS(
    'カレンダー'[日付],
    製品[カテゴリ],
    "売上金額", SUM(売上[金額]),
    "販売数量", SUM(売上[数量]),
    "利益", SUM(売上[金額]) - SUM(売上[原価])
)
```

この手法は、特に以下の場合に有効です：

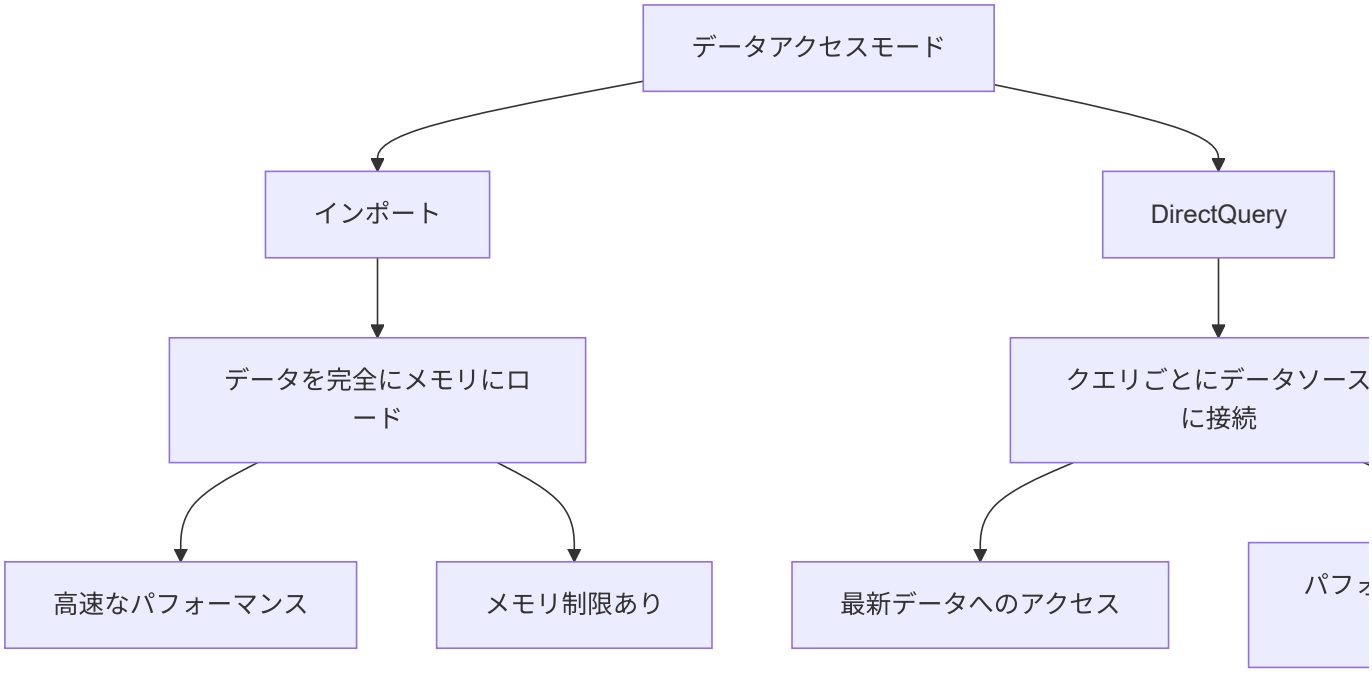
- 複雑な計算が頻繁に使用される
- ビジュアルの応答性が重要
- データが頻繁に更新されない

8.5 大規模データセットでのパフォーマンス向上策

大規模なデータセット（数百万行以上）を扱う場合、通常の最適化だけでは不十分なことがあります。ここでは、特に大規模データセットでのパフォーマンス向上策を紹介します。

インポートモードとDirectQueryの選択

Power BIでは、データの取得方法として「インポート」と「DirectQuery」の2つのモードがあります：



インポートモード:

- すべてのデータをメモリにロード
- 高速なパフォーマンス
- 更新が必要
- メモリ使用量が大きい

DirectQuery:

- データソースに直接クエリ
- 常に最新データ
- パフォーマンスが低いことがある
- メモリ使用量が少ない

複合モデル（Composite Model）の活用

Power BIでは、インポートモードとDirectQueryを組み合わせた「複合モデル」を作成できます：

```
// 複合モデルの一般的なアプローチ
// 1. 小さなディメンションテーブルはインポートモード
// 2. 大きなファクトテーブルはDirectQuery
// 3. テーブル間のリレーションシップを設定
```

複合モデルの利点：

- 頻繁に変更されるデータにはDirectQuery
- 安定したマスターデータにはインポート
- きめ細かなパフォーマンス制御

増分更新の設定

非常に大きなデータセットでは、増分更新を設定することで更新時間を短縮できます：

```
// 増分更新のためのパーティション分割条件（M言語）
分割条件 = Table.SelectRows(
    ソース,
    each [更新日時] > RangeStart and [更新日時] <= RangeEnd
)
```

増分更新の設定手順：

1. データセットにDateTimeタイプの更新/作成日時列を含める
2. Power Query内でパラメータを使用して範囲を設定
3. Power BI Desktopで増分更新ポリシーを構成
    - 履歴データを保持する期間（例：過去2年間）
    - 増分更新の頻度（例：過去7日間を更新）

増分更新の利点：

- 更新時間の大幅短縮
- ネットワーク負荷の軽減
- 最新データへのアクセス維持

> 💡 **ベテランの知恵袋**
> 増分更新は魔法のように思えますが、適切に設計しないと問題が生じることがあります。データに「更新日時」だけでなく「削除フラグ」も含めると、削除されたレコードも適切に処理できます。また、定期的に完全更新を行うスケジュールも組み込むとよいでしょう。

集計テーブルとAggregations機能

Power BIの「Aggregations」機能を使用すると、詳細データと集計データを組み合わせて最適なパフォーマンスを実現できます：

1. 詳細データテーブル（DirectQuery）を作成
2. 集計レベルのテーブル（インポート）を作成
3. 集計テーブルで「Aggregations」を構成

```
// 集計テーブルの定義例（DAX）
売上集計 =
SUMMARIZECOLUMNS(
    'カレンダー'[年], 'カレンダー'[月],
    製品[カテゴリ], 顧客[地域],
    "売上金額", SUM(売上[金額]),
    "販売数量", SUM(売上[数量])
)
```

Aggregationsの設定：

- 集計テーブル内の集計値（例：売上金額）
- 対応する詳細テーブルの列と集計方法（例：売上[金額]のSUM）
- 集計レベルの粒度（例：月、カテゴリ、地域レベル）

パフォーマンスアナライザーとクエリ診断

Power BIには、パフォーマンスを診断するためのツールが用意されています：

1. **パフォーマンスアナライザー**:
   - Power BI Desktop→「表示」タブ→「パフォーマンスアナライザー」
   - 各ビジュアルの読み込み時間を測定
   - DAXクエリとSQLクエリの詳細を表示
2. **DAXスタジオ**:
   - 外部ツール（無料）
   - 詳細なDAXクエリプロファイリング
   - クエリプランの分析
   - メモリ使用量のモニタリング

これらのツールを使用して、パフォーマンスのボトルネックを特定し、重点的に最適化することができます。

大規模データセットのためのハードウェア考慮事項

大規模データセットを扱う際は、ハードウェアの制約も考慮する必要があります：

1. **メモリ**:
   - Power BI Desktopは32ビット版で最大2GB、64ビット版で利用可能なメモリの最大70%
   - Power BI Serviceはプラン（容量）によって制限が異なる
2. **CPU**:
   - 複雑なDAX計算は複数のCPUコアを使用

- 並列処理能力がパフォーマンスに影響

3. **ストレージ**:
   - SSDはHDDよりも大幅に高速
   - 特にDirectQueryモードでは重要
4. **ネットワーク**:
   - DirectQueryの場合、ネットワーク帯域幅とレイテンシが重要
   - データゲートウェイの配置を最適化

✅ 第8章 チェックリスト

- [ ] 効率的なデータモデル設計の原則を理解した
- [ ] Power Queryクエリの最適化テクニックを習得した
- [ ] DAX式を最適化するための方法を学んだ
- [ ] メモリ使用量を削減するための手法を把握した
- [ ] 大規模データセットのパフォーマンス向上策を理解した

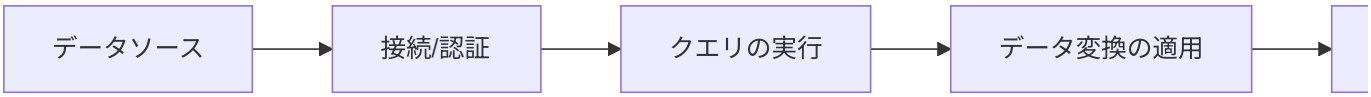
第9章: データ更新と自動化

9.1 データ更新の仕組み

データの更新は、最新の情報に基づいた分析を行うために重要です。Excel と Power BI では、データの更新方法が異なるため、それぞれの仕組みを理解しておく必要があります。

データ更新の基本概念

データ更新の基本的な流れは以下の通りです：



更新プロセスの詳細

1. **接続と認証**:
   - データソースへの接続確立
   - 必要に応じてログイン認証
   - 接続情報の保存/管理
2. **クエリの実行**:
   - Power Queryクエリがデータソースに送信
   - 可能な場合はクエリの折りたたみが適用
3. **データ変換の適用**:
   - Power Queryの変換ステップが順次実行
   - 結果がメモリに一時保存
4. **データモデルへのロード**:
   - 変換済みデータがデータモデルに読み込まれる
   - リレーションシップの確立
   - 圧縮アルゴリズムの適用（Power BIの場合）
5. **計算の更新**:
   - 計算列が計算される
   - メジャーはビジュアル表示時に計算

> 💡 **ベテランの知恵袋**
> 複雑なデータモデルでは、更新順序が重要になることがあります。例えば、ある変換済みテーブルが別のテーブルを参照している場合、参照元を先に更新する必要があります。Power Queryエディタでクエリの順序を確認し、必要に応じて調整しましょう。また、更新に時間がかかりそうな大きなクエリがある場合は、開発中はそのクエリを無効化して作業すると効率的です。

データソース別の更新特性

データソースの種類によって更新プロセスの性質が異なります：

ファイルベースのデータソース（Excel, CSV, Text）

```
// Excelファイルへの接続例
ソース = Excel.Workbook(File.Contents("C:\Data\Sales.xlsx"), null, true)
```

- **特徴**:
  - ファイルパスを指定して接続
  - フォルダ内の複数ファイルにも対応
  - ファイルパスは絶対パスまたは相対パス
- **更新時の考慮点**:
  - ファイルが存在する必要がある
  - 共有フォルダの場合はアクセス権が必要
  - ファイル形式やレイアウトが変更されると問題が発生する可能性

データベース（SQL Server, Oracle, MySQL）

```
// SQLデータベースへの接続例
ソース = Sql.Database("server", "database", [Query="SELECT * FROM Sales"])
```

- **特徴**:
  - サーバー名、データベース名、認証情報が必要
  - SQLクエリを直接実行可能
  - クエリの折りたたみが効率的に機能
- **更新時の考慮点**:
  - データベース接続の維持
  - 認証情報の管理
  - データベーススキーマの変更に注意

オンラインサービス（SharePoint, Dynamics 365, Salesforce）

```
// SharePointリストへの接続例
ソース = SharePoint.Tables("https://company.sharepoint.com/sites/data", [ApiVersion = 15])
```

- **特徴**:
  - APIを通じてデータにアクセス
  - 多くの場合OAuthによる認証
  - サービス固有の制限がある場合あり
- **更新時の考慮点**:
  - APIの利用制限
  - 認証トークンの期限
  - サービスの可用性

接続情報とプライバシー設定

データ更新時には、接続情報とプライバシー設定が重要な役割を果たします：

1. 接続情報の保存

Excel と Power BI はどちらも接続情報をファイル内に保存します：

- **保存される情報**:
  - サーバー名/ファイルパス
  - データベース名/テーブル名
  - 認証方法（Windows認証、データベース認証など）
  - ユーザー名（一部のケース）
- **パスワードの扱い**:
  - Power BI Desktop: パスワードを暗号化して保存可能
  - Excel: 通常はパスワードを保存しない（毎回入力）

2. プライバシーレベルとデータソースの設定

Power Queryでは、各データソースに対してプライバシーレベルを設定する必要があります：

```
// プライバシーレベルの設定
let
    ソース = Sql.Database("server", "database", [PrivacyLevel = "Public"])
in
    ソース
```

- **プライバシーレベルの種類**:
  - 公開（Public）: 誰でもアクセス可能なデータ
  - 組織（Organizational）: 組織内でアクセス可能なデータ
  - 個人（Private）: 個人のみがアクセス可能なデータ
- **影響**:
  - 異なるプライバシーレベルのデータソースを組み合わせると、バッファリングが発生しパフォーマンスが低下
  - 「高速データ結合を有効にする」オプションでこの制限をバイパス可能（セキュリティリスクを理解した上で）

9.2 Excelでの自動更新設定

Excelでのデータ更新は、Power Queryの「データの取得と変換」機能を通じて行われます。ここでは、Excelでのデータ更新の設定方法と自動化について説明します。

手動更新の方法

Excelでデータを手動で更新する方法はいくつかあります：

1. **個別クエリの更新**:
   - クエリを右クリック→「更新」を選択
   - または、「データ」タブ→「更新」→クエリを選択
2. **すべてのクエリの更新**:
   - 「データ」タブ→「すべて更新」を選択
   - これにより、ブック内のすべてのクエリが更新される
3. **クエリと接続パネルからの更新**:
   - 「データ」タブ→「クエリと接続を表示」
   - クエリを右クリック→「更新」を選択

自動更新のスケジュール設定

Excelでの自動更新は、以下の方法で設定できます：

1. ブックを開いたときに更新

```
// ブックを開いたときにデータを更新する設定
// 1. 「データ」タブ→「接続のプロパティ」
// 2. 「使用状況」タブで「ブックを開いたときに更新する」にチェック
```

2. 定期的な更新

Excel Desktopでの定期的な更新設定（制限あり）：

```
// 定期的な更新を設定
// 1. 「データ」タブ→「接続のプロパティ」
// 2. 「使用状況」タブで「次の間隔でデータを更新する」にチェック
// 3. 分単位で間隔を設定（例：60分ごと）
```

注意点：

- この機能はExcelが開いている間のみ動作
- 閉じている間は更新されない
- 一般的に短時間の間隔でのみ実用的

3. ExcelオンラインとOneDriveを使用した更新

Excelオンライン（Office 365）では、バックグラウンド更新に制限があります。主な方法は：

```
// ExcelオンラインとOneDriveでの更新
// 1. ファイルをOneDriveまたはSharePointに保存
// 2. Excelオンラインで開く
// 3. 手動で更新を実行
```

- Office 365管理者による追加設定が必要な場合あり
- Excel Desktopに比べて機能が限定的

> **🚫 失敗から学ぶ**
> あるプロジェクトでは、Excelファイルのデータを毎日自動更新するという要件がありました。Excel内の自動更新設定だけで実装しようとしましたが、PCを常にオンにしておく必要があり現実的ではありませんでした。最終的には、Power Automateを使ってファイルを開き、更新し、保存する自動化フローを構築することで解決しました。Excel単体での自動更新には限界があることを理解し、必要に応じて外部ツールを検討することが重要です。

VBAを使用した更新の自動化

より高度な更新の自動化には、VBA（Visual Basic for Applications）を使用できます：

```
' VBAを使用したデータ更新の例
Sub RefreshAllQueries()
```

```
    ' すべての接続を更新
    ActiveWorkbook.RefreshAll

    ' 完了メッセージ
    MsgBox "すべてのデータが更新されました。", vbInformation
End Sub

' 特定のクエリのみを更新
Sub RefreshSpecificQuery()
    ' 特定のクエリを名前で更新
    ActiveWorkbook.Queries("売上データ").Refresh

    ' 完了メッセージ
    MsgBox "売上データが更新されました。", vbInformation
End Sub
```

VBAマクロは以下のようなトリガーで実行できます：

- ボタンクリック
- ブックオープン時
- 特定のセル変更時
- 時間ベースのトリガー（Applicationイベント使用）

Power Automateによる更新の自動化

Microsoft Power Automateを使用すると、より柔軟なExcelデータの更新自動化が可能です：

```
// Power Automateフローの基本ステップ
// 1. トリガー設定（例：スケジュール、ファイル変更、メール受信など）
// 2. OneDrive/SharePoint上のExcelファイルを開く
// 3. Excelファイル内のテーブル/クエリを更新
// 4. ファイルを保存
// 5. 通知送信（オプション）
```

Power Automateの利点：

- クラウドベースで実行（PCを常時オンにする必要なし）
- 多様なトリガーとアクション
- 他のサービスとの連携が容易
- エラー処理とモニタリング機能

更新エラーのトラブルシューティング

Excel でのデータ更新エラーを解決するための一般的なアプローチ：

1. **接続の問題**:
   - ネットワーク接続を確認
   - VPNが必要な場合は接続を確認
   - ファイアウォール設定を確認
2. **認証の問題**:
   - ユーザー名とパスワードが正しいか確認

- 認証情報を更新
- アクセス権限を確認

3. **データソースの変更**:
   - データソースのパスや名前が変更されていないか確認
   - データベーススキーマの変更を確認
   - ファイル形式が変更されていないか確認
4. **更新履歴の確認**:
   - Excelでは直接確認する方法が限られている
   - 手動でログを取る仕組みをVBAで実装することを検討

9.3 Power BIのリフレッシュオプション

Power BIでは、Excelよりも多くのデータ更新（リフレッシュ）オプションが用意されています。ここでは、Power BI DesktopおよびPower BI Serviceでのリフレッシュオプションについて説明します。

Power BI Desktopでのリフレッシュ

Power BI Desktopでのデータ更新は主に手動で行います：

1. **手動リフレッシュの方法**:
   - 「ホーム」タブ→「更新」ボタン（すべてのクエリを更新）
   - 特定のクエリを右クリック→「更新」を選択
   - 「変換」タブ→「データの更新」→「更新」
2. **部分リフレッシュ**:
   - 「ホーム」タブ→「更新」の横の下向き矢印→「選択範囲を更新」
   - 特定のテーブルのみを選択して更新
3. **更新履歴の表示**:
   - Power BI Desktopでは更新履歴の確認機能は限定的
   - 更新中のステータスは表示される

Power BI Serviceでのリフレッシュ設定

Power BI Serviceでは、自動リフレッシュのための多様なオプションが用意されています：

1. **手動リフレッシュ**:
   - データセットの「...」メニュー→「更新」
   - データセット設定内の「今すぐ更新」ボタン
2. **スケジュールリフレッシュ**:
   - データセット設定→「スケジュール更新」を有効化
   - 更新頻度を設定（日次、週次など）
   - 特定の時間帯を選択（複数可）

```
// 一般的なスケジュール設定例
// 1. 平日の朝8時と夕方5時に更新
// 2. 週末は更新しない
// 3. 月末日の夜間に更新
```

3. **リフレッシュ通知**:

- 更新の成功/失敗をメールで通知
- メール受信者の設定
- 更新失敗時のみ通知するオプション

> **❓ 若手の疑問解決**
> **Q**: Power BI Serviceでのデータ更新回数に制限はありますか？
> **A**: はい、制限があります。Power BI Proライセンスでは1日最大8回のスケジュール更新が可能です。Power BI Premium容量を使用すると、1日最大48回（30分ごと）のスケジュール更新が可能になります。特に頻繁な更新が必要な場合は、Power BI Premiumの使用を検討するとよいでしょう。

ライセンスによる更新オプションの違い

Power BIのライセンスタイプによって、利用可能な更新オプションが異なります：

機能	Free	Pro	Premium Per User	Premium Capacity
スケジュール更新	×	○	○	○
最大更新頻度	-	8回/日	8回/日	8回/日
オンプレミスデータソース	×	○*	○*	○*
増分更新	×	○*	○	○
リフレッシュAPIアクセス	×	○	○	○

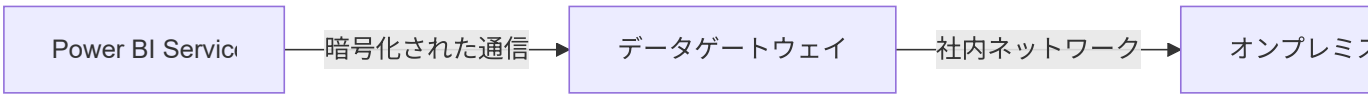
* オンプレミスデータソースの更新には、データゲートウェイの設定が必要

Power BI Premiumの主な利点：

- 更新頻度の上限が高い（30分間隔が可能）
- 大規模データセットのサポート
- より高度な更新機能（増分更新など）

データゲートウェイの設定と管理

オンプレミスのデータソース（社内サーバーのデータベースやファイル共有など）を使用している場合、データゲートウェイの設定が必要です：



1. **ゲートウェイのインストール**:
   - Power BI公式サイトからゲートウェイをダウンロード
   - Windows Server または高可用性のWindows PCにインストール
   - 会社のネットワーク内でデータソースにアクセス可能なマシンを選択
   - リカバリーキーを安全に保管
2. **ゲートウェイの構成**:
   - Power BI管理ポータルでゲートウェイを登録
   - 管理者とユーザーを割り当て
   - ゲートウェイのステータスモニタリングを設定
3. **データソースの設定**:
   - ゲートウェイに各データソースを構成
   - 接続情報と認証方法を指定
   - 資格情報の保存と更新

> 💡 **ベテランの知恵袋**
> データゲートウェイは組織の重要なインフラとなるため、適切な計画が必要です。本番環境では必ず「スタンダードモード」のゲートウェイを使用し、可能であれば冗長構成（複数のゲートウェイをクラスター化）を検討しましょう。また、ゲートウェイをインストールするマシンは専用のサーバーが理想的です。個人のPCでは、シャットダウンやスリープ状態になると更新が失敗します。リソース監視も重要で、特にメモリ使用量を定期的にチェックしてください。

オンプレミスとクラウドでの更新の違い

データソースの場所によって、更新プロセスが異なります：

クラウドデータソース

```
// クラウドデータソースの更新パス
Power BI Service --> インターネット --> クラウドデータソース（Azure SQL、Dynamics 365など）
```

- **メリット**:
  - ゲートウェイ不要
  - 直接接続が可能
  - 設定が簡単
- **注意点**:
  - インターネット接続に依存
  - クラウドサービスの可用性に影響される
  - 一部のクラウドサービスではAPIの呼び出し制限がある場合あり

オンプレミスデータソース

```
// オンプレミスデータソースの更新パス
Power BI Service --> インターネット --> データゲートウェイ --> 社内ネットワーク --> オンプレミスデータソース
```

- **メリット**:
  - 社内データへの安全なアクセス
  - VPNなしでアクセス可能
  - 複数のデータソースを1つのゲートウェイで管理可能
- **注意点**:
  - ゲートウェイの設定と管理が必要
  - ゲートウェイマシンの可用性がボトルネックになりうる
  - ネットワーク構成に依存

更新の失敗と監視

データ更新の失敗は避けられないため、適切な監視と対応が重要です：

1. **一般的な失敗の原因**:
   - データソースへの接続問題
   - 認証エラー（期限切れのパスワードなど）
   - データソースの変更（スキーマ変更など）
   - ゲートウェイの問題
   - リソース制限（メモリ不足など）
2. **モニタリングオプション**:
   - Power BI管理ポータルでのリフレッシュ履歴
   - メール通知の設定
   - PowerShellやAPIを使用した更新履歴の取得
   - カスタムモニタリングダッシュボードの作成
3. **失敗時の対応策**:
   - エラーメッセージの詳細確認
   - データゲートウェイのログ確認
   - 手動での更新テスト
   - 障害の根本原因の特定と解決

```
# PowerShellでデータセットの更新履歴を取得する例
Login-PowerBI
```

```powershell
$refreshHistory = Get-PowerBIDatasetRefreshHistory -DatasetId "dataset-id"
$refreshHistory | Format-Table -Property StartTime, EndTime, Status
```

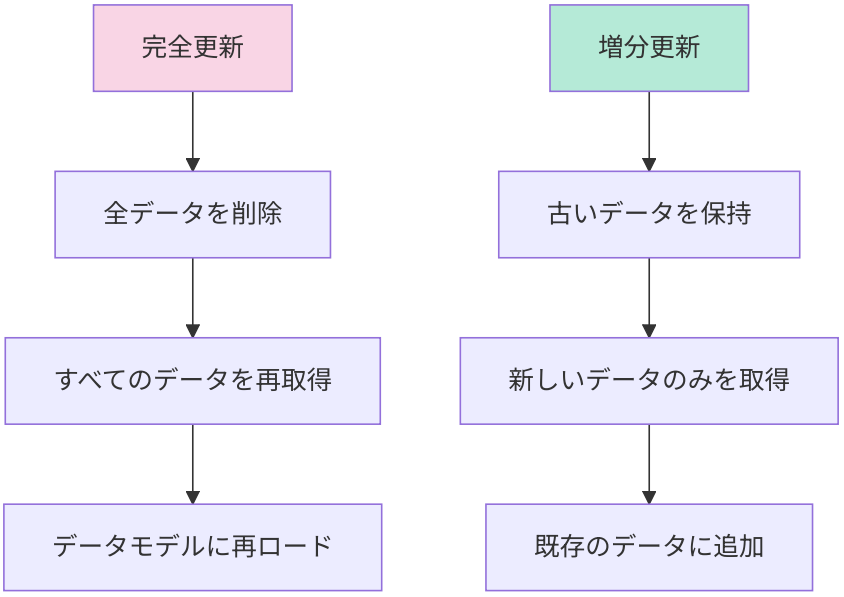
9.4 増分更新の設定

大規模なデータセットを扱う場合、毎回すべてのデータを更新するのではなく、新しいデータや変更されたデータのみを更新する「増分更新」が有効です。Power BIでは、この機能を使って更新時間を短縮し、効率的なデータ管理を実現できます。

増分更新の概要

増分更新は以下のメリットを提供します：

- **更新時間の短縮**: 変更部分のみを更新するため、処理時間が大幅に短縮
- **サーバー負荷の軽減**: データソースから取得するデータ量が減少
- **信頼性の向上**: 小さなバッチ処理のため失敗リスクが低減
- **履歴データの管理**: 古いデータの保持期間を自動管理



> **🚫 失敗から学ぶ**
> あるプロジェクトでは、3億行以上の販売データを含むPower BIレポートを作成していました。当初は完全更新を設定していたため、更新に8時間以上かかり、頻繁に失敗していました。増分更新を実装したところ、更新時間が15分に短縮され、信頼性も大幅に向上しました。しかし、実装時に日付パラメータの設定に問題があり、一部のデータが重複して取り込まれるというトラブルが発生しました。増分更新はパラメータの設定が重要で、実装前に小規模なテストデータでの検証が不可欠です。

増分更新の要件

増分更新を設定するには、以下の要件があります：

1. **Power BI PremiumまたはPower BI Pro**:
   - 基本的な増分更新はPro以上で利用可能
   - 高度な機能（より短い間隔など）はPremiumが必要

2. **日付/時刻列の存在**:
   - データに日付または日時列が必要
   - この列によってデータをフィルタリング
3. **Power Queryでのパラメータ化**:
   - RangeStart と RangeEnd パラメータの作成
   - これらを使用したクエリフィルタの実装

増分更新の実装手順

1. Power Queryでのパラメータ設定

Power BI Desktopでの準備：

```
// 1. パラメータの作成
// 「ホーム」→「パラメータの管理」→「新しいパラメータ」
// RangeStart: 日付型, デフォルト値 = #date(2020, 1, 1)
// RangeEnd: 日付型, デフォルト値 = #date(2023, 12, 31)

// 2. クエリにフィルタを適用
フィルタ済み = Table.SelectRows(
    前のステップ,
    each [更新日時] >= RangeStart and [更新日時] < RangeEnd
)
```

2. Power BI Desktopでの増分更新ポリシー設定

データモデル内のテーブルに増分更新ポリシーを設定：

1. テーブルを右クリック→「増分更新」を選択
2. 「増分更新」を有効化
3. 以下のパラメータを設定：
   - アーカイブデータ: 履歴データを保持する期間（例：5年間）
   - 増分データ: 各更新で取得するデータの期間（例：3日間）
   - 「Getを早期に処理」オプション（推奨）
   - 「古いデータの更新を除外」オプション（一般的に有効化）

```
// 一般的な増分更新設定例
アーカイブデータ: 5年（履歴データの保持期間）
増分データ: 3日（各更新で取得するデータの期間）
```

3. テスト方法

実装後のテスト方法：

1. Power BI Desktopでパラメータ値を変更して動作確認
2. Power BI Serviceにデータセットをパブリッシュ
3. 初回の完全更新を実行
4. 増分更新のスケジュールを設定

5. 更新ログを確認して期待通り動作しているか確認

高度な増分更新設定

より複雑なシナリオでの増分更新の実装方法：

1. 複数日付列での更新

```
// 作成日と更新日の両方を考慮したフィルタ
フィルタ済み = Table.SelectRows(
    前のステップ,
    each ([作成日] >= RangeStart and [作成日] < RangeEnd) or
         ([更新日] >= RangeStart and [更新日] < RangeEnd)
)
```

2. 削除処理の扱い

論理削除（削除フラグ）がある場合：

```
// 削除フラグを考慮したフィルタ
フィルタ済み = Table.SelectRows(
    前のステップ,
    each ([更新日] >= RangeStart and [更新日] < RangeEnd) or
         ([削除フラグ] = true and [削除日] >= RangeStart and [削除日] < RangeEnd)
)
```

物理削除の場合は「完全更新」を定期的に実行する設定が必要。

3. ディメンションテーブルの更新戦略

ディメンションテーブル（マスターデータ）の更新方法：

- **小規模な場合**: 完全更新を使用（シンプルさ優先）
- **大規模な場合**: 増分更新を実装
- **ハイブリッド**: 完全更新と増分更新の組み合わせ

```
// ディメンションテーブル更新の例（更新日のある場合）
ディメンションフィルタ = Table.SelectRows(
    マスターデータ,
    each [最終更新日] >= RangeStart and [最終更新日] < RangeEnd
)
```

増分更新のベストプラクティス

効果的な増分更新の実装のための推奨事項：

1. **日付インデックス**:
   - 増分更新に使用する日付列にはインデックスを作成する
   - 特にデータベースソースの場合に重要

2. **適切な期間設定**:
   - 増分データの期間は更新頻度よりも長くする
   - 例：毎日更新する場合、2-3日分のデータを取得
3. **正確な日付範囲**:
   - 日付ではなく日時を使用するとより精度が高い
   - タイムゾーンの違いに注意
4. **エラー処理**:
   - 増分更新の失敗に備えた対策を用意
   - 定期的な完全更新をスケジュール（例：月1回）
5. **パーティション分割**:
   - Premium容量では詳細なパーティション設定が可能
   - 年や月ごとのパーティションで管理効率向上

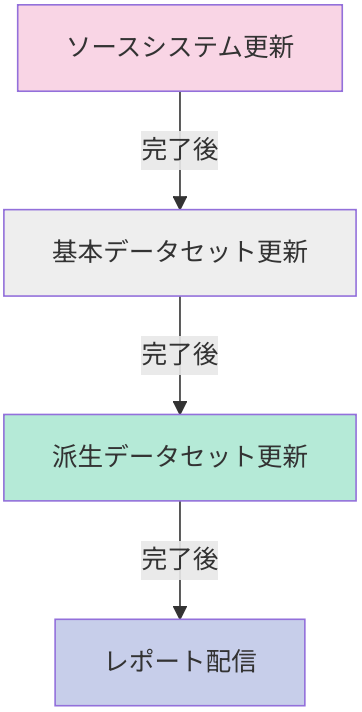
9.5 スケジュールと監視

データ更新の効果的な管理には、適切なスケジューリングと監視が重要です。ここでは、Excel と Power BI での更新スケジュールの設定と監視方法について説明します。

効果的な更新スケジュールの設計

最適な更新頻度とタイミングを設計する際の考慮事項：

1. **データの鮮度要件**:
   - リアルタイム性が必要か、日次更新で十分か
   - ビジネス上の意思決定タイミングとの整合性
2. **データソースの更新タイミング**:
   - 元データが更新される時間帯を把握
   - 更新後に適切な間隔を設定
3. **リソース要件**:
   - 大規模データセットの更新時間を考慮
   - システム負荷が低い時間帯を選択
4. **依存関係**:
   - データセット間の依存関係を特定
   - 更新順序を適切に設計



スケジュール設定のバリエーション

さまざまなスケジュールパターンとその適用シナリオ：

1. **日次更新**:
   - 一般的なビジネスレポート向け
   - 例：毎朝7時に前日データを更新
2. **複数回更新**:
   - 日中の意思決定に使用される場合
   - 例：午前9時、正午、午後3時、午後6時
3. **週次/月次更新**:
   - 長期傾向分析や要約レポート向け
   - 例：毎週月曜午前、または毎月1日
4. **カスケード更新**:
   - 複数のデータセットの順次更新
   - 例：基本データ→集計データ→ダッシュボード

> 💡 **ベテランの知恵袋**
> 更新スケジュールを設計する際は、「ビジネスリズム」に合わせることが重要です。例えば、朝のミーティングで使うレポートは、そのミーティングの少なくとも30分前には更新が完了しているべきです。また、データソースの更新タイミングも考慮してください。夜間バッチで更新されるデータソースに対して日中に何度も更新をスケジュールしても意味がありません。「更新の無駄」を減らすことで、システムリソースを効率的に使用できます。

Power BI Serviceでの詳細な更新設定

Power BI Serviceでの高度な更新設定：

1. **更新時間枠**:
   - 複数の更新時間を設定
   - 一日の特定の時間帯に集中させる
2. **エラー通知**:
   - 失敗時のメール通知の設定
   - 通知先の管理
3. **更新タイムアウト**:
   - デフォルト：2時間（スタンダード）、5時間（Premium）
   - 大規模データセットでは調整が必要
4. **休止オプション**:
   - 一時的に更新を停止
   - メンテナンス期間中などに使用

```
// Power BI 更新設定の一般的な例
スケジュールオプション:
- 毎日更新: 有効
- 時刻: 07:00, 12:00, 17:00
- タイムゾーン: (UTC+09:00) 大阪、札幌、東京
- プレミアム更新:
  - 時間ごとの更新: 無効
  - 増分更新: 有効
- メール通知: 失敗時のみ
```

更新ステータスの監視

更新プロセスの監視方法：

1. **Power BI Serviceでの監視**:
   - データセット設定→「更新履歴」
   - ステータス、開始時刻、終了時刻、期間を確認
   - エラーメッセージの詳細を確認
2. **Power BI 管理ポータル**:
   - テナント全体の更新状況を監視
   - 問題のあるデータセットを特定
3. **Power BI REST API**:
   - プログラムによる更新履歴の取得
   - カスタム監視ダッシュボードの作成

```
// PowerShellを使用した更新履歴の取得と分析
$refreshHistory = Get-PowerBIDatasetRefreshHistory -DatasetId "dataset-id"

// 更新時間の分析
$avgDuration = ($refreshHistory | Measure-Object -Property Duration -Average).Average

// 失敗した更新の特定
$failedRefreshes = $refreshHistory | Where-Object { $_.Status -eq "Failed" }
```

4. **メール通知**:
   - 更新の成功/失敗に関する通知
   - 設定可能な通知オプション

カスタム監視ソリューション

より高度な監視ニーズに対応するカスタムソリューション：

1. **Power BI 監視ダッシュボード**:
   - Power BI REST APIからデータを取得
   - 更新履歴、エラー率、平均時間などを表示
   - アラートと通知を設定
2. **Power Automateを使用した監視**:
   - 更新エラーを検出するフロー
   - 担当者への通知（メール、Teams、Slackなど）
   - エラー発生時の自動対応（再試行など）
3. **Azure Monitorとの統合**:
   - より包括的な監視
   - 長期的なトレンド分析
   - 詳細なログ分析

```
// Power Automateフローの例（更新失敗時）
トリガー: Power BI データセット更新失敗
アクション1: 詳細情報の取得（REST API経由）
アクション2: 条件分岐（最初の失敗か複数回の失敗か）
アクション3a: 最初の失敗→自動的に再試行
アクション3b: 複数回の失敗→担当者に通知
アクション4: インシデントログに記録
```

問題発生時の対応プロセス

更新エラーが発生した場合の体系的な対応方法：

1. **エラーの分類**:
   - 接続エラー（ネットワーク、認証）
   - データソースエラー（スキーマ変更、データ不整合）
   - リソースエラー（タイムアウト、メモリ不足）
   - 設定エラー（パラメータ不正など）
2. **調査手順**:
   - エラーメッセージの確認
   - 更新履歴のパターン分析
   - データゲートウェイのログ確認
   - 手動更新での再現確認
3. **解決アプローチ**:
   - 一時的な問題→再試行
   - 認証問題→資格情報の更新
   - データソース変更→クエリの修正

- リソース不足→更新スケジュールの最適化

4. **プロアクティブな対策**:
   - 定期的なメンテナンス手順の確立
   - 監視アラートの設定
   - ドキュメントの整備と共有

> 💼 **プロジェクト事例**
> ある製造業では、工場の生産データを毎時更新するPower BIレポートを運用していました。当初は単純な更新スケジュールでしたが、夜間の更新が頻繁に失敗していることが判明しました。調査の結果、夜間のバックアップ処理と競合していたことがわかりました。そこで、Power Automateを使って更新状況を監視し、失敗した場合に15分後に自動的に再試行するフローを構築しました。さらに、複数回失敗した場合はチームにTeamsで通知するようにしました。このアプローチにより、データの鮮度が向上し、手動介入の必要性が大幅に減少しました。

✅ 第9章 チェックリスト

- [ ] データ更新の基本的な仕組みを理解した
- [ ] Excelでの自動更新の設定方法を学んだ
- [ ] Power BIの様々なリフレッシュオプションを把握した
- [ ] 増分更新の設定と利点を理解した
- [ ] 効果的なスケジュール設定と監視の方法を習得した

第10章: 実践的なユースケースとソリューション

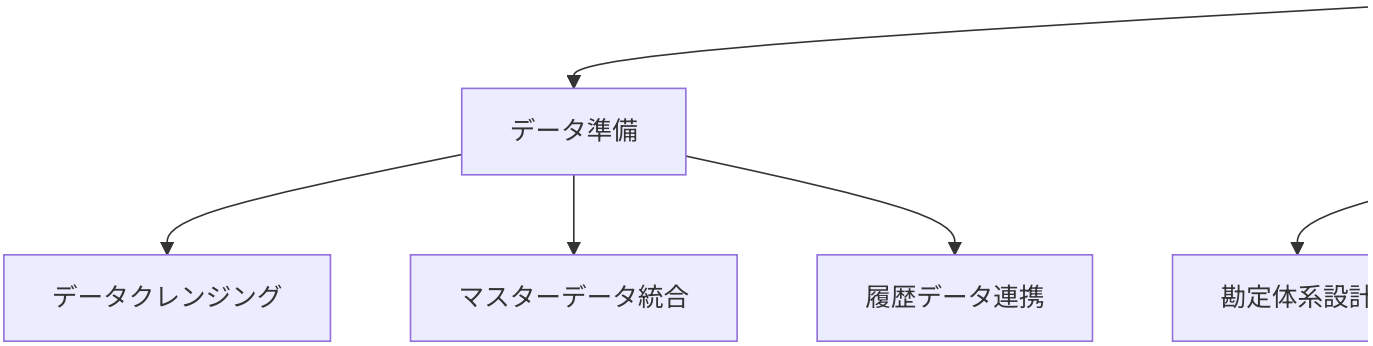
10.1 財務データ分析シナリオ

財務データは、多くの組織で最も重要なデータの一つです。Power QueryとDAXを活用して、効果的な財務分析ソリューションを構築する方法を見ていきましょう。

財務データ分析の課題と必要なスキル

財務データ分析における一般的な課題：

1. **データの一貫性と信頼性**:
   - 複数のソースからのデータ統合
   - 為替レートやマスターデータの一貫性確保
   - 履歴データとの整合性
2. **複雑な計算要件**:
   - 期間比較（前年同期、前期比など）
   - 累計・移動平均・予測
   - 部門間の配賦計算
3. **データモデルの設計**:
   - 勘定科目階層の表現
   - 多次元分析の実現
   - 時間インテリジェンスの活用



財務データの取得と変換

Power Queryを使った財務データの準備：

1. 複数ソースからのデータ統合

```
// 複数の財務データソースの統合例（M言語）
let
    // 総勘定元帳データ（SQL Server）
    元帳 = Sql.Database("FinanceServer", "GLDatabase"),
    GLデータ = 元帳{[Schema="dbo",Item="JournalEntries"]}[Data],
```

```
    // 予算データ（Excel）
    予算ファイル = Excel.Workbook(File.Contents("\\SharePoint\Finance\Budget.xlsx"),
null, true),
    予算データ = 予算ファイル{[Name="BudgetData"]}[Data],

    // 為替レートデータ（Web API）
    為替URL = "https://api.exchangerate.com/historical?key=xxxx",
    為替レート = Json.Document(Web.Contents(為替URL)),
    為替テーブル = Table.FromRecords(為替レート[rates]),

    // データ型の統一
    型変換GL = Table.TransformColumnTypes(GLデータ, {
        {"TransactionDate", type date},
        {"Amount", type number}
    }),
    型変換予算 = Table.TransformColumnTypes(予算データ, {
        {"Period", type date},
        {"BudgetAmount", type number}
    })
in
    // 各データソースは別々のクエリとして保存
    型変換GL
```

2. 勘定科目マスターの作成と管理

```
// 勘定科目階層の作成（M言語）
let
    // 勘定科目マスター
    ソース = Excel.Workbook(File.Contents("\\SharePoint\Finance\ChartOfAccounts.xlsx"),
null, true),
    勘定科目テーブル = ソース{[Name="AccountMaster"]}[Data],
    型変換 = Table.TransformColumnTypes(勘定科目テーブル, {
        {"AccountID", Int64.Type},
        {"AccountName", type text},
        {"AccountType", type text},
        {"ParentAccountID", Int64.Type},
        {"Level", Int64.Type}
    }),

    // 親勘定科目名の追加
    親科目追加 = Table.NestedJoin(
        型変換,
        {"ParentAccountID"},
        型変換,
        {"AccountID"},
        "ParentAccount",
        JoinKind.LeftOuter
    ),
```

```
    親科目展開 = Table.ExpandTableColumn(
        親科目追加,
        "ParentAccount",
        {"AccountName"},
        {"ParentAccountName"}
    ),

    // 完全なパス名の作成
    パス名追加 = Table.AddColumn(
        親科目展開,
        "AccountPath",
        each if [ParentAccountName] = null then [AccountName] else [ParentAccountName]
& " > " & [AccountName],
        type text
    )
in
    パス名追加
```

> 💡 **ベテランの知恵袋**
> 財務データのモデリングでは、勘定科目階層の設計が特に重要です。多くの組織では、「集計勘定」と「投稿勘定」が存在し、それらの階層関係を正確に表現する必要があります。勘定科目マスターは、ID、名称、親ID、レベル、勘定タイプなどの情報を含む完全なテーブルとして設計し、DAXの親子関数（PATH, PATHITEM, PATHCONTAINS）を活用することで、複雑な階層分析も可能になります。また、勘定体系は時間とともに変化するため、有効期間の管理も検討すべきポイントです。

3. データのクレンジングと標準化

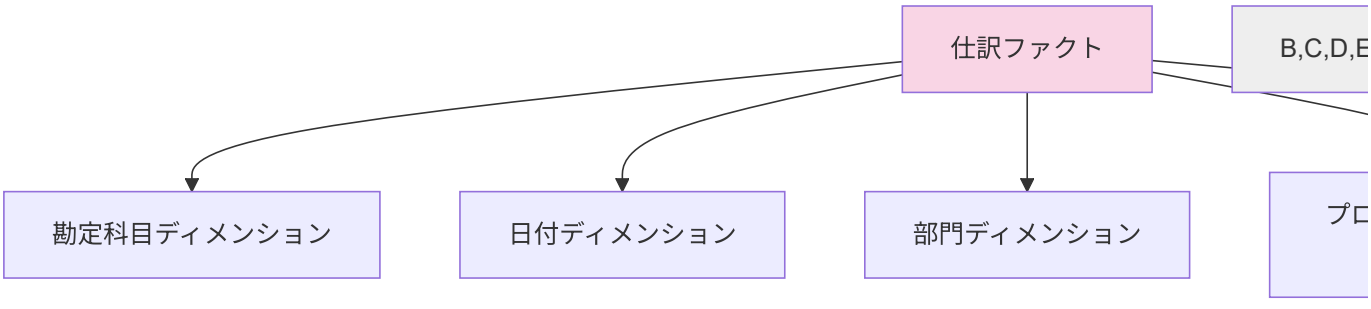
```
// 仕訳データのクレンジング（M言語）
let
    ソース = 元帳クエリ,

    // 不要な列の削除
    列選択 = Table.SelectColumns(
        ソース,
        {"JournalID", "TransactionDate", "AccountID", "DepartmentID", "Amount",
"Currency", "Description"}
    ),

    // 空白や異常値のフィルタリング
    空白除去 = Table.SelectRows(列選択, each
        [JournalID] <> null and
        [AccountID] <> null and
        [Amount] <> null
    ),

    // 日付の標準化（日付部分のみ取得）
    日付標準化 = Table.TransformColumns(
        空白除去,
        {{"TransactionDate", each Date.From(_), type date}}
```

```
    ),

    // 為替レートの適用で金額を標準化
    為替適用 = Table.NestedJoin(
        日付標準化,
        {"Currency", "TransactionDate"},
        為替テーブル,
        {"Currency", "Date"},
        "為替情報",
        JoinKind.LeftOuter
    ),
    為替展開 = Table.ExpandTableColumn(
        為替適用,
        "為替情報",
        {"Rate"},
        {"ExchangeRate"}
    ),
    金額標準化 = Table.AddColumn(
        為替展開,
        "StandardAmount",
        each if [Currency] = "JPY" then [Amount] else [Amount] * [ExchangeRate],
        type number
    )
in
    金額標準化
```

財務データモデルの構築

効果的な財務分析のためのデータモデル設計：

1. スタースキーマの基本構造



2. 日付テーブルの拡張

財務分析に特化した日付テーブルの作成：

```
// 財務カレンダーの作成（DAX）
財務カレンダー =
VAR 開始日 = DATE(2018, 1, 1)
```

```
VAR 終了日 = DATE(2025, 12, 31)
VAR 日数 = DATEDIFF(開始日, 終了日, DAY) + 1
RETURN
    ADDCOLUMNS(
        CALENDAR(開始日, 終了日),
        "年", YEAR([Date]),
        "月", MONTH([Date]),
        "日", DAY([Date]),
        "年月", FORMAT([Date], "yyyy/MM"),
        "月名", FORMAT([Date], "MMM"),
        "四半期", "Q" & FORMAT([Date], "Q"),
        "会計年度", IF(MONTH([Date]) >= 4, YEAR([Date]), YEAR([Date])-1),
        "会計四半期", "Q" &
            IF(
                MONTH([Date]) >= 4,
                MOD(CEILING(MONTH([Date]) - 3, 3) / 3, 4) + 1,
                MOD(CEILING(MONTH([Date]) + 9, 3) / 3, 4) + 1
            ),
        "会計月", IF(MONTH([Date]) >= 4, MONTH([Date]) - 3, MONTH([Date]) + 9),
        "会計年度月", FORMAT(
            IF(MONTH([Date]) >= 4, YEAR([Date]), YEAR([Date])-1),
            "0000"
        ) & "/" &
        FORMAT(
            IF(MONTH([Date]) >= 4, MONTH([Date]) - 3, MONTH([Date]) + 9),
            "00"
        ),
        "曜日", WEEKDAY([Date]),
        "曜日名", FORMAT([Date], "dddd"),
        "週末フラグ", IF(WEEKDAY([Date]) > 5, 1, 0)
    )
```

3. 勘定科目階層と集計レベル

```
// 勘定科目階層の設定（DAX）
勘定科目パス =
VAR 現在の科目 = 勘定科目[AccountID]
RETURN
    PATH(
        現在の科目,
        勘定科目[AccountID],
        勘定科目[ParentAccountID]
    )

// 勘定科目レベルの設定
勘定科目レベル =
PATHLENGTH(
    [勘定科目パス]
)
```

```
// 特定レベルの勘定科目名を取得
レベル1勘定科目 =
VAR パス = [勘定科目パス]
RETURN
    LOOKUPVALUE(
        勘定科目[AccountName],
        勘定科目[AccountID],
        PATHITEM(パス, 1)
    )
```

主要財務指標の計算

DAXを使用した主要な財務指標の計算：

1. 期間比較と成長率

```
// 当期売上
当期売上 =
CALCULATE(
    SUM(仕訳[Amount]),
    仕訳[AccountType] = "収益"
)

// 前年同期売上
前年同期売上 =
CALCULATE(
    [当期売上],
    SAMEPERIODLASTYEAR('財務カレンダー'[Date])
)

// 売上成長率
売上成長率 =
DIVIDE(
    [当期売上] - [前年同期売上],
    [前年同期売上],
    0
)

// 前四半期比
前四半期比 =
DIVIDE(
    [当期売上],
    CALCULATE(
        [当期売上],
        DATEADD('財務カレンダー'[Date], -1, QUARTER)
    ),
    0
)
```

2. 累計と移動平均

```
// 年度累計売上
年度累計売上 =
CALCULATE(
    SUM(仕訳[Amount]),
    仕訳[AccountType] = "収益",
    DATESYTD(
        '財務カレンダー'[Date],
        '財務カレンダー'[会計年度]
    )
)

// 3ヶ月移動平均売上
三ヶ月移動平均売上 =
AVERAGEX(
    DATESINPERIOD(
        '財務カレンダー'[Date],
        MAX('財務カレンダー'[Date]),
        -3,
        MONTH
    ),
    [月次売上]
)
```

> ❓ **若手の疑問解決**
> **Q**: 会計年度と暦年の違いをどう処理すればよいですか？
> **A**: 会計年度（日本企業では多くが4月始まり）を扱うには、カスタム日付テーブルを作成するのがベストです。日付テーブルに「会計年度」「会計四半期」「会計月」などの列を追加し、それらを使って計算します。例えば、4月始まりの会計年度なら、1-3月は前年度、4-12月は当年度として計算します。DAXの標準関数（DATESYTD）も使えますが、第2引数に「会計年度末」を指定する必要があります。

3. 予算と実績の比較

```
// 予算達成率
予算達成率 =
DIVIDE(
    [当期売上],
    SUM(予算[Amount]),
    0
)

// 予算差異
予算差異 =
[当期売上] - SUM(予算[Amount])

// 予算差異率
予算差異率 =
```

```
DIVIDE(
    [予算差異],
    SUM(予算[Amount]),
    0
)

// 予算進捗率
予算進捗率 =
VAR 経過日数 = COUNTROWS(FILTER('財務カレンダー', '財務カレンダー'[Date] <= MAX('財務カレンダー'[Date])))
VAR 期間合計日数 = COUNTROWS('財務カレンダー')
RETURN
    DIVIDE(
        [予算達成率],
        DIVIDE(経過日数, 期間合計日数, 0),
        0
    )
```

4. 財務比率の計算

```
// 粗利益率
粗利益率 =
DIVIDE(
    [粗利益],
    [売上],
    0
)

// 営業利益率
営業利益率 =
DIVIDE(
    [営業利益],
    [売上],
    0
)

// 経費率
経費率 =
DIVIDE(
    CALCULATE(
        SUM(仕訳[Amount]),
        仕訳[AccountType] = "費用",
        NOT(CONTAINSSTRING(仕訳[AccountPath], "売上原価"))
    ),
    [売上],
    0
)

// 資産回転率
```

```
資産回転率 =
DIVIDE(
    [年間売上],
    AVERAGE(資産[金額]),
    0
)
```

財務分析ダッシュボードの構築

効果的な財務分析ダッシュボードの設計：

1. 経営者向けサマリービュー

```
// 主要KPIの計算
売上目標達成率 =
DIVIDE(
    [当期売上],
    [売上目標],
    0
)

営業利益率 =
DIVIDE(
    [営業利益],
    [当期売上],
    0
)

前年同期比 =
DIVIDE(
    [当期売上],
    [前年同期売上],
    0
)

キャッシュコンバージョンサイクル =
[売上債権回転日数] + [在庫回転日数] - [買入債務回転日数]
```

経営サマリーダッシュボードの主要構成要素：

- 主要財務KPIカード（売上、利益、前年比など）
- 時系列トレンドチャート（月次・四半期推移）
- 予算vs実績の比較ビジュアル
- セグメント別業績分布（部門、製品など）
- 異常値や注目ポイントのアラート

2. 財務部門向け詳細分析ビュー

```
// 各種財務指標の計算
売上総利益率推移 =
ADDCOLUMNS(
    VALUES('財務カレンダー'[年月]),
    "粗利益率",
    CALCULATE(
        DIVIDE(
            SUM(仕訳[Amount]),
            CALCULATE(
                SUM(仕訳[Amount]),
                仕訳[AccountType] = "収益"
            ),
            0
        ),
        仕訳[AccountType] = "粗利益"
    )
)

部門別コスト構造 =
SUMMARIZECOLUMNS(
    部門[部門名],
    "人件費", CALCULATE(SUM(仕訳[Amount]), 仕訳[AccountName] = "人件費"),
    "材料費", CALCULATE(SUM(仕訳[Amount]), 仕訳[AccountName] = "材料費"),
    "設備費", CALCULATE(SUM(仕訳[Amount]), 仕訳[AccountName] = "設備費"),
    "その他", CALCULATE(SUM(仕訳[Amount]), 仕訳[AccountName] = "その他費用")
)
```

財務部門向け詳細ビューの主要構成要素：

- 勘定科目ドリルダウン分析
- キャッシュフロー詳細
- 部門別コスト構造分析
- 財務比率の詳細分析
- 異常トランザクションの検出
- シナリオ分析と予測

3. レポーティングパッケージ

```
// 期間比較財務諸表
損益計算書比較 =
SUMMARIZECOLUMNS(
    勘定科目[AccountName],
    勘定科目[AccountType],
    勘定科目[勘定科目レベル],
    "当期", CALCULATE(SUM(仕訳[Amount])),
    "前期", CALCULATE(SUM(仕訳[Amount]), DATEADD('財務カレンダー'[Date], -1, YEAR)),
    "差異", [当期] - [前期],
```

```
    "差異率", DIVIDE([差異], [前期], 0)
)
```

財務レポーティングパッケージの主要構成要素：

- 標準財務諸表（損益計算書、貸借対照表、キャッシュフロー計算書）
- 予算vs実績レポート
- 部門別業績レポート
- 詳細取引一覧
- 注釈付きハイライト

> 🚫 **失敗から学ぶ**
> ある金融機関の財務分析プロジェクトで、当初はすべての計算を計算列として実装していました。数十の財務指標を各行レベルで計算したため、データモデルのサイズが10倍以上に膨れ上がり、更新に数時間かかるようになりました。問題を解決するために、すべての集計計算をメジャーに変換し、必要なディメンション分析のみを計算列として残しました。この変更により、データモデルのサイズが大幅に削減され、更新時間も10分以内に短縮されました。財務分析では特に、「行レベルの属性」と「集計計算」を明確に区別し、適切な実装方法を選択することが重要です。

財務分析の実践と活用

財務データ分析の実践的な活用例：

1. **財務予測と計画**:
   - 傾向分析に基づく予測
   - What-if分析とシナリオモデリング
   - 予算計画への反映
2. **異常値検出と監査**:
   - 統計的手法による異常取引の検出
   - パターン分析による不正兆候の発見
   - 監査証跡の追跡
3. **投資評価と意思決定**:
   - 投資収益率（ROI）分析
   - 資本収益率（ROIC）計算
   - 設備投資・プロジェクト評価
4. **経営戦略への活用**:
   - 製品/サービス収益性分析
   - 顧客セグメント別収益性
   - 市場投資効果の測定

10.2 販売データのダッシュボード作成

販売データの分析は、多くの企業にとって重要な業務の一つです。ここでは、Power QueryとDAXを活用して、効果的な販売データダッシュボードを構築する方法を見ていきます。

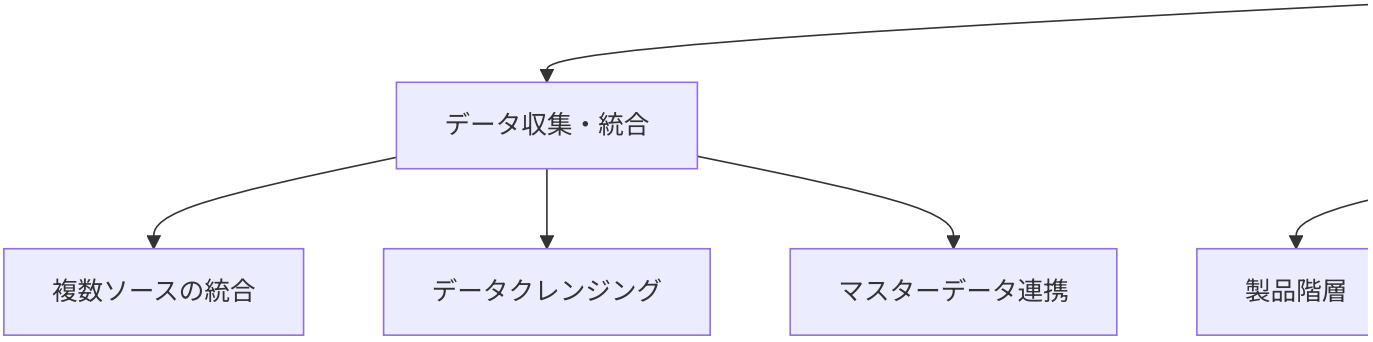
販売データ分析の課題とアプローチ

販売データ分析における一般的な課題：

1. **多様なデータソース**:

- POSシステム、ECサイト、営業報告など
- データ形式や粒度の違い
- リアルタイム性の要求

2. **多角的な分析ニーズ**:
   - 製品、顧客、地域、時間などの多次元分析
   - 階層型データ（製品カテゴリ、地域区分など）
   - 複雑なKPI計算
3. **使いやすいインターフェース**:
   - 直感的なナビゲーション
   - インタラクティブなフィルタリング
   - 適切な視覚化の選択



販売データの準備と変換

Power Queryを使った販売データの準備：

1. 複数ソースの販売データ統合

```
// 複数販売チャネルのデータ統合（M言語）
let
    // 店舗POS販売データ
    店舗販売 = Sql.Database("RetailServer", "POSDatabase"),
    POSデータ = 店舗販売{[Schema="dbo",Item="Transactions"]}[Data],

    // ECサイト販売データ
    EC販売 = Json.Document(Web.Contents("https://api.ecommerce.com/sales/daily")),
    ECテーブル = Table.FromRecords(EC販売[sales]),

    // CSVエクスポートされた営業報告
    営業報告 = Csv.Document(File.Contents("\\SharePoint\Sales\FieldSales.csv")),
    営業テーブル = Table.PromoteHeaders(営業報告),

    // 各データソースの列名と型を標準化
    POS標準化 = Table.TransformColumnTypes(
        Table.RenameColumns(
```

```
        POSデータ,
        {
            {"transaction_date", "SalesDate"},
            {"store_id", "ChannelID"},
            {"product_id", "ProductID"},
            {"quantity", "Quantity"},
            {"unit_price", "UnitPrice"},
            {"sale_amount", "SalesAmount"}
        }
    ),
    {
        {"SalesDate", type date},
        {"ChannelID", type text},
        {"ProductID", type text},
        {"Quantity", Int64.Type},
        {"UnitPrice", type number},
        {"SalesAmount", type number}
    }
),

// ECデータの標準化も同様に実施
EC標準化 = Table.TransformColumnTypes(
    Table.RenameColumns(
        ECテーブル,
        {
            {"date", "SalesDate"},
            {"site_id", "ChannelID"},
            {"sku", "ProductID"},
            {"qty", "Quantity"},
            {"price", "UnitPrice"},
            {"total", "SalesAmount"}
        }
    ),
    {
        {"SalesDate", type date},
        {"ChannelID", type text},
        {"ProductID", type text},
        {"Quantity", Int64.Type},
        {"UnitPrice", type number},
        {"SalesAmount", type number}
    }
),

// 営業データの標準化も同様に実施
営業標準化 = ...,

// チャネル識別用の列を追加
POS識別 = Table.AddColumn(POS標準化, "ChannelType", each "実店舗", type text),
EC識別 = Table.AddColumn(EC標準化, "ChannelType", each "ECサイト", type text),
営業識別 = Table.AddColumn(営業標準化, "ChannelType", each "直販営業", type text),
```

```
    // 全チャネルデータの結合
    全販売データ = Table.Combine({POS識別, EC識別, 営業識別})
in
    全販売データ
```

> 💡 **ベテランの知恵袋**
> 販売データを統合する際の最大の課題は「キーの一致」です。製品IDや顧客IDが各システムで異なる形式や命名規則を持っていることがよくあります。例えば、ある販売チャネルでは製品コードがSKU-12345の形式で、別のチャネルではただの数字12345として保存されていることがあります。統合の前に、すべてのIDを標準形式に変換するためのマッピングテーブルやロジックを用意しておくと、後々の分析が格段にスムーズになります。また、重複データの検出と処理も重要なポイントです。

2. 製品マスターと階層の準備

```
// 製品階層の作成（M言語）
let
    // 製品マスターデータ
    ソース = Excel.Workbook(File.Contents("\\SharePoint\Master\Products.xlsx"), null,
true),
    製品テーブル = ソース{[Name="ProductMaster"]}[Data],
    型変換 = Table.TransformColumnTypes(
        製品テーブル,
        {
            {"ProductID", type text},
            {"ProductName", type text},
            {"CategoryID", type text},
            {"SubCategoryID", type text},
            {"UnitCost", Currency.Type},
            {"ListPrice", Currency.Type}
        }
    ),

    // カテゴリマスター
    カテゴリソース = Excel.Workbook(File.Contents("\\SharePoint\Master\Categories.xlsx"),
null, true),
    カテゴリテーブル = カテゴリソース{[Name="CategoryMaster"]}[Data],
    カテゴリ型変換 = Table.TransformColumnTypes(
        カテゴリテーブル,
        {
            {"CategoryID", type text},
            {"CategoryName", type text}
        }
    ),

    // サブカテゴリマスター
    サブカテゴリソース = ...,
```

```
    // 製品データにカテゴリ情報を結合
    カテゴリ結合 = Table.NestedJoin(
        型変換,
        {"CategoryID"},
        カテゴリ型変換,
        {"CategoryID"},
        "Category",
        JoinKind.LeftOuter
    ),
    カテゴリ展開 = Table.ExpandTableColumn(
        カテゴリ結合,
        "Category",
        {"CategoryName"},
        {"CategoryName"}
    ),

    // サブカテゴリ情報も同様に結合
    サブカテゴリ結合 = ...,

    // 製品分類パスの作成
    パス追加 = Table.AddColumn(
        最終結合,
        "ProductPath",
        each [CategoryName] & " > " & [SubCategoryName] & " > " & [ProductName],
        type text
    ),

    // 利益率の計算
    利益率追加 = Table.AddColumn(
        パス追加,
        "GrossMargin",
        each ([ListPrice] - [UnitCost]) / [ListPrice],
        type number
    )
in
    利益率追加
```

3. 販売データのクレンジングと強化

```
// 販売データのクレンジングと強化（M言語）
let
    ソース = 全販売データ,

    // 返品データの特定（金額がマイナスの取引）
    返品フラグ追加 = Table.AddColumn(
        ソース,
        "IsReturn",
        each [SalesAmount] < 0,
        type logical
```

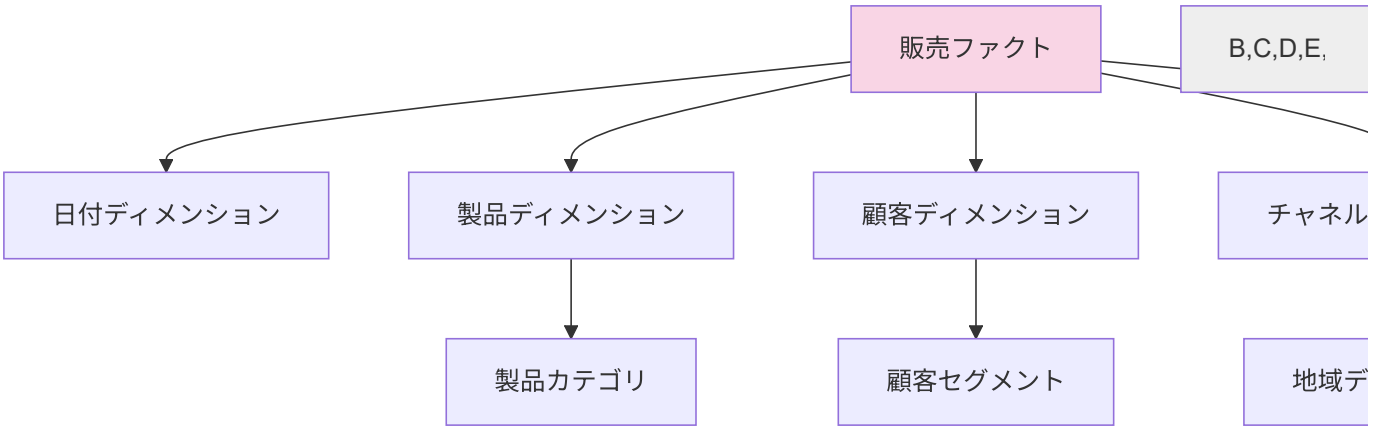
```
    ),

    // 販売日から各種時間属性を生成
    時間属性追加 = Table.AddColumn(
        返品フラグ追加,
        "YearMonth",
        each Date.ToText([SalesDate], "yyyy/MM"),
        type text
    ),
    年追加 = Table.AddColumn(
        時間属性追加,
        "Year",
        each Date.Year([SalesDate]),
        Int64.Type
    ),
    四半期追加 = Table.AddColumn(
        年追加,
        "Quarter",
        each "Q" & Text.From(Date.QuarterOfYear([SalesDate])),
        type text
    ),
    月追加 = Table.AddColumn(
        四半期追加,
        "Month",
        each Date.Month([SalesDate]),
        Int64.Type
    ),

    // 製品マスターとの結合で製品情報を追加
    製品結合 = Table.NestedJoin(
        月追加,
        {"ProductID"},
        製品マスター,
        {"ProductID"},
        "ProductInfo",
        JoinKind.LeftOuter
    ),
    製品展開 = Table.ExpandTableColumn(
        製品結合,
        "ProductInfo",
        {"ProductName", "CategoryName", "SubCategoryName", "UnitCost", "GrossMargin"},
        {"ProductName", "CategoryName", "SubCategoryName", "UnitCost", "GrossMargin"}
    ),

    // 粗利益の計算
    粗利益追加 = Table.AddColumn(
        製品展開,
        "GrossProfit",
        each [SalesAmount] - ([UnitCost] * [Quantity]),
        type number
```

```
    )
in
    粗利益追加
```

販売データモデルの構築

効果的な販売分析のためのデータモデル設計：

1. 販売分析用スタースキーマ



2. 販売データの粒度と集計の設計

```
// ファクトテーブルの粒度設計（DAX）
販売ファクト =
    SELECTCOLUMNS(
        販売データ,
        "販売ID", [SalesID],
        "日付", [SalesDate],
        "製品ID", [ProductID],
        "顧客ID", [CustomerID],
        "チャネルID", [ChannelID],
        "販売員ID", [SalesPersonID],
        "数量", [Quantity],
        "売上金額", [SalesAmount],
        "原価", [UnitCost] * [Quantity],
        "粗利益", [SalesAmount] - ([UnitCost] * [Quantity])
    )
```

3. 時間インテリジェンスの設定

```
// 販売分析用の日付テーブル（DAX）
日付テーブル =
VAR 開始日 = DATE(2019, 1, 1)
VAR 終了日 = DATE(2025, 12, 31)
RETURN
```

```
    ADDCOLUMNS(
        CALENDAR(開始日, 終了日),
        "年", YEAR([Date]),
        "四半期", "Q" & FORMAT([Date], "Q"),
        "年四半期", FORMAT([Date], "yyyy") & "/Q" & FORMAT([Date], "Q"),
        "月", MONTH([Date]),
        "月名", FORMAT([Date], "MMM"),
        "年月", FORMAT([Date], "yyyy/MM"),
        "週番号", WEEKNUM([Date]),
        "曜日", WEEKDAY([Date]),
        "曜日名", FORMAT([Date], "dddd"),
        "週末フラグ", IF(WEEKDAY([Date]) > 5, 1, 0),
        "月初", IF(DAY([Date]) = 1, 1, 0),
        "月末", IF(DAY([Date]) = EOMONTH([Date], 0), 1, 0)
    )
```

> ❓ **若手の疑問解決**
> **Q**: 販売データモデルでディメンションとファクトの関係をどう設計すべきですか？
> **A**: 基本的にはスタースキーマが最適です。販売ファクトを中心に据え、製品、顧客、日付、チャネルなどをディメンションテーブルとして配置します。各ディメンションからファクトへは1対多の関係を設定します。複雑な階層（製品カテゴリなど）がある場合、そのディメンション内で親子関係を表現するか、場合によっては正規化された複数テーブル（スノーフレークスキーマ）にすることもあります。重要なのは、分析の目的に合わせて適切な粒度とリレーションシップを設計することです。例えば、顧客と製品の両方から見た分析が必要なら、その両方のディメンションが適切に接続されている必要があります。

販売分析の主要KPIと計算

DAXを使用した販売分析のためのKPI計算：

1. 基本的な販売指標

```
// 売上合計
売上合計 =
SUM(販売ファクト[売上金額])

// 販売数量
販売数量 =
SUM(販売ファクト[数量])

// 平均単価
平均単価 =
DIVIDE(
    [売上合計],
    [販売数量],
    0
)

// 粗利益
粗利益 =
```

```
SUM(販売ファクト[粗利益])

// 粗利益率
粗利益率 =
DIVIDE(
    [粗利益],
    [売上合計],
    0
)
```

2. 前年比と成長率

```
// 前年同期売上
前年同期売上 =
CALCULATE(
    [売上合計],
    SAMEPERIODLASTYEAR('日付'[Date])
)

// 売上成長率
売上成長率 =
DIVIDE(
    [売上合計] - [前年同期売上],
    [前年同期売上],
    0
)

// 累積成長率
累積成長率 =
VAR 当期累計 = CALCULATE(
    [売上合計],
    DATESYTD('日付'[Date])
)
VAR 前年同期累計 = CALCULATE(
    [売上合計],
    DATESYTD(
        SAMEPERIODLASTYEAR('日付'[Date])
    )
)
RETURN
    DIVIDE(
        当期累計 - 前年同期累計,
        前年同期累計,
        0
    )
```

3. 製品と顧客の分析指標

```
// 製品別売上寄与率
製品売上寄与率 =
DIVIDE(
    [売上合計],
    CALCULATE(
        [売上合計],
        ALL(製品)
    ),
    0
)

// 製品カテゴリ内シェア
カテゴリ内シェア =
DIVIDE(
    [売上合計],
    CALCULATE(
        [売上合計],
        ALLEXCEPT(
            製品,
            製品[CategoryName]
        )
    ),
    0
)

// 顧客平均購入額
顧客平均購入額 =
DIVIDE(
    [売上合計],
    DISTINCTCOUNT(販売ファクト[顧客ID]),
    0
)

// 顧客リピート率
顧客リピート率 =
DIVIDE(
    CALCULATE(
        DISTINCTCOUNT(販売ファクト[顧客ID]),
        FILTER(
            顧客購入回数,
            [購入回数] > 1
        )
    ),
    DISTINCTCOUNT(販売ファクト[顧客ID]),
    0
)
```

4. 販売チャネル分析

```
// チャネル別売上比率
チャネル別売上比率 =
DIVIDE(
    [売上合計],
    CALCULATE(
        [売上合計],
        ALL(チャネル)
    ),
    0
)

// チャネル別平均単価
チャネル別平均単価 =
DIVIDE(
    [売上合計],
    [販売数量],
    0
)

// チャネル別顧客数
チャネル別顧客数 =
DISTINCTCOUNT(販売ファクト[顧客ID])

// クロスチャネル顧客比率
クロスチャネル顧客比率 =
DIVIDE(
    CALCULATE(
        DISTINCTCOUNT(販売ファクト[顧客ID]),
        FILTER(
            顧客チャネル数,
            [利用チャネル数] > 1
        )
    ),
    DISTINCTCOUNT(販売ファクト[顧客ID]),
    0
)
```

販売ダッシュボードの設計

効果的な販売ダッシュボードの設計と構築：

1. エグゼクティブサマリー

エグゼクティブサマリーダッシュボードの主要構成要素：

- 売上KPIと前年比（カードビジュアル）
- 月次売上トレンドチャート（成長率表示）
- 製品カテゴリ売上分布（ドーナツチャート）
- 地域別売上ヒートマップ

- チャネル別売上比較（棒グラフ）
- 上位顧客/製品リスト

```
// エグゼクティブダッシュボード用の主要KPI
// 全体KPI
全体売上 = [売上合計]
全体成長率 = [売上成長率]
全体粗利益率 = [粗利益率]
前月比 = DIVIDE([当月売上], [前月売上], 0)

// トレンド分析用の月次データ
月次売上トレンド =
SUMMARIZECOLUMNS(
    '日付'[年月],
    "売上", [売上合計],
    "前年同月", [前年同期売上],
    "成長率", [売上成長率]
)
```

2. 製品分析ビュー

製品分析ダッシュボードの主要構成要素：

- 製品カテゴリ階層によるドリルダウン
- 製品別売上・粗利益のパレート図
- 製品成長マトリックス（売上規模×成長率）
- 時系列による製品売上推移
- 販売チャネル別の製品パフォーマンス

```
// 製品分析用の計算
// 製品成長マトリックス用
製品成長区分 =
VAR 成長率 = [売上成長率]
RETURN
    SWITCH(
        TRUE(),
        成長率 >= 0.2, "高成長",
        成長率 >= 0, "成長",
        成長率 >= -0.1, "横ばい",
        "縮小"
    )

製品売上規模 =
VAR 売上 = [売上合計]
RETURN
    SWITCH(
        TRUE(),
        売上 >= 10000000, "大型製品",
        売上 >= 1000000, "中型製品",
```

```
        "小型製品"
    )
```

3. 顧客行動分析

顧客分析ダッシュボードの主要構成要素：

- 顧客セグメント別売上分布
- 購入頻度と客単価のクロス分析
- 新規/リピート顧客の推移
- 顧客生涯価値（LTV）指標
- 顧客流出（チャーン）分析

```
// 顧客分析用の計算
// RFM分析用
最終購入日 =
MAX(販売ファクト[SalesDate])

購入頻度 =
COUNT(販売ファクト[SalesID])

購入金額合計 =
SUM(販売ファクト[売上金額])

// RFMスコア
Rスコア =
VAR 最終日 = MAX('日付'[Date])
VAR 経過日数 = DATEDIFF([最終購入日], 最終日, DAY)
RETURN
    SWITCH(
        TRUE(),
        経過日数 <= 30, 5,
        経過日数 <= 90, 4,
        経過日数 <= 180, 3,
        経過日数 <= 365, 2,
        1
    )

Fスコア =
SWITCH(
    TRUE(),
    [購入頻度] >= 10, 5,
    [購入頻度] >= 5, 4,
    [購入頻度] >= 3, 3,
    [購入頻度] = 2, 2,
    1
)

Mスコア =
```

```
SWITCH(
    TRUE(),
    [購入金額合計] >= 1000000, 5,
    [購入金額合計] >= 500000, 4,
    [購入金額合計] >= 100000, 3,
    [購入金額合計] >= 50000, 2,
    1
)

RFM総合スコア = [Rスコア] + [Fスコア] + [Mスコア]

顧客セグメント =
SWITCH(
    TRUE(),
    [RFM総合スコア] >= 13, "VIP",
    [RFM総合スコア] >= 10, "優良顧客",
    [RFM総合スコア] >= 7, "一般顧客",
    [RFM総合スコア] >= 4, "要フォロー",
    "流出リスク"
)
```

> 💼 **プロジェクト事例**
> ある小売業では、複数の店舗とオンラインショップの販売データが別々のシステムで管理されていました。Power BIを使って統合ダッシュボードを構築したところ、オンラインと実店舗での顧客の購買行動に大きな違いがあることが明らかになりました。特に、同じ顧客でもチャネルによって購入する商品カテゴリが異なる傾向があり、このインサイトを元にクロスセルキャンペーンを実施したところ、顧客単価が15%向上するという成果が得られました。また、ダッシュボードによって地域ごとの在庫と売上のバランスが可視化され、在庫の最適化にも役立ちました。データ統合による全体像の把握が、具体的なビジネスアクションにつながった好例です。

4. 地域・チャネル分析

地域・チャネル分析ダッシュボードの主要構成要素：

- 地域別売上マップビジュアル
- 地域別売上・成長率のマトリックス
- チャネル別売上と粗利益の比較
- チャネル間のカニバリゼーション分析
- 地域×チャネルのクロス分析ヒートマップ

```
// 地域・チャネル分析用の計算
// 地域別市場浸透率
地域別市場浸透率 =
DIVIDE(
    [売上合計],
    LOOKUPVALUE(
        地域マスター[市場規模],
        地域マスター[地域ID],
        MIN(地域[地域ID])
```

```
    ),
    0
)

// チャネルシフト分析
チャネルシフト率 =
DIVIDE(
    [当期チャネル売上] - [前期チャネル売上],
    [前期チャネル売上],
    0
)

// クロスチャネル効果
クロスチャネル購買率 =
DIVIDE(
    CALCULATE(
        DISTINCTCOUNT(販売ファクト[顧客ID]),
        FILTER(
            ALL(チャネル),
            チャネル[チャネル名] <> SELECTEDVALUE(チャネル[チャネル名])
        )
    ),
    DISTINCTCOUNT(販売ファクト[顧客ID]),
    0
)
```

販売予測と高度な分析

Power BIでの販売予測と高度な分析手法：

1. 時系列予測の実装

```
// 時系列予測用のメジャー
予測売上 =
VAR 履歴データ =
    SUMMARIZECOLUMNS(
        '日付'[年月],
        "売上", [売上合計]
    )
VAR 履歴系列 =
    SELECTCOLUMNS(
        履歴データ,
        "Time", '日付'[年月],
        "Value", [売上合計]
    )
RETURN
    FORECAST.ETS(
        TODAY() + 90,  // 予測終了日
        履歴系列,
```

```
        60,             // 季節性の周期
        1,              // 信頼区間（0～1）
        TRUE            // 季節性あり
    )
```

2. 購買パターンの分析

```
// 併売分析（一緒に購入される製品）
製品併売率 =
VAR 現在製品 = SELECTEDVALUE(製品[製品ID])
VAR 現在製品注文 =
    CALCULATETABLE(
        VALUES(販売ファクト[SalesID]),
        製品[製品ID] = 現在製品
    )
VAR 併売製品販売数 =
    CALCULATE(
        COUNT(販売ファクト[SalesID]),
        製品[製品ID] <> 現在製品,
        販売ファクト[SalesID] IN 現在製品注文
    )
VAR 総販売数 =
    CALCULATE(
        COUNT(販売ファクト[SalesID]),
        製品[製品ID] = 現在製品
    )
RETURN
    DIVIDE(
        併売製品販売数,
        総販売数,
        0
    )
```

3. 異常検出と分析

```
// 平均と標準偏差を使った異常検出
売上異常スコア =
VAR 現在値 = [売上合計]
VAR 履歴データ =
    CALCULATETABLE(
        SUMMARIZECOLUMNS(
            '日付'[年月],
            "売上", [売上合計]
        ),
        DATESBETWEEN(
            '日付'[Date],
            DATEADD(MAX('日付'[Date]), -365, DAY),
            DATEADD(MAX('日付'[Date]), -1, DAY)
```

```
        )
    )
VAR 平均値 = AVERAGEX(履歴データ, [売上])
VAR 標準偏差 = STDEVX.P(履歴データ, [売上])
RETURN
    IF(
        標準偏差 = 0,
        0,
        (現在値 - 平均値) / 標準偏差
    )
```

✅ 第10章 チェックリスト（販売データ分析）

- [ ] 多様な販売データソースの統合方法を学んだ
- [ ] 販売分析に適したデータモデルの設計方法を理解した
- [ ] 主要な販売KPIとその計算方法を習得した
- [ ] 効果的な販売ダッシュボードの設計原則を把握した
- [ ] 高度な分析手法の適用方法を理解した

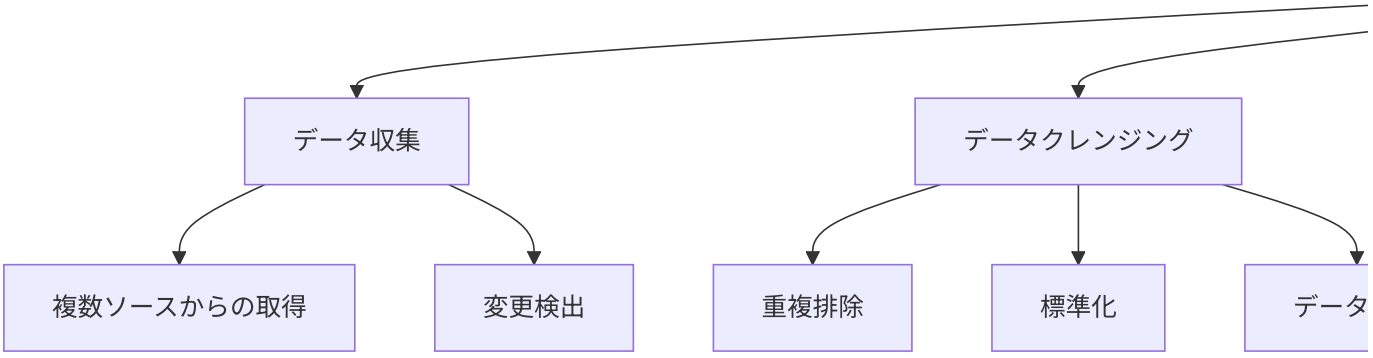
10.3 マスターデータ管理

マスターデータ（製品、顧客、従業員などの基準となるデータ）は、あらゆるビジネスインテリジェンス活動の基盤となります。Power QueryとPower BIを使ったマスターデータ管理（MDM）のアプローチを見ていきましょう。

マスターデータ管理の課題と戦略

マスターデータ管理における一般的な課題：

1. **データの一貫性と正確性**:
   - 複数システム間でのデータ不一致
   - 重複レコードと識別の問題
   - 更新のタイミングと変更管理
2. **データガバナンスとセキュリティ**:
   - データ所有権と責任の明確化
   - 変更履歴と監査証跡の管理
   - アクセス制御と機密データの保護
3. **統合と配布**:
   - 様々なソースからのデータ収集
   - 標準形式への変換と調和
   - 各システムへの配布と同期



マスターデータの収集と統合

Power Queryを使ったマスターデータの収集と統合：

1. 複数ソースからのマスターデータ収集

```
// 複数システムからの製品マスターデータ収集（M言語）
let
    // ERPからの製品データ
    ERP製品 = Sql.Database("ERPServer", "ProductionDB"),
    ERP製品テーブル = ERP製品{[Schema="dbo",Item="Products"]}[Data],

    // CRMからの製品データ
    CRM製品 = Sql.Database("CRMServer", "SalesDB"),
    CRM製品テーブル = CRM製品{[Schema="dbo",Item="ProductCatalog"]}[Data],

    // Eコマースシステムからの製品データ
    EC製品 = Json.Document(Web.Contents("https://api.ecommerce.com/products")),
    EC製品テーブル = Table.FromRecords(EC製品[products]),

    // 各ソースのデータ構造と列名の標準化
    ERP標準化 = Table.TransformColumnTypes(
        Table.RenameColumns(
            ERP製品テーブル,
            {
                {"ProductCode", "ProductID"},
                {"ProductDescription", "ProductName"},
                {"ProductGroup", "CategoryID"},
                {"UnitCost", "Cost"},
                {"SalesPrice", "Price"}
            }
        ),
        {
            {"ProductID", type text},
            {"ProductName", type text},
```

```
            {"CategoryID", type text},
            {"Cost", type number},
            {"Price", type number}
        }
    ),

    // CRMデータの標準化
    CRM標準化 = ...,

    // ECデータの標準化
    EC標準化 = ...,

    // ソース識別子の追加
    ERP識別 = Table.AddColumn(ERP標準化, "DataSource", each "ERP", type text),
    CRM識別 = Table.AddColumn(CRM標準化, "DataSource", each "CRM", type text),
    EC識別 = Table.AddColumn(EC標準化, "DataSource", each "Eコマース", type text),

    // ソースデータの結合（まだ重複除去・統合は行わない）
    全ソース = Table.Combine({ERP識別, CRM識別, EC識別})
in
    全ソース
```

> 🚫 **失敗から学ぶ**
> あるプロジェクトでは、製品マスターデータの統合に取り組みましたが、各システムの製品IDの形式が異なることに気づきませんでした。ERPでは純粋な数字（例：12345）、CRMではプレフィックス付き（例：PROD-12345）、Eコマースではさらに別形式（例：P12345）でした。単純に統合しようとしたため大量の重複が発生し、レポートの数値が大幅に誤っていました。この問題を解決するために、各システムのID形式を分析し、正規化ルールを作成（プレフィックスの削除など）することで、一貫した識別子を作ることができました。マスターデータ統合では、識別子（キー）の標準化が最も重要なステップの一つです。

2. マスターデータのクレンジングと標準化

```
// 製品マスターデータのクレンジングと標準化（M言語）
let
    ソース = 全ソース,

    // データクレンジング：空白、特殊文字などの処理
    空白トリム = Table.TransformColumns(
        ソース,
        {{"ProductName", Text.Trim, type text}}
    ),
    特殊文字処理 = Table.TransformColumns(
        空白トリム,
        {{"ProductName", each Text.Clean(_), type text}}
    ),

    // 製品IDの標準化（システム固有プレフィックスの削除など）
    ID標準化 = Table.AddColumn(
```

```
        特殊文字処理,
        "StandardProductID",
        each
            if Text.StartsWith([ProductID], "PROD-") then Text.RemoveRange([ProductID],
0, 5)
            else if Text.StartsWith([ProductID], "P") then
Text.RemoveRange([ProductID], 0, 1)
            else [ProductID],
        type text
    ),

    // カテゴリ名の標準化（同義語の統一など）
    カテゴリマッピング = #table(
        {"OriginalCategory", "StandardCategory"},
        {
            {"HDWR", "Hardware"},
            {"HW", "Hardware"},
            {"ハードウェア", "Hardware"},
            {"SW", "Software"},
            {"ソフトウェア", "Software"},
            {"SFTWR", "Software"}
        }
    ),
    カテゴリ結合 = Table.NestedJoin(
        ID標準化,
        {"CategoryID"},
        カテゴリマッピング,
        {"OriginalCategory"},
        "CategoryMap",
        JoinKind.LeftOuter
    ),
    カテゴリ展開 = Table.ExpandTableColumn(
        カテゴリ結合,
        "CategoryMap",
        {"StandardCategory"},
        {"StandardCategory"}
    ),
    カテゴリ処理 = Table.ReplaceValue(
        カテゴリ展開,
        null,
        each [CategoryID],
        Replacer.ReplaceValue,
        {"StandardCategory"}
    )
in
    カテゴリ処理
```

3. 重複排除とゴールデンレコードの作成

```
// 重複の検出と統合（M言語）
let
    ソース = カテゴリ処理,

    // 標準化されたIDでグループ化して重複を特定
    グループ化 = Table.Group(
        ソース,
        {"StandardProductID"},
        {
            {"重複数", each Table.RowCount(_), Int64.Type},
            {"製品名リスト", each [ProductName], type list},
            {"データソース", each [DataSource], type list},
            {"カテゴリ", each List.Distinct([StandardCategory]), type list},
            {"平均コスト", each List.Average([Cost]), type number},
            {"平均価格", each List.Average([Price]), type number},
            {"最新レコード", each Table.First(Table.Sort(_, {{"LastUpdated",
Order.Descending}})), type table}
        }
    ),

    // ゴールデンレコードの作成ルール適用
    ゴールデンレコード = Table.AddColumn(
        グループ化,
        "ProductName",
        each
            // 優先ソースがある場合そちらを使用、なければ最新レコードの値
            if List.Contains([データソース], "ERP") then
                Table.SelectRows([最新レコード], each [DataSource] = "ERP"){0}
[ProductName]
            else
                [最新レコード]{0}[ProductName],
        type text
    ),

    // 他の属性も同様のルールで統合
    カテゴリ統合 = Table.AddColumn(
        ゴールデンレコード,
        "CategoryID",
        each
            if List.Count([カテゴリ]) = 1 then
                [カテゴリ]{0}
            else if List.Contains([カテゴリ], "Hardware") then
                "Hardware"
            else
                [カテゴリ]{0},
        type text
    ),

    // 価格と原価の統合（ERPを優先、なければ平均）
```

```
    コスト統合 = Table.AddColumn(
        カテゴリ統合,
        "Cost",
        each
            if List.Contains([データソース], "ERP") then
                Table.SelectRows([最新レコード], each [DataSource] = "ERP"){0}[Cost]
            else
                [平均コスト],
        type number
    ),

    // 最終的なマスターレコードの整形
    最終形成 = Table.SelectColumns(
        コスト統合,
        {"StandardProductID", "ProductName", "CategoryID", "Cost", "Price", "重複数",
"データソース"}
    ),
    列名変更 = Table.RenameColumns(
        最終形成,
        {{"StandardProductID", "ProductID"}}
    )
in
    列名変更
```

> 💡 **ベテランの知恵袋**
> マスターデータの統合で最も難しいのは、「最良のレコード」を選ぶルールの設計です。通常、優先する「信頼できるソース」を決め、そのソースからの値を優先します。例えば製品情報はERPを、顧客情報はCRMを信頼するというルールです。しかし、すべての属性に同じルールを適用するわけではありません。例えば、製品名はマーケティングシステムの値が最新かもしれませんし、価格情報はEコマースプラットフォームが最も正確かもしれません。属性ごとに「ゴールデンソース」を定義し、そのルールをクエリに組み込むことが、効果的なマスターデータ管理の鍵となります。

マスターデータの変更管理と履歴追跡

データガバナンスの重要な側面は、「いつ、誰が、何を、なぜ変更したか」を追跡することです：

```
// マスターデータの変更履歴を管理するモデル（M言語）
let
    // 前回のマスターデータスナップショット
    前回スナップショット =
Excel.Workbook(File.Contents("\\Share\MDM\Previous\MasterData.xlsx"), null, true),
    前回製品 = 前回スナップショット{[Item="Products",Kind="Table"]}[Data],

    // 現在のマスターデータ
    現在のマスター = 最終統合製品マスター,

    // 変更検出ロジック
    変更検出 = Table.NestedJoin(
        現在のマスター,
```

```
        {"ProductID"},
        前回製品,
        {"ProductID"},
        "以前のデータ",
        JoinKind.LeftOuter
    ),

    変更展開 = Table.ExpandTableColumn(
        変更検出,
        "以前のデータ",
        {"ProductName", "CategoryID", "Cost", "Price"},
        {"旧_ProductName", "旧_CategoryID", "旧_Cost", "旧_Price"}
    ),

    // 変更の検出（新規、更新、削除）
    変更タイプ追加 = Table.AddColumn(
        変更展開,
        "変更タイプ",
        each if [旧_ProductName] = null then "新規"
             else if ([ProductName] <> [旧_ProductName]) or
                     ([CategoryID] <> [旧_CategoryID]) or
                     ([Cost] <> [旧_Cost]) or
                     ([Price] <> [旧_Price]) then "更新"
             else "変更なし",
        type text
    ),

    // 変更されたフィールドを特定
    変更フィールド追加 = Table.AddColumn(
        変更タイプ追加,
        "変更フィールド",
        each if [変更タイプ] = "新規" then "すべて"
             else if [変更タイプ] = "変更なし" then ""
             else
                Text.Combine(
                    List.Select(
                        {
                            if [ProductName] <> [旧_ProductName] then "製品名" else
null,
                            if [CategoryID] <> [旧_CategoryID] then "カテゴリ" else
null,
                            if [Cost] <> [旧_Cost] then "原価" else null,
                            if [Price] <> [旧_Price] then "価格" else null
                        },
                        each _ <> null
                    ),
                    ", "
                ),
        type text
    ),
```

```
    // 監査情報の追加
    監査情報追加 = Table.AddColumn(
        変更フィールド追加,
        "更新日時",
        each DateTime.LocalNow(),
        type datetime
    ),
    最終形成 = Table.SelectColumns(
        監査情報追加,
        {"ProductID", "ProductName", "CategoryID", "Cost", "Price",
         "変更タイプ", "変更フィールド", "更新日時"}
    )
in
    最終形成
```

マスターデータの配布と同期プロセス

統合したマスターデータを各システムに配布するプロセス：

```
// 他システムへの配布用データ抽出（M言語）
// ERPシステム向けエクスポート
ERP用データ =
let
    ソース = マスターデータ,
    変更フィルタ = Table.SelectRows(ソース, each [変更タイプ] <> "変更なし"),
    形式変換 = Table.TransformColumnTypes(
        変更フィルタ,
        {
            {"ProductID", type text},
            {"ProductName", type text},
            {"CategoryID", type text}
        }
    ),
    // ERP固有の形式に変換
    列名変更 = Table.RenameColumns(
        形式変換,
        {
            {"ProductID", "PRODUCT_CODE"},
            {"ProductName", "DESCRIPTION"},
            {"CategoryID", "CATEGORY_CODE"}
        }
    ),
    最終形成 = Table.SelectColumns(
        列名変更,
        {"PRODUCT_CODE", "DESCRIPTION", "CATEGORY_CODE", "Cost", "Price"}
    )
in
    最終形成
```

マスターデータの品質監視

データ品質を継続的に監視するためのダッシュボード作成：

```
// マスターデータの品質メトリクス計算（DAX）
// 完全性スコア（必須項目の入力率）
データ完全性 =
VAR 全レコード数 = COUNTROWS(マスターデータ)
VAR 必須項目完備数 =
    CALCULATE(
        COUNTROWS(マスターデータ),
        マスターデータ[ProductName] <> BLANK(),
        マスターデータ[CategoryID] <> BLANK(),
        マスターデータ[Cost] <> BLANK(),
        マスターデータ[Price] <> BLANK()
    )
RETURN
    DIVIDE(必須項目完備数, 全レコード数, 0)

// データ一意性（重複の割合）
データ一意性 =
VAR 全レコード数 = COUNTROWS(マスターデータ)
VAR 一意なID数 = DISTINCTCOUNT(マスターデータ[ProductID])
RETURN
    DIVIDE(一意なID数, 全レコード数, 0)

// データの一貫性（カテゴリマスターに存在するカテゴリの割合）
データ一貫性 =
VAR 有効なカテゴリ =
    CALCULATE(
        COUNTROWS(マスターデータ),
        TREATAS(
            VALUES(カテゴリマスター[CategoryID]),
            マスターデータ[CategoryID]
        )
    )
VAR 全レコード数 = COUNTROWS(マスターデータ)
RETURN
    DIVIDE(有効なカテゴリ, 全レコード数, 0)
```

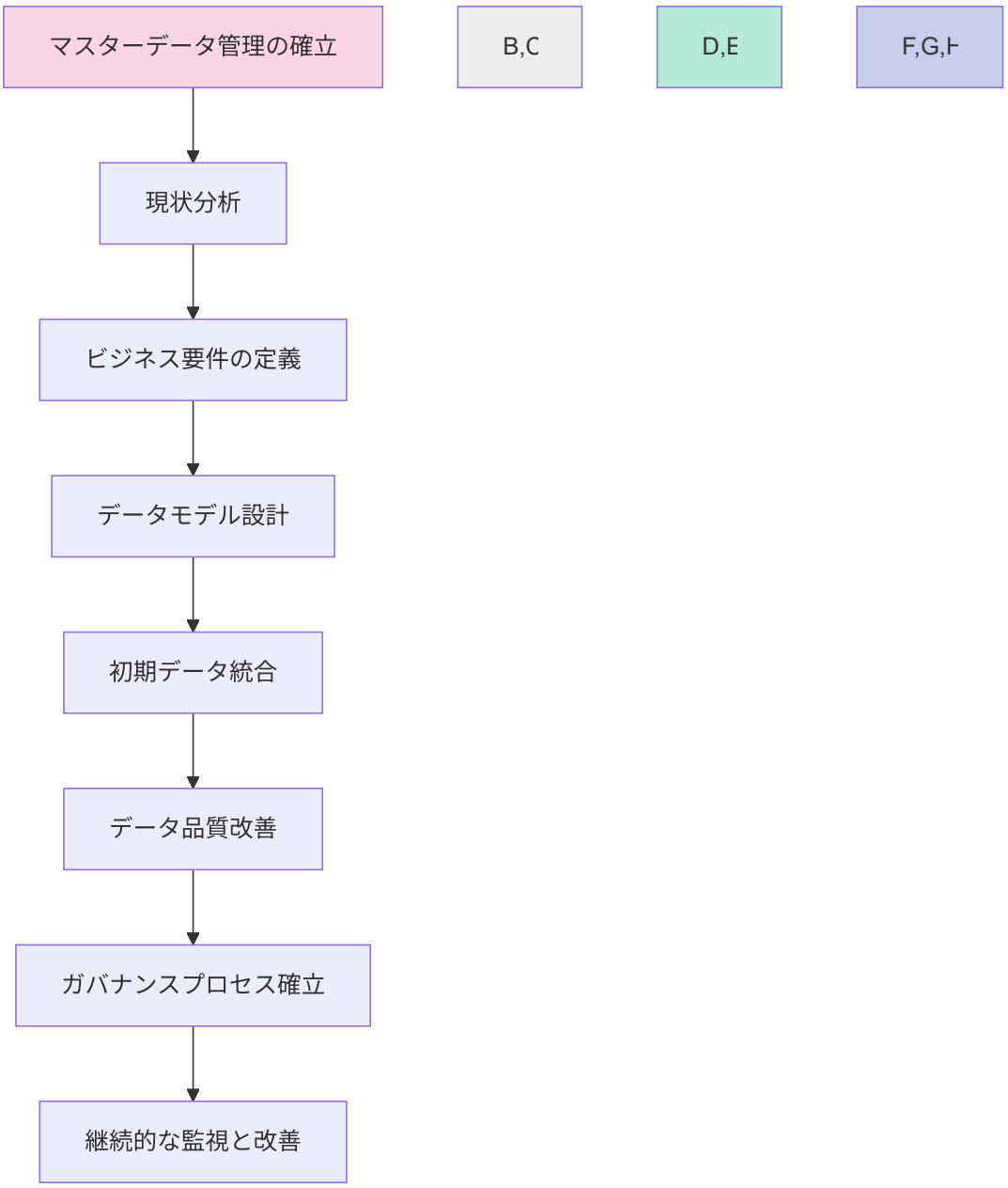
> ❓ **若手の疑問解決**
> **Q**: マスターデータ管理を効率的に始めるにはどうすれば良いでしょうか？
> **A**: まずは小さく始めることが重要です。すべてのマスターデータを一度に整備しようとすると、プロジェクトが大きくなりすぎて頓挫するリスクがあります。最も重要なマスターデータ（多くの場合は顧客または製品）から始め、以下のステップで進めると良いでしょう：
>
> 1. まず現状分析：どのシステムにどんなデータがあるか棚卸し
> 2. データモデルの設計：必要な属性と関係の定義
> 3. データの統合と標準化プロセスの構築（Power Queryが有効）

4. 品質チェックとクレンジングの自動化
5. ガバナンスプロセスの確立（誰が、どのように更新するか）
6. 最後に他のマスターデータ（取引先、従業員など）へ拡大

MDMの実践的アプローチ

効果的なマスターデータ管理(MDM)のための実践的アプローチ：

1. **ビジネス主導のアプローチ**:
   - データオーナーシップの明確化
   - 品質基準の定義
   - 業務プロセスとの統合
2. **段階的な実装**:
   - 最も重要なドメイン（製品、顧客など）から開始
   - 小さな成功を積み重ねる
   - フィードバックループの確立
3. **持続可能なガバナンス**:
   - 明確なデータ管理ポリシー
   - 役割と責任の定義
   - 定期的な監査とレビュー



> 💼 **プロジェクト事例**
> あるメーカーでは、複数のシステムに分散した製品マスターデータの不整合が原因で、誤った価格設定や在庫不足などの問題が頻発していました。Power BIとPower Queryを活用して、週次で各システムからデータを抽出・統合し、不一致を検出するダッシュボードを構築しました。このダッシュボードは各部門のデータ担当者に共有され、不整合を早期に発見・修正できるようになりました。さらに、データ品質の指標（完全性、正確性、一貫性）をKPIとして設定し、毎月のマネジメント会議で報告することで、データ品質に対する全社的な意識が向上しました。実装から6ヶ月で、データ品質スコアは65%から92%に向上し、データ不整合による業務トラブルは80%減少しました。

✓ マスターデータ管理チェックリスト

- [ ] 各システムからのデータ抽出プロセスを自動化した
- [ ] データクレンジングと標準化のルールを定義した
- [ ] 重複識別と統合のロジックを確立した
- [ ] 変更管理と履歴追跡の仕組みを実装した
- [ ] データ品質の監視ダッシュボードを構築した
- [ ] データガバナンスの役割と責任を明確にした
- [ ] 定期的なレビューと改善プロセスを確立した

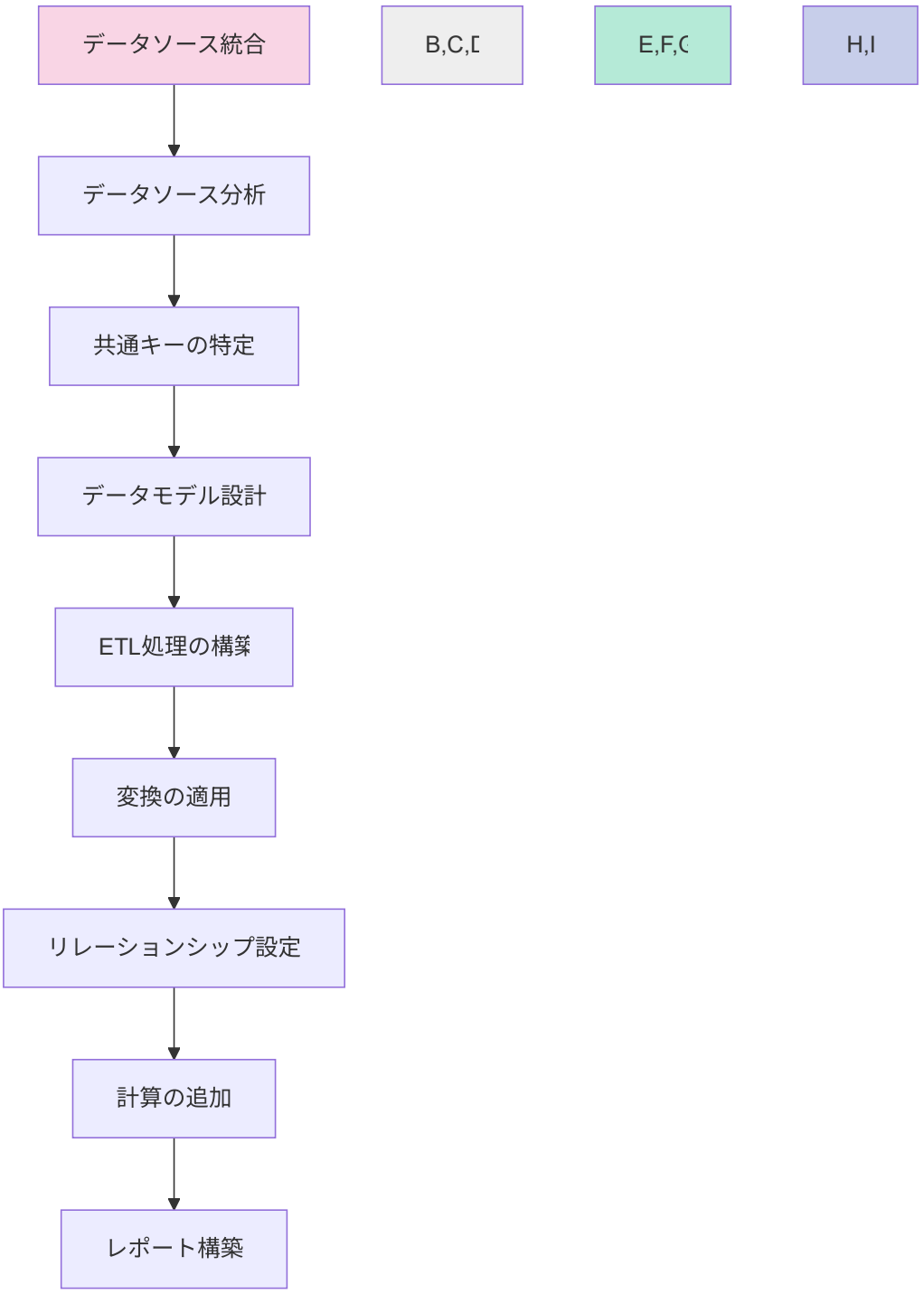
10.4 複数ソースの統合レポート

異なるシステムやデータソースからデータを統合し、包括的なレポートを作成することは、ビジネスインテリジェンスの重要な要素です。ここでは、さまざまなソースからのデータを効果的に統合するテクニックを紹介します。

統合レポートの課題とアプローチ

複数ソースからのデータ統合における主な課題：

1. **データ粒度の違い**:
   - 日次データと月次データの混在
   - 取引レベルと集計レベルの違い
   - 異なる集計単位（個人vs部門など）
2. **キー列と関連付け**:
   - 異なるIDシステムの統合
   - マスターデータの不一致
   - 欠落したリレーションシップ
3. **パフォーマンスの問題**:
   - 大量データの処理
   - クエリの最適化
   - リフレッシュ戦略



異なる粒度のデータ統合

異なる粒度のデータソース（例：日次販売データと月次予算データ）を統合する方法：

```
// 日次販売データを月次に集計（M言語）
let
    ソース = 販売データ,
    月次集計 = Table.Group(
        ソース,
        {"年", "月", "商品カテゴリ", "地域"},
        {
            {"売上金額", each List.Sum([売上金額]), type number},
            {"販売数量", each List.Sum([数量]), type number},
            {"取引数", each Table.RowCount(_), Int64.Type}
        }
    ),
    // 日付フィールド作成（月の初日）
    日付追加 = Table.AddColumn(
        月次集計,
        "日付",
        each #date([年], [月], 1),
        type date
    )
in
    日付追加

// 予算データとの結合（M言語）
let
    月次売上 = #"月次集計済み売上データ",
    予算データ = Excel.Workbook(File.Contents("Budget.xlsx"), null, true),
    予算テーブル = 予算データ{[Name="MonthlyBudget",Kind="Table"]}[Data],

    // 日付形式の標準化
    予算日付変換 = Table.TransformColumns(
        予算テーブル,
        {{"予算年月", each Date.FromText(_), type date}}
    ),

    // 売上データと予算データの結合
    結合 = Table.NestedJoin(
        月次売上,
        {"日付", "商品カテゴリ", "地域"},
        予算日付変換,
        {"予算年月", "商品カテゴリ", "地域"},
        "予算データ",
        JoinKind.LeftOuter
    ),

    // 結合テーブルの展開
    結合展開 = Table.ExpandTableColumn(
        結合,
        "予算データ",
        {"予算金額"},
```

```
        {"予算金額"}
    ),

    // 欠損値の処理
    Null処理 = Table.ReplaceValue(
        結合展開,
        null,
        0,
        Replacer.ReplaceValue,
        {"予算金額"}
    ),

    // 予算達成率の計算
    達成率追加 = Table.AddColumn(
        Null処理,
        "予算達成率",
        each if [予算金額] = 0 then null else [売上金額] / [予算金額],
        type number
    ),

    // 書式の調整
    書式調整 = Table.TransformColumnTypes(
        達成率追加,
        {{"予算達成率", Percentage.Type}}
    )
in
    書式調整
```

> 💡 **ベテランの知恵袋**
> 異なる粒度のデータを統合する際は、「共通分母」を見つけることが重要です。例えば、日次データと月次データがある場合、日次データを月次に集計するか、月次データを日次に配分するかを決める必要があります。一般的には、詳細なデータを集約する方がより正確です。逆に、集計データを詳細レベルに分解する場合は、何らかの配分ルール（均等、加重平均など）を決める必要があります。いずれにしても、選択した方法とその影響をレポートに明記することが重要です。

異なるデータモデルの統合

異なるデータモデルを持つシステムからのデータを統合する方法：

```
// ERPとCRMの異なるデータモデルの統合（M言語）
let
    // ERPからの受注データ
    ERP受注 = Sql.Database("ERPServer", "SalesDB"),
    受注ヘッダ = ERP受注{[Schema="dbo",Item="SalesHeader"]}[Data],
    受注明細 = ERP受注{[Schema="dbo",Item="SalesDetail"]}[Data],

    // ヘッダと明細の結合
    ERP結合 = Table.NestedJoin(
        受注ヘッダ,
```

```
    {"OrderID"},
    受注明細,
    {"OrderID"},
    "明細",
    JoinKind.Inner
),

// 必要な列の展開
ERP展開 = Table.ExpandTableColumn(
    ERP結合,
    "明細",
    {"ProductID", "Quantity", "UnitPrice"},
    {"ProductID", "Quantity", "UnitPrice"}
),

// 必要なカラムの選択と名前の標準化
ERP標準化 = Table.SelectColumns(
    ERP展開,
    {
        {"OrderID", "注文番号"},
        {"OrderDate", "注文日"},
        {"CustomerID", "顧客ID"},
        {"ProductID", "商品ID"},
        {"Quantity", "数量"},
        {"UnitPrice", "単価"}
    }
),

// CRMからの顧客データ
CRM顧客 = Odbc.Query("DSN=CRM", "SELECT * FROM Customers"),

// 必要なカラムの選択と名前の標準化
CRM標準化 = Table.SelectColumns(
    CRM顧客,
    {
        {"CustomerNumber", "顧客ID"},
        {"CustomerName", "顧客名"},
        {"CustomerSegment", "顧客セグメント"},
        {"SalesRepID", "営業担当ID"}
    }
),

// データ型の統一
CRM型変換 = Table.TransformColumnTypes(
    CRM標準化,
    {{"顧客ID", type text}}
),

// ERPとCRMのデータ結合
最終結合 = Table.NestedJoin(
```

```
        ERP標準化,
        {"顧客ID"},
        CRM型変換,
        {"顧客ID"},
        "顧客情報",
        JoinKind.LeftOuter
    ),

    // 結合テーブルの展開
    最終展開 = Table.ExpandTableColumn(
        最終結合,
        "顧客情報",
        {"顧客名", "顧客セグメント", "営業担当ID"},
        {"顧客名", "顧客セグメント", "営業担当ID"}
    )
in
    最終展開
```

2. 時間インテリジェンスの統合

異なるシステムの日付を統合して分析する方法：

```
// 共通の日付テーブルの作成とリレーションシップの設定（DAX）
共通日付テーブル =
ADDCOLUMNS(
    CALENDAR(DATE(2018, 1, 1), DATE(2025, 12, 31)),
    "年", YEAR([Date]),
    "四半期", "Q" & FORMAT([Date], "Q"),
    "月", MONTH([Date]),
    "月名", FORMAT([Date], "MMM"),
    "週番号", WEEKNUM([Date]),
    "曜日", WEEKDAY([Date]),
    "曜日名", FORMAT([Date], "DDDD"),
    "年月", FORMAT([Date], "yyyy/mm"),
    "会計年度", IF(MONTH([Date]) >= 4, "FY" & YEAR([Date]), "FY" & (YEAR([Date])-1)),
    "会計四半期",
        VAR 月 = MONTH([Date])
        VAR 調整月 = IF(月 >= 4, 月 - 3, 月 + 9)
        RETURN
            IF(調整月 <= 3, "Q1",
                IF(調整月 <= 6, "Q2",
                    IF(調整月 <= 9, "Q3", "Q4")
                )
            ),
    "カレンダー種別", "標準"
)

// 複数の日付列を持つファクトテーブルの処理（DAX）
リンク済み販売データ =
```

```
VAR 販売 = 販売データ
VAR 注文日リンク =
    TREATAS(
        VALUES('日付'[Date]),
        販売[注文日]
    )
VAR 出荷日リンク =
    TREATAS(
        VALUES('日付'[Date]),
        販売[出荷日]
    )
VAR 請求日リンク =
    TREATAS(
        VALUES('日付'[Date]),
        販売[請求日]
    )
RETURN
    販売
```

> 🚫 **失敗から学ぶ**
> あるプロジェクトでは、複数システムからのデータを統合する際に、各システムの日付形式の違いを見落としていました。一部のシステムは日付を文字列として保存し、フォーマットも「MM/DD/YYYY」や「DD-MM-YYYY」など様々でした。これにより、1月2日のデータが2月1日として解釈されるなど、深刻なデータ不整合が発生しました。この問題を解決するには、Power Queryでデータを読み込む際に、まず日付のフォーマットを統一的に処理する変換ステップを追加し、すべての日付を標準形式（date型）に変換することが重要です。

複合指標とKPIの作成

複数データソースを組み合わせた高度な指標の作成：

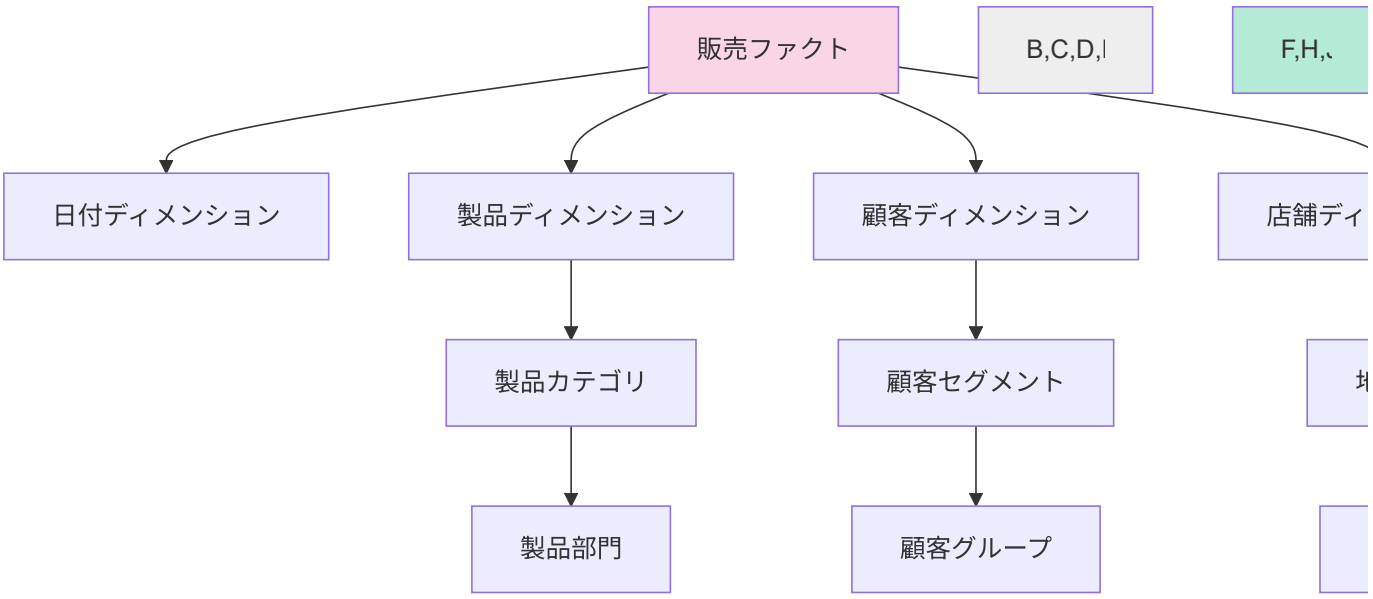
```
// 複合KPIの計算例（DAX）
// 顧客生涯価値（LTV）の計算
顧客生涯価値 =
VAR 平均購入頻度 = [年間平均購入回数]
VAR 平均顧客単価 = [顧客平均購入金額]
VAR 平均顧客寿命 = [平均継続年数]
VAR 平均粗利益率 = [全体粗利益率]
RETURN
    平均購入頻度 * 平均顧客単価 * 平均顧客寿命 * 平均粗利益率

// 顧客獲得コスト対顧客生涯価値比率
CAC対LTV比率 =
DIVIDE(
    [顧客獲得コスト],
    [顧客生涯価値],
    0
)
```

```
// 地域別市場シェア
市場シェア =
VAR 対象地域売上 = [売上合計]
VAR 市場規模 =
    LOOKUPVALUE(
        市場データ[市場規模],
        市場データ[地域], MAX(地域[地域名]),
        市場データ[年], MAX('日付'[年])
    )
RETURN
    DIVIDE(対象地域売上, 市場規模, 0)

// クロスセル率
クロスセル率 =
VAR 複数カテゴリ購入顧客 =
    CALCULATE(
        DISTINCTCOUNT(販売[顧客ID]),
        CALCULATETABLE(
            SUMMARIZE(
                販売,
                販売[顧客ID],
                "カテゴリ数", DISTINCTCOUNT(製品[カテゴリ])
            ),
            [カテゴリ数] > 1
        )
    )
VAR 全顧客数 = DISTINCTCOUNT(販売[顧客ID])
RETURN
    DIVIDE(複数カテゴリ購入顧客, 全顧客数, 0)
```

高度なデータモデリング手法

複雑な分析要件を満たすための高度なデータモデリング手法：

1. スター・スキーマとスノーフレーク・スキーマの組み合わせ



2. ブリッジテーブルによる多対多関係の実装

```
// 多対多関係の実装（M言語）
let
    // 製品とカテゴリの多対多関係
    製品 = 製品マスター,
    カテゴリ = カテゴリマスター,

    // 製品カテゴリマッピングテーブル
    マッピング = Excel.Workbook(File.Contents("ProductCategoryMapping.xlsx"), null,
true),
    製品カテゴリマップ = マッピング{[Name="Mapping",Kind="Table"]}[Data],

    // データ型の変換
    型変換 = Table.TransformColumnTypes(
        製品カテゴリマップ,
        {
            {"ProductID", type text},
            {"CategoryID", type text}
        }
    )
in
    型変換

// リレーションシップの設定（Power BIデータモデル）
// 1. 製品テーブルと製品カテゴリマップの間に1対多の関係
// 2. カテゴリテーブルと製品カテゴリマップの間に1対多の関係
// 3. 製品カテゴリマップを「多対多」カーディナリティで設定
```

3. 計算グループの活用（Power BI Premium機能）

```
// 計算グループの定義（DAX）
// 期間比較用の計算グループ
期間比較 =
NAMEOF(
    "当期", [売上金額],
    "前年同期", CALCULATE([売上金額], SAMEPERIODLASTYEAR('日付'[Date])),
    "前年比", DIVIDE([売上金額], CALCULATE([売上金額], SAMEPERIODLASTYEAR('日付'[Date]))),
    "前期", CALCULATE([売上金額], DATEADD('日付'[Date], -1, QUARTER)),
    "前期比", DIVIDE([売上金額], CALCULATE([売上金額], DATEADD('日付'[Date], -1,
QUARTER))))

// 指標グループの定義
主要指標 =
NAMEOF(
    "売上", [売上金額],
    "販売数", [販売数],
    "平均単価", [平均単価],
    "粗利益", [粗利益],
    "粗利益率", [粗利益率]
)
```

複合レポートのパフォーマンス最適化

複数データソースを統合する際のパフォーマンス最適化テクニック：

1. **集計テーブルの活用**:
   - 詳細データと集計データの分離
   - 適切な粒度でのサマリーテーブル作成

```
// 前計算された集計テーブルの作成（M言語）
let
    ソース = 販売詳細データ,
    日別集計 = Table.Group(
        ソース,
        {"日付", "製品カテゴリ", "顧客セグメント", "地域"},
        {
            {"売上", each List.Sum([売上金額]), type number},
            {"数量", each List.Sum([数量]), type number},
            {"原価", each List.Sum([原価]), type number},
            {"取引数", each Table.RowCount(_), Int64.Type}
        }
    ),
    月別集計 = Table.Group(
        日別集計,
        {
            each Date.Year([日付]),
            each Date.Month([日付]),
            "製品カテゴリ",
            "顧客セグメント",
```

```
            "地域"
        },
        {
            {"年", each List.First([年])},
            {"月", each List.First([月])},
            {"月間売上", each List.Sum([売上]), type number},
            {"月間数量", each List.Sum([数量]), type number},
            {"月間原価", each List.Sum([原価]), type number},
            {"月間取引数", each List.Sum([取引数]), Int64.Type}
        }
    )
in
    月別集計
```

2. **インポートモードとDirectQueryの使い分け**:
   - 基幹データはインポートモード
   - 頻繁に更新される大量データはDirectQuery
   - 複合モデルの活用
3. **計算の最適化**:
   - メジャーの依存関係の最小化
   - VAR変数の活用
   - 不要なフィルターコンテキスト変更の回避

```
// 最適化前のDAX（非効率）
非効率な計算 =
CALCULATE(
    SUM(販売[売上金額]),
    FILTER(
        ALL(製品),
        RELATED(製品カテゴリ[カテゴリ名]) = "電子機器"
    )
) /
CALCULATE(
    SUM(販売[売上金額]),
    ALL(製品)
)

// 最適化後のDAX
最適化された計算 =
VAR 対象カテゴリ売上 =
    CALCULATE(
        SUM(販売[売上金額]),
        FILTER(
            ALL(製品),
            RELATED(製品カテゴリ[カテゴリ名]) = "電子機器"
        )
    )
VAR 全体売上 =
```

```
    CALCULATE(
        SUM(販売[売上金額]),
        ALL(製品)
    )
RETURN
    DIVIDE(対象カテゴリ売上, 全体売上, 0)
```

> 💡 **ベテランの知恵袋**
> 複数データソースを統合する大規模なレポートでは、「スライス・アンド・ダイス」の原則を覚えておくと良いでしょう。つまり、すべてのデータを常に詳細レベルで保持するのではなく、分析の目的に応じて適切な粒度のデータを準備しておくことです。例えば、経営ダッシュボード用には月次集計、部門分析用には日次集計、詳細分析用には取引レベルというように、目的別にデータセットを用意しておくと、パフォーマンスと使いやすさの両方が向上します。

✓ 複数ソース統合レポートのチェックリスト

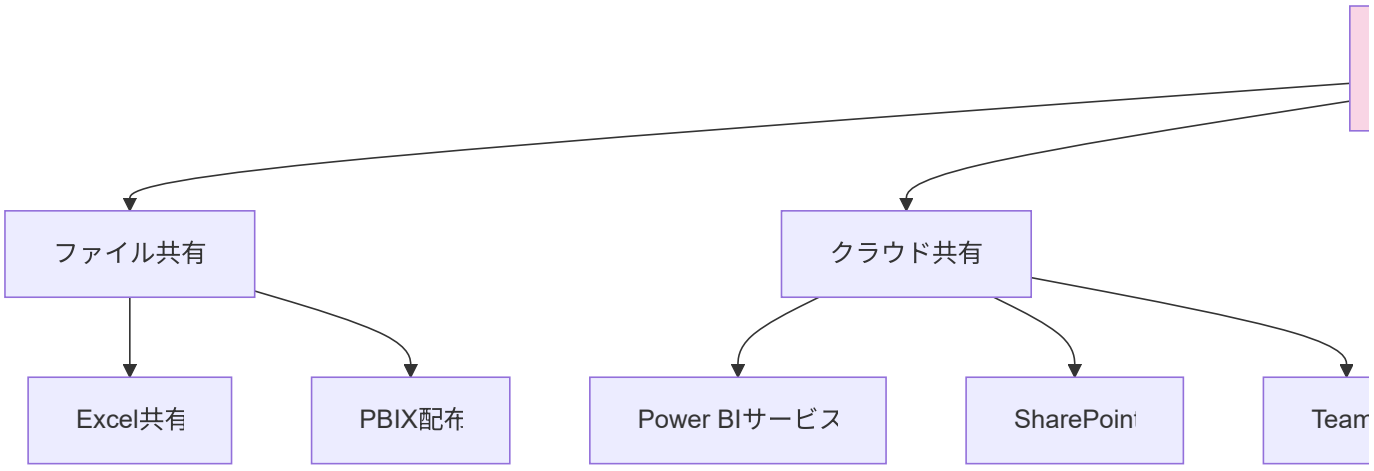
- [ ] 各データソースの構造とリレーションシップを理解した
- [ ] 共通キーを特定し、マッピングテーブルを作成した
- [ ] 日付や数値の形式を標準化した
- [ ] 適切なデータモデル（スター・スキーマなど）を設計した
- [ ] 複合インサイトを生み出すKPIを定義した
- [ ] パフォーマンス最適化のための集計テーブルを作成した
- [ ] 更新スケジュールと依存関係を設定した
- [ ] エラー処理とデータ検証のメカニズムを実装した

10.5 モデルの共有と展開

データモデルの共有と展開は、ビジネスインテリジェンスプロジェクトの最終段階です。効果的な展開と維持管理によって、データ駆動型の意思決定を組織全体に広げることができます。

モデル共有のオプション

Power BIとExcelのデータモデルを共有するための様々な方法：



1. Power BIサービスでの共有

Power BIレポートとダッシュボードを組織内で共有する方法：

1. **ワークスペース**:
   - チーム共同作業のための共有スペース
   - 権限管理（閲覧者、メンバー、寄稿者、管理者）
   - レポートとデータセットの集中管理
2. **アプリ**:
   - パッケージ化されたレポート集
   - ナビゲーションとセクション分け
   - エンドユーザー向けの配布方法
3. **共有リンク**:
   - 特定のユーザーへの直接共有
   - 権限の個別設定
   - 外部ユーザーへの制限付き共有

2. Excelデータモデルの共有

Excelで作成したデータモデルを共有する方法：

1. **ファイル共有**:
   - ネットワーク共有やクラウドストレージでの共有
   - データ接続の相対パス設定
   - 外部データ接続の管理
2. **Power BI連携**:
   - ExcelブックをPower BIにアップロード
   - Excel OnlineでのPower BIビジュアル埋め込み
   - Power QueryクエリのPower BIでの再利用

```
// Excelファイルパスのパラメータ化（M言語）
let
    // ファイルパスをパラメータ化
    ファイルパス = Excel.CurrentWorkbook(){[Name="パラメータ"]}[Content]{0}[ファイルパス],
    ソース = Excel.Workbook(File.Contents(ファイルパス), null, true),
    データシート = ソース{[Item="売上データ",Kind="Sheet"]}[Data],

    // 以降の処理
    ヘッダー昇格 = Table.PromoteHeaders(データシート, [PromoteAllScalars=true])
in
    ヘッダー昇格
```

> **❓ 若手の疑問解決**
> **Q**: Power BIレポートを共有する最適な方法は何ですか？
> **A**: 最適な共有方法は利用シーンによって異なります。少人数チーム内での共有なら「直接共有」が簡単です。部門全体での共有なら「ワークスペース」と「アプリ」の組み合わせが効果的です。組織全体に標準ダッシュボードを展開する場合は「アプリ」と「アプリの自動インストール」が適しています。また、社外の顧客やパートナーと共有する場合は「Power BI Embedded」や「セキュアなリンク共有」を検討すべきです。重要なのは、データの機密性と更新頻度に合わせて方法を選ぶことです。

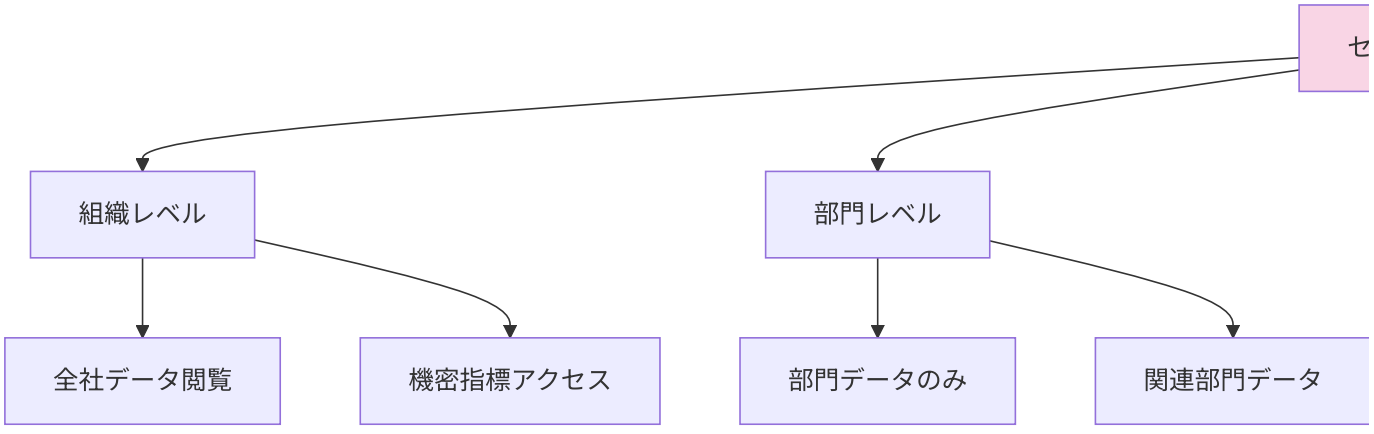
セキュリティとアクセス制御

データモデルとレポートのセキュリティを確保する方法：

1. 行レベルセキュリティの実装

```
// 役割ベースのセキュリティ実装（DAX）
// 営業担当者用のセキュリティフィルター
営業担当者セキュリティ =
DEFINE
    ROLE 営業担当者 AS
    [顧客].[担当営業ID] = USERPRINCIPALNAME() ||
    USERNAME() = "admin@contoso.com"

// 部門別セキュリティ
部門別セキュリティ =
DEFINE
    ROLE 営業部 AS
        [部門].[部門名] = "営業部"
    ROLE 財務部 AS
        [部門].[部門名] = "財務部"
    ROLE 管理部 AS
        [部門].[部門名] = "管理部" ||
        [部門].[部門名] = "人事部"
    ROLE 経営陣 AS
        TRUE()
```

2. データアクセスレベルの設計



3. データセットの共有とライセンス

```
// データセット共有の設定
1. Power BIサービスでデータセットを共有
2. ワークスペースアクセス権の設定
   - ビルド権限：レポート作成可能
```

```
    - 読み取り権限：レポート閲覧のみ
3. データセットの認証設定
    - ゲートウェイ接続の構成
    - 資格情報の保存
    - 更新スケジュールの設定
```

> **🚫 失敗から学ぶ**
> ある企業では、社内の全ダッシュボードを「全員に共有」設定で公開していました。しかし、適切な行レベルセキュリティを設定していなかったため、営業担当者が他チームの顧客データにもアクセスできてしまい、データ漏洩の懸念が生じました。この問題を解決するため、ユーザーのActive Directoryグループに基づく行レベルセキュリティを実装し、各ユーザーが自分の担当データのみを閲覧できるよう制限しました。さらに、Power BIの監査ログを定期的に確認するプロセスを導入し、不適切なアクセスがないか監視する体制も整えました。

継続的なメンテナンスと管理

データモデルとレポートの長期的な管理：

1. バージョン管理と変更履歴

```
// バージョン管理の例
1. ソース管理システムの利用（Git、Azure DevOpsなど）
2. 変更ログの実装
    - レポート内に変更履歴ページを作成
    - 更新日時とバージョン番号の記録
    - 主要な変更点の文書化

// 変更履歴テーブルの例（M言語）
let
    ソース = #table(
        type table [バージョン = text, 更新日 = date, 更新者 = text, 変更内容 = text],
        {
            {"1.0.0", #date(2023, 1, 15), "山田太郎", "初回リリース"},
            {"1.0.1", #date(2023, 2, 3), "佐藤花子", "販売予測ダッシュボード追加"},
            {"1.1.0", #date(2023, 3, 10), "鈴木一郎", "地域別分析機能追加"},
            {"1.1.1", #date(2023, 3, 22), "山田太郎", "データ更新頻度を日次に変更"}
        }
    )
in
    ソース
```

2. パフォーマンスモニタリングと最適化

```
// パフォーマンスモニタリングダッシュボード
// Power BI管理用メトリクス（DAX）
データセットサイズ =
MEASURE 'パフォーマンス指標'[データセットサイズ] =
    FIRSTNONBLANK('管理ログ'[データセットサイズMB], 0)
```

```
平均クエリ時間 =
MEASURE 'パフォーマンス指標'[平均クエリ時間] =
    AVERAGE('クエリログ'[実行時間_ミリ秒])

ユーザーセッション数 =
MEASURE 'パフォーマンス指標'[日次アクティブユーザー] =
    DISTINCTCOUNT('利用ログ'[ユーザーID])
```

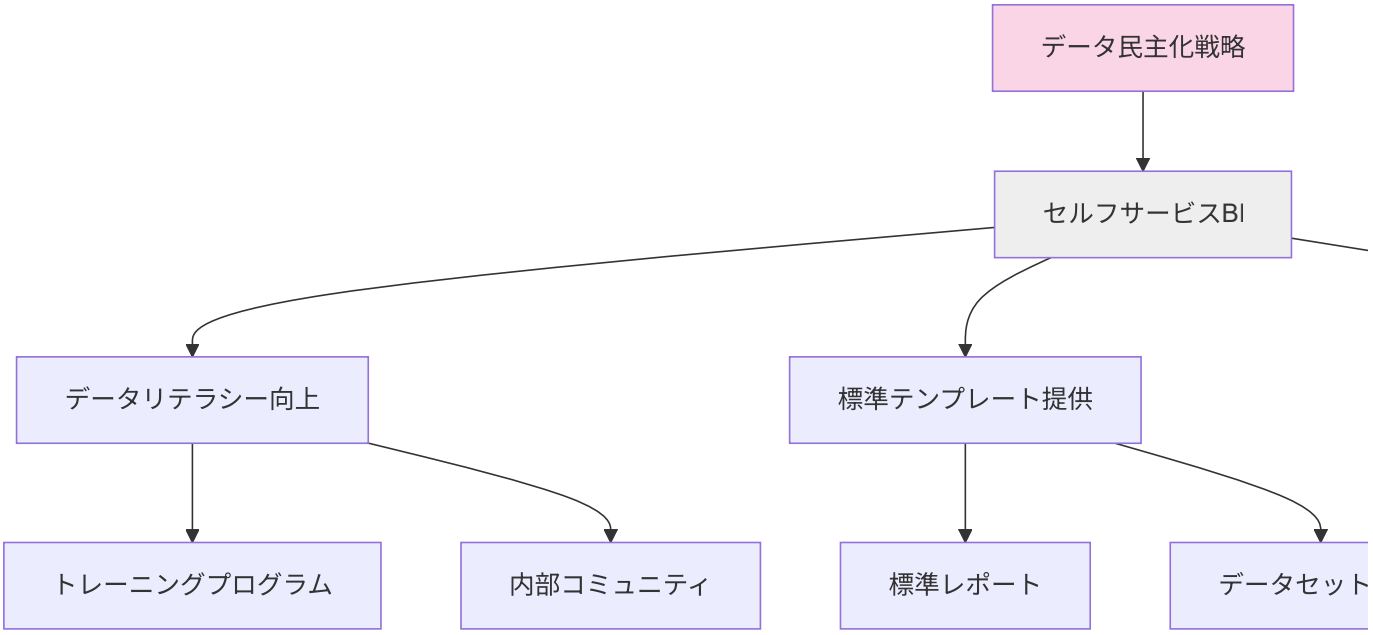
3. データパイプラインの自動化

```
// Power Automateを使用したデータ更新の自動化
1. トリガー：スケジュール（毎日午前7時）
2. アクション：Power BIデータセット更新
   - ワークスペース：販売分析
   - データセット：販売実績データ
3. 条件分岐：
   - 成功：Teams通知「データ更新完了」
   - 失敗：メール通知「更新エラー」+ エラー詳細

// 増分更新の設定（Power BI Premium機能）
1. データセット設定で増分更新を有効化
2. 履歴保持ポリシー：過去2年分
3. 増分更新期間：過去7日分
4. リフレッシュ頻度：
   - 増分部分：1日2回（午前9時・午後5時）
   - 完全更新：毎週月曜日午前3時
```

データ駆動型組織への展開

データモデルとレポートを組織全体に展開し、データ駆動型の意思決定を促進する方法：

1. セルフサービスBIの促進



2. データ文化の醸成

```
// データ文化構築アプローチ
1. エグゼクティブスポンサーシップの確立
   - 経営層による定期的なダッシュボードレビュー
   - データ駆動型意思決定の奨励

2. データリテラシー向上プログラム
   - レベル別トレーニング
   - 認定プログラム
   - 継続的な学習機会

3. コミュニティ構築
   - Power BIユーザーグループ
   - 定期的なナレッジシェアセッション
   - 社内コンペティション

4. 成功事例の共有
   - 事例集の作成
   - データ活用による成功ストーリーの共有
   - 定量的効果の測定と公表
```

> **💼 プロジェクト事例**
> ある製造業では、Power BIの導入初期に「センターオブエキセレンス」チームを結成し、データモデルとレポートの標準化を進めました。各部門から1名ずつ「データチャンピオン」を選出し、彼らに集中的なトレーニングを提供。その後、彼らが各部門でのPower BI活用を推進する役割を担いました。また、共通のデータモデルとテンプレートを開発し、全社で再利用できるようにしました。その結果、導入から1年で社内の90%の部門がPower BIを日常的に活用するようになり、レポート作成時

> 間が平均70%削減されました。重要なのは、技術だけでなく「人」と「プロセス」に焦点を当てたアプローチでした。

✓ モデル共有と展開のチェックリスト

- [ ] 適切な共有方法を選択した（ファイル共有、Power BIサービスなど）
- [ ] データセットのセキュリティを設定した（行レベルセキュリティなど）
- [ ] 更新スケジュールとデータパイプラインを構築した
- [ ] ドキュメントとバージョン管理の仕組みを確立した
- [ ] パフォーマンスモニタリングの仕組みを導入した
- [ ] ユーザートレーニングと支援体制を整備した
- [ ] フィードバック収集と継続的改善のプロセスを確立した

付録

A. Power Query関数クイックリファレンス

テーブル操作関数

関数名	説明	例
Table.AddColumn	テーブルに列を追加	`Table.AddColumn(テーブル, "新列名", each [列1] + [列2])`
Table.SelectColumns	特定の列を選択	`Table.SelectColumns(テーブル, {"列1", "列2"})`
Table.RemoveColumns	特定の列を削除	`Table.RemoveColumns(テーブル, {"不要列1", "不要列2"})`
Table.RenameColumns	列名を変更	`Table.RenameColumns(テーブル, {{"旧名", "新名"}})`
Table.TransformColumns	複数列に変換を適用	`Table.TransformColumns(テーブル, {{"列1", Text.Trim}})`
Table.SelectRows	条件に一致する行を選択	`Table.SelectRows(テーブル, each [列1] > 100)`
Table.Sort	テーブルを並べ替え	`Table.Sort(テーブル, {{"列1", Order.Descending}})`
Table.Group	グループ化と集計	`Table.Group(テーブル, {"グループ列"}, {{"合計", each List.Sum([値]), type number}})`
Table.Join	テーブルの結合	`Table.Join(テーブル1, "結合キー", テーブル2, "結合キー", JoinKind.Inner)`
Table.Combine	テーブルの縦結合	`Table.Combine({テーブル1, テーブル2, テーブル3})`
Table.Distinct	重複行の削除	`Table.Distinct(テーブル, {"キー列"})`
Table.PromoteHeaders	1行目をヘッダーに設定	`Table.PromoteHeaders(テーブル)`
Table.Pivot	ピボットテーブル作成	`Table.Pivot(テーブル, 列一覧, "ピボット列", "値列")`

関数名	説明	例
Table.Unpivot	アンピボット変換	`Table.Unpivot(テーブル, {"維持列"}, "属性", "値")`

テキスト操作関数

関数名	説明	例
Text.Trim	前後の空白を削除	`Text.Trim(" テキスト ")`
Text.Clean	印刷できない文字を削除	`Text.Clean(テキスト)`
Text.Length	テキストの長さを取得	`Text.Length("テキスト")`
Text.Lower	小文字に変換	`Text.Lower("TEXT")`
Text.Upper	大文字に変換	`Text.Upper("text")`
Text.Proper	先頭文字を大文字に	`Text.Proper("sample text")`
Text.Combine	文字列を結合	`Text.Combine({"a", "b", "c"}, ",")`
Text.Split	文字列を分割	`Text.Split("a,b,c", ",")`
Text.Contains	部分文字列を含むか	`Text.Contains("テキスト", "スト")`
Text.StartsWith	指定文字列で始まるか	`Text.StartsWith("テスト", "テ")`
Text.EndsWith	指定文字列で終わるか	`Text.EndsWith("テスト", "ト")`
Text.Replace	文字列置換	`Text.Replace("abcabc", "a", "x")`

日付と時刻の関数

関数名	説明	例
Date.From	日付に変換	`Date.From("2023-01-15")`
DateTime.LocalNow	現在の日時	`DateTime.LocalNow()`
Date.Year	年を抽出	`Date.Year(#date(2023, 5, 15))`
Date.Month	月を抽出	`Date.Month(#date(2023, 5, 15))`
Date.Day	日を抽出	`Date.Day(#date(2023, 5, 15))`
Date.AddDays	日数を加算	`Date.AddDays(#date(2023, 5, 15), 10)`
Date.AddMonths	月数を加算	`Date.AddMonths(#date(2023, 5, 15), 3)`
Date.AddYears	年数を加算	`Date.AddYears(#date(2023, 5, 15), 1)`
Date.EndOfMonth	月末日を取得	`Date.EndOfMonth(#date(2023, 5, 15))`
Date.DayOfWeek	曜日番号を取得	`Date.DayOfWeek(#date(2023, 5, 15))`
Date.DaysInMonth	月の日数を取得	`Date.DaysInMonth(#date(2023, 2, 1))`

数値処理関数

関数名	説明	例
Number.Round	四捨五入	`Number.Round(3.14159, 2)`
Number.RoundDown	切り捨て	`Number.RoundDown(3.9)`
Number.RoundUp	切り上げ	`Number.RoundUp(3.1)`
Number.Abs	絶対値	`Number.Abs(-42)`
Number.FromText	文字列から数値へ変換	`Number.FromText("123.45")`
Number.ToText	数値から文字列へ変換	`Number.ToText(123.45, "0.00")`
Number.IsEven	偶数か判定	`Number.IsEven(4)`
Number.IsOdd	奇数か判定	`Number.IsOdd(3)`

リスト操作関数

関数名	説明	例
List.Count	リストの要素数	`List.Count({1, 2, 3})`
List.Sum	合計	`List.Sum({1, 2, 3})`
List.Average	平均	`List.Average({1, 2, 3})`
List.Min	最小値	`List.Min({1, 2, 3})`
List.Max	最大値	`List.Max({1, 2, 3})`
List.Contains	値の存在確認	`List.Contains({1, 2, 3}, 2)`
List.Select	条件に合う要素を選択	`List.Select({1, 2, 3}, each _ > 1)`
List.Transform	各要素に関数を適用	`List.Transform({1, 2, 3}, each _ * 2)`
List.Distinct	重複を除去	`List.Distinct({1, 2, 2, 3})`
List.Sort	リストを並べ替え	`List.Sort({3, 1, 2})`

条件分岐と論理演算

関数名	説明	例
if ... then ... else	条件分岐	`if x > 10 then "大" else "小"`
and	論理積	`[価格] > 100 and [在庫] > 0`
or	論理和	`[部門] = "営業" or [部門] = "販売"`
not	論理否定	`not [完了]`
true	真定数	`true`
false	偽定数	`false`
null	ヌル値	`null`

B. DAX関数クイックリファレス

集計関数

関数名	説明	例
SUM	合計を計算	`SUM(売上[金額])`
AVERAGE	平均を計算	`AVERAGE(売上[金額])`
MIN	最小値を取得	`MIN(売上[日付])`
MAX	最大値を取得	`MAX(売上[金額])`
COUNT	値のカウント	`COUNT(顧客[顧客ID])`
COUNTA	空でない値をカウント	`COUNTA(コメント[内容])`
COUNTROWS	行数をカウント	`COUNTROWS(テーブル)`
DISTINCTCOUNT	一意の値をカウント	`DISTINCTCOUNT(売上[顧客ID])`
COUNTX	式の結果の行数	`COUNTX(テーブル, [列] > 0)`
SUMX	式の結果の合計	`SUMX(テーブル, [数量] * [単価])`
AVERAGEX	式の結果の平均	`AVERAGEX(テーブル, [売上] - [原価])`

フィルター関数

関数名	説明	例
CALCULATE	フィルターコンテキスト変更	`CALCULATE(SUM(売上[金額]), 年[年] = 2023)`
FILTER	条件に基づくフィルタリング	`FILTER(テーブル, [金額] > 1000)`
ALL	フィルターを解除	`ALL(製品)`
ALLEXCEPT	指定列以外のフィルター解除	`ALLEXCEPT(製品, 製品[カテゴリ])`
ALLSELECTED	ビジュアルレベルフィルタ保持	`ALLSELECTED(製品)`
CALCULATETABLE	テーブル式にフィルター適用	`CALCULATETABLE(テーブル, フィルター)`
KEEPFILTERS	既存フィルター保持	`CALCULATE(SUM(売上[金額]), KEEPFILTERS(日付[年] = 2023))`
REMOVEFILTERS	特定フィルター除去	`REMOVEFILTERS(製品)`

日付と時間の関数

関数名	説明	例
DATESYTD	年初から現在までの期間	`CALCULATE(SUM(売上[金額]), DATESYTD('日付'[日付]))`
DATESMTD	月初から現在までの期間	`CALCULATE(SUM(売上[金額]), DATESMTD('日付'[日付]))`

関数名	説明	例
DATESQTD	四半期初から現在までの期間	`CALCULATE(SUM(売上[金額]), DATESQTD('日付'[日付]))`
SAMEPERIODLASTYEAR	前年同期間	`CALCULATE(SUM(売上[金額]), SAMEPERIODLASTYEAR('日付'[日付]))`
DATEADD	期間の加減算	`CALCULATE(SUM(売上[金額]), DATEADD('日付'[日付], -1, MONTH))`
DATESBETWEEN	指定期間のすべての日付	`CALCULATE(SUM(売上[金額]), DATESBETWEEN('日付'[日付], [開始日], [終了日]))`
PARALLELPERIOD	並行期間	`CALCULATE(SUM(売上[金額]), PARALLELPERIOD('日付'[日付], -1, YEAR))`
STARTOFMONTH	月の初日	`STARTOFMONTH('日付'[日付])`
ENDOFMONTH	月の末日	`ENDOFMONTH('日付'[日付])`
STARTOFQUARTER	四半期の初日	`STARTOFQUARTER('日付'[日付])`
ENDOFQUARTER	四半期の末日	`ENDOFQUARTER('日付'[日付])`

テキスト関数

関数名	説明	例
CONCATENATE	2つの文字列を連結	`CONCATENATE(顧客[姓], 顧客[名])`
CONCATENATEX	テーブル内の値を連結	`CONCATENATEX(テーブル, [列], ",")`
FORMAT	書式設定	`FORMAT([日付], "yyyy年MM月")`
LEFT	左からの文字取得	`LEFT([商品コード], 3)`
RIGHT	右からの文字取得	`RIGHT([電話番号], 4)`
MID	指定位置からの文字取得	`MID([コード], 3, 2)`
FIND	部分文字列の位置	`FIND("-", [コード], 1, 0)`
LEN	文字列の長さ	`LEN([説明])`
UPPER	大文字に変換	`UPPER([名前])`
LOWER	小文字に変換	`LOWER([メールアドレス])`

数学・統計関数

関数名	説明	例
ABS	絶対値	`ABS([変動額])`
ROUND	四捨五入	`ROUND([金額], 0)`
ROUNDUP	切り上げ	`ROUNDUP([金額], 0)`
ROUNDDOWN	切り捨て	`ROUNDDOWN([金額], 0)`
INT	整数部分	`INT([数値])`
MOD	剰余（割り算の余り）	`MOD([ID], 2)`

関数名	説明	例
DIVIDE	安全な除算	`DIVIDE([売上], [数量], 0)`
RANKX	ランキング	`RANKX(ALL(製品), [売上合計])`
PERCENTILE.EXC	パーセンタイル	`PERCENTILE.EXC(テーブル[値], 0.9)`

条件付き関数と論理演算

関数名	説明	例
IF	条件分岐	`IF([売上] > 1000000, "A", "B")`
SWITCH	複数条件の分岐	`SWITCH([ランク], 1, "Gold", 2, "Silver", "Bronze")`
AND	論理積	`AND([売上] > 0, [利益] > 0)`
OR	論理和	`OR([状態] = "完了", [状態] = "承認済")`
NOT	論理否定	`NOT([無効フラグ])`
IFERROR	エラー処理	`IFERROR([売上]/[原価], 0)`
ISBLANK	空白チェック	`ISBLANK([コメント])`
ISFILTERED	フィルター適用確認	`ISFILTERED(製品[カテゴリ])`

リレーションシップと関連関数

関数名	説明	例
RELATED	1対多関連の取得	`RELATED(顧客[顧客名])`
RELATEDTABLE	多対1関連テーブル取得	`RELATEDTABLE(受注明細)`
USERELATIONSHIP	非アクティブ関連を使用	`CALCULATE(SUM(売上[金額]), USERELATIONSHIP(売上[出荷日], 日付[日付]))`
CROSSFILTER	リレーションシップ方向変更	`CALCULATE(SUM(売上[金額]), CROSSFILTER(顧客[顧客ID], 売上[顧客ID], BOTH))`
TREATAS	仮想リレーションシップ作成	`TREATAS(VALUES(商品[商品ID]), 在庫[商品ID])`

階層とパス関数

関数名	説明	例
PATH	親子階層パスの生成	`PATH(製品[製品ID], 製品[親製品ID])`
PATHITEM	パスから特定アイテム抽出	`PATHITEM([製品パス], 1)`
PATHCONTAINS	パスに特定値が含まれるか	`PATHCONTAINS([製品パス], "100")`
PATHLENGTH	パスの長さ	`PATHLENGTH([製品パス])`

変数と評価順序

関数名	説明	例
VAR	変数定義	`VAR 売上合計 = SUM(売上[金額])`
RETURN	変数使用後の戻り値	`RETURN 売上合計 / 変数2`
CALCULATE	コンテキスト変更	`CALCULATE([合計], フィルター)`
EARLIER	外側イテレーションの値	`EARLIER([列名])`
EARLIEST	最も外側の参照	`EARLIEST([列名])`

テーブル操作関数

関数名	説明	例
SUMMARIZE	テーブルの集計	`SUMMARIZE(販売, 販売[製品], "合計", SUM(販売[金額]))`
SUMMARIZECOLUMNS	複数列でグループ化・集計	`SUMMARIZECOLUMNS(日付[年], 製品[カテゴリ], "売上", SUM(販売[金額]))`
ADDCOLUMNS	テーブルに列追加	`ADDCOLUMNS(テーブル, "新列", [式])`
SELECTCOLUMNS	列の選択	`SELECTCOLUMNS(テーブル, "列1", [列1], "列2", [列2])`
GROUPBY	グループ化	`GROUPBY(テーブル, [列1], "集計", SUMX(CURRENTGROUP(), [値]))`
TOPN	上位N件	`TOPN(10, 製品, [売上])`
CROSSJOIN	クロス結合	`CROSSJOIN(VALUES(地域[地域名]), VALUES(製品[カテゴリ]))`
GENERATE	2つのテーブル結合	`GENERATE(テーブル1, テーブル2)`
GENERATEALL	すべての組み合わせ生成	`GENERATEALL(テーブル1, テーブル2)`

C. よくあるエラーと解決法

Power Queryのエラー

エラー	考えられる原因	解決策
"式にエラーがあります"	構文エラー、変数名間違い	式の構文を確認、変数名のスペルチェック
"指定された列'XX'は存在しません"	列名のスペルミス、大文字小文字の区別	正確な列名を確認、フィールドリストから選択
"式.Error"	データ型の不一致、nullとの演算	null値のチェック、データ型の変換を追加
"サーバーに接続できません"	ネットワーク接続の問題、認証エラー	ネットワーク設定確認、認証情報の更新

エラー	考えられる原因	解決策
"フォルダパスが無効です"	ファイルパスの誤り、アクセス権限	パスの正確性確認、相対パスの使用検討
"クエリがタイムアウトしました"	処理時間が長すぎる	クエリ最適化、フィルタを早期適用
"テーブルを展開できません"	参照テーブルが存在しない	テーブル参照の確認、順序の見直し

解決のための一般的なアプローチ：

1. **段階的なデバッグ**:
   - 各ステップの結果を確認
   - エラーが発生する直前のステップまで実行
   - 小さな変更を加えて再試行
2. **クエリの分割**:
   - 複雑なクエリを小さなステップに分割
   - 中間結果を検証
   - 問題のある部分を特定
3. **データ型の確認**:
   - 演算に関わる列のデータ型を確認
   - 適切な型変換を追加
   - null値や空白の処理を追加

DAXのエラーと解決策

エラー	考えられる原因	解決策
"関数 XXX は集計関数を含む式を引数として受け付けません"	不適切なコンテキスト変換	CALCULATE関数の使用を検討
"引数の型が正しくありません"	データ型の不一致	適切な型変換関数を使用
"テーブル式が必要です"	スカラー式をテーブルとして使用	VALUES()やFILTER()でテーブル式に変換
"循環参照が検出されました"	メジャー間の循環参照	依存関係を見直し、VAR使用を検討
"数式に列の参照が含まれています"	メジャー内での列の直接参照	CALCULATE、SUMX等の適切な関数使用
"XXXテーブルとYYYテーブル間の関係が見つかりません"	リレーションシップの欠如	データモデルの見直し、USERELATIONSHIPの使用

DAXパフォーマンス改善のヒント：

1. **VAR変数の活用**:
   - 複数回使用する式を変数に格納
   - 可読性向上と計算の効率化
2. **フィルターの適切な使用**:
   - 必要最小限のフィルタリング
   - ALLの過剰使用を避ける

- 大きなテーブルのフィルタは早期に適用

3. **計算列とメジャーの使い分け**:
   - 行レベルの計算は計算列に
   - 集計や動的計算はメジャーに
4. **関数の適切な選択**:
   - SUM vs SUMX（単純合計か行ごとの計算か）
   - FILTER vs CALCULATETABLE（パフォーマンス考慮）
   - RELATED vs LOOKUPVALUE（状況に応じて）

D. 便利なリソースとコミュニティ

公式ドキュメントとチュートリアル

リソース	URL	概要
Microsoft Power BI公式サイト	https://powerbi.microsoft.com/	公式情報、更新情報、ダウンロード
Power BIドキュメント	https://docs.microsoft.com/power-bi/	公式ドキュメント、チュートリアル
Power Query M関数リファレンス	https://docs.microsoft.com/powerquery-m/	M言語の完全なリファレンス
DAX関数リファレンス	https://docs.microsoft.com/dax/	DAX関数の完全なリファレンス
Microsoft Learn	https://docs.microsoft.com/learn/powerplatform/	体系的な学習コンテンツ
Power BI Blog	https://powerbi.microsoft.com/blog/	最新情報、ベストプラクティス

コミュニティリソース

リソース	URL	概要
Power BIコミュニティ	https://community.powerbi.com/	質問、回答、ディスカッション
SQLBI	https://www.sqlbi.com/	DAXの専門家によるコンテンツ
Guy in a Cube	https://guyinacube.com/	チュートリアル動画、テクニック
Enterprise DNA	https://enterprisedna.co/	トレーニング、テンプレート
Power BI日本ユーザーグループ	https://pbijug.com/	日本語のコミュニティ
Power Query.Training	https://powerquery.training/	Power Query専門のトレーニング
DAX Patterns	https://www.daxpatterns.com/	DAXの一般的なパターン集

書籍とブログ

リソース	著者	概要
Definitive Guide to DAX	Marco Russo & Alberto Ferrari	DAXの決定版解説書

リソース	著者	概要
Power Query for Power BI and Excel	Chris Webb	Power Query詳細解説
Power Pivot and Power BI	Rob Collie & Avichal Singh	実践的なPower BI活用法
SQLBI Blog	Marco Russo & Alberto Ferrari	高度なDAXテクニック
Curbal	Ruth Pozuelo	実践的なPower BIテクニック
Kasper On BI	Kasper de Jonge	Microsoft社員による技術解説
データ分析の実務プロセス	清水龍太郎	日本語での実践的なデータ分析手法

ツールとアドイン

ツール名	用途	入手先
DAX Studio	DAXクエリの最適化、デバッグ	https://daxstudio.org/
Tabular Editor	高度なモデリング	https://tabulareditor.com/
ALM Toolkit	モデル比較とデプロイ	https://tabulareditor.github.io/ALMToolkit/
Power BI Helper	モデル分析とドキュメント化	https://radacad.com/power-bi-helper
Vertipaq Analyzer	モデルパフォーマンス分析	https://www.sqlbi.com/tools/vertipaq-analyzer/
Power BI Theme Builder	カスタムテーマ作成	https://themes.powerbi.tips/
DAX Formatter	DAX式の整形	https://www.daxformatter.com/

認定資格

資格名	レベル	概要
PL-300: Microsoft Power BI Data Analyst	基本〜中級	Power BIを使ったデータ分析の基本スキル
DA-100: Analyzing Data with Microsoft Power BI	基本〜中級	PL-300の前身（同等レベル）
PL-500: Microsoft Power Automate RPA Developer	中級	Power Platformを使った自動化
Microsoft Certified: Data Analyst Associate	基本〜中級	データ分析の認定資格
MCSA: BI Reporting	中級	Power BIとExcelによるBI報告
Power BI認定資格	基本〜上級	Enterprise DNAなど民間の認定

以上で「Excel & Power BI データ変換マスターガイド: Power Query, Power FX, DAXを極める」の内容を完了します。本書が読者の皆様のデータ分析スキル向上に役立ち、業務効率化やより深い分析の一助となれば幸いです。データ分析は日々進化する分野です。ぜひ継続的な学習とコミュニティへの参加を通じて、スキルを磨き続けてください。

著者一同