目次

改訂情報

変更年月日	変更内容
2014-05-01	初版
2014-08-01	第2版 下記を追加・変更しました ■ 「*検索対象の作成・登録・削除*」および「*検索処理の実装*」のサンプルコード内の誤記を修正
2014-09-01	第3版 下記を追加・変更しました ■ 「*検索対象の作成・登録・削除*」および「*標準の全文検索画面の拡張*」に複数のTYPEフィールドと絞り込み条件の階層化についての記載を追加
2015-08-01	第4版 下記を追加・変更しました ■ 「*検索対象の作成・登録・削除*」の絞り込み条件（TYPEフィールド）の名称についての重複した記載を削除
2018-12-01	第5版 下記を追加しました ■ 「*製品にて利用しているTYPE*」に「intra-mart Accel GroupMail」および「IM-Knowledge」で利用しているTYPEフィールドの値を追加しました。

はじめに

本書の目的

本書では IM-ContentsSearch for Accel Platform の機能を利用して、独自に検索機能の開発を行うために必要となる基本的な方法や注意点等について説明します。

対象読者

次の開発者を対象としています。

- 独自に開発したアプリケーションを IM-ContentsSearch による全文検索の対象としたい。
- 独自の全文検索画面を作成したい。

次の内容を理解していることが必須となります。

- Javaの基礎

対象開発モデル

本書では以下の開発モデルを対象としています。

- JavaEE開発モデル （検索対象の作成）
- スクリプト開発モデル （検索結果テンプレートの作成）

用語解説

本書における文言の表記について記載します。

- IM-ContentsSearch for Accel Platform を省略して IM-ContentsSearch と表記します。
- Apache Solr を省略して Solr と表記します。
- リレーショナルデータベース を省略して RDB と表記します。

サンプルコードについて

本書に掲載されているサンプルコードは可読性を重視しており、性能面や保守性といった観点において必ずしも適切な実装ではありません。
サンプルコードを参考に開発される場合には、十分に注意してください。

本書の構成

- *基本概念*

  IM-ContentsSearch の概要や動作の概念について説明します。

- *共通仕様*

  検索処理の実装、および、検索対象となるコンテンツ作成の実装に共通して必要となる情報について説明します。

- *検索処理の実装*

  検索処理のプログラミング方法について説明します。

- *検索対象の作成・登録・削除*

  全文検索対象となるコンテンツの作成、および、クローラを作成するプログラミング方法について説明します。

- *標準の全文検索画面の拡張*

  標準の全文検索機能を利用するために必要な設定や、プログラミング方法について説明します。

- *付録*

  実装における注意事項などの情報を記載します。

基本概念

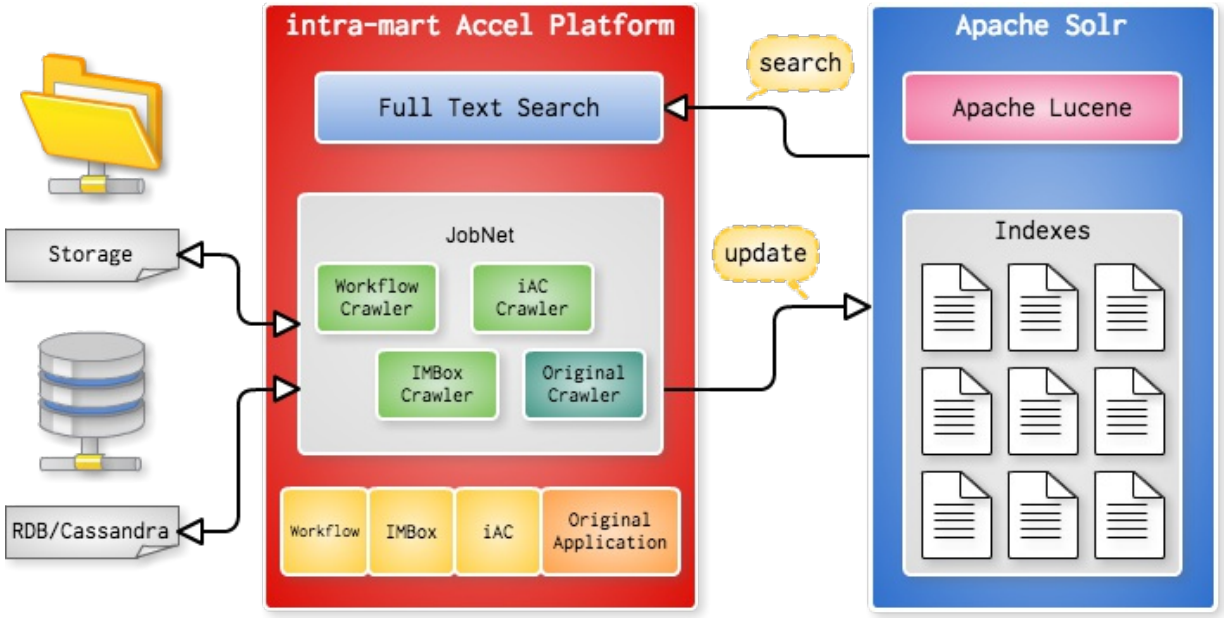
この章では、 IM-ContentsSearch における全文検索の実現方法について説明します。

アーキテクチャ

IM-ContentsSearch が全文検索機能を実現するためのアーキテクチャについて説明します。

概念図

IM-ContentsSearch では全文検索エンジンとして、 Apache Solr を利用しています。
IM-ContentsSearch における全文検索処理の概要図は以下の通りです。



1. クローラ（JobNet）で全文検索対象となるコンテンツを生成し、Solrサーバのインデックスとして登録します。
2. 登録されたインデックスに対して全文検索画面や検索APIから全文検索（Full Text Search）を実行して結果を取得します。

> **コラム**
>
> Apache Solr の概要やセットアップ方法などについては、「Solr管理者ガイド」を参照してください。

コアライブラリの基本機能

IM-ContentsSearch コアライブラリ には、全文検索を実現するために以下の機能を有しています。

- intra-mart Accel Platform と連携して動作するAPI
  - コンテンツ作成API
  - コンテンツ検索API

- クローラ作成用のAPI
  - 基底ジョブ
  - ユーティリティ
- 検索画面
  - 拡張可能な標準全文検索画面
  - グローバル検索ナビ機能 （グローバルナビ右にある虫眼鏡アイコン）

コラム

IM-ContentsSearch のAPIは Java で提供されています。
複雑なデータ構成やデータ型を扱う必要があるため、スクリプト開発向けのAPIを提供する予定はありません。

共通仕様

この章では、 IM-ContentsSearch の機能を利用したプログラミングを行う上で必要となる、共通的な仕様や概念について説明します。

コンテンツ

IM-ContentsSearch では検索結果として表示される 1 つの索引データを **コンテンツ** という形で扱います。
コンテンツの型には、以下の 2 種類が存在します。

1. **登録用コンテンツ**
   新たに検索対象を登録する際に使用するコンテンツです。
   検索結果として表示したい単位データごとに、登録用コンテンツを作成していきます。
   使用方法の詳細や実装例については、*検索対象の作成・登録・削除* の章にて説明します。

2. **検索結果コンテンツ**
   検索処理の実行結果として返却されるコンテンツです。
   検索処理を実行すると、検索結果コンテンツが格納されたリストを取得できます。
   使用方法の詳細や実装例については、*検索処理の実装* の章にて説明します。

IM-ContentsSearch を利用した基本的な処理の流れは以下の通りです。

1. 任意のデータを格納した**登録用コンテンツ**を作成して登録する。
2. 他のアプリケーションと同時に横串検索を可能にするために、標準の全文検索画面用の設定を行う。
3. 検索APIを利用して**検索結果コンテンツ**を取得し、画面の描画など任意の処理に利用する。

フィールド

コンテンツには**フィールド**というデータを格納する単位を保持しており、フィールドの種類によって格納可能なデータ型が異なります。
フィールドは**標準フィールド**と**動的フィールド**の 2 種類が定義されています。

標準フィールド

標準で用意されているフィールドです。
IM-ContentsSearch の標準の全文検索画面に対応するために標準フィールドを設定する必要があります。

標準フィールドを使用するには、下記サンプルのように `Fields` クラスから取得します。

標準フィールドを取得するサンプル

```
// IDフィールド
Fields.ID;

// TYPEフィールド
Fields.TYPE;
```

標準フィールドとして以下が定義されています。

No.	フィールド名	型	必須	検索	説明
1	ID	String	○		コンテンツを一意に判別することが可能なIDを登録するためのフィールドです。 RDBにおけるプライマリキーと同様の役割を果たします。
2	TYPE	String配列	○		**コンテンツ種別** を登録するためのフィールドです。 **コンテンツ種別** の詳細については *検索対象の作成・登録・削除* にて記述します。
3	URL	String			コンテンツの元データにアクセスするためのURLを登録するためのフィールドです。
4	ID_ORIGINAL	String			コンテンツの元データを辿るのに必要なデータなど、必要に応じて任意の文字列を登録するためのフィールドです。 JSON形式で登録するためのユーティリティが用意されています。
5	TITLE	テキスト配列		○	コンテンツにおけるタイトルのテキストデータを登録するためのフィールドです。
6	TEXT	テキスト配列		○	コンテンツのメインとなるテキストデータを登録するためのフィールドです。
7	ATTACHMENT	テキスト配列		○	添付ファイル内のテキストデータを登録するためのフィールドです。 登録時は直接利用せず、APIから添付ファイルの情報を登録することで添付ファイル内の文字列を格納します。
8	RECORD_DATE	Date			標準検索画面の日付レンジ検索にて使用されるフィールドです。 コンテンツの元となる情報が作成された日付を登録することで、標準の全文検索画面にて日付による絞り込み検索が可能です。

- **必須** の項目に ○ が付いているフィールドは、登録時に必ず値を設定する必要があります。
- **検索** の項目に ○ が付いているフィールドは、 IM-ContentsSearch の全文検索画面における検索対象です。

登録用コンテンツには標準フィールドに対するaddメソッド、および、setメソッドが、**検索結果コンテンツ**にはgetメソッドが用意されています。
そのため、通常は **Fields** クラスを利用することなく、以下のように標準フィールドへアクセスが可能です。

コンテンツの標準フィールドを操作するサンプル

```
// 登録用コンテンツ
InputContent inputContent = new StandardInputContent();
inputContent.setId("コンテンツの一意なID");
inputContent.setTitle("コンテンツのタイトル");

// 検索結果用コンテンツ
ResultContent resultContent = // 検索処理にて取得します
String id = resultContent.getId();
String title = resultContent.getTitle();
```

動的フィールド

フィールド名と共に任意のキーを指定することで、標準フィールドにはない独自のデータ型を持ったフィールドの追加が可能です。
動的フィールドを利用してコンテンツに独自データを保持させることで、以下のようなことが実現可能です。

- 標準の全文検索画面における検索結果画面に任意の情報を追加
- コンテンツを作成したタイミングのデータを保持

動的フィールドも標準フィールドと同様に、 `Fields` クラスを利用して取得できます。

動的フィールドを取得するサンプル

```
// 製品名の保存に利用するString型の動的フィールドの例
Fields.STRING.toField("code");

// 価格の保存に利用するint型の動的フィールドの例
Fields.INT.toField("price");
```

> **注意**
>
> システムで予約されているため、次の名称は動的フィールドに利用できません。
>
> - 標準フィールドのフィールド名（大文字、小文字の区別なし）
> - “snippets”
> - “typeBreadcrumbs”

動的フィールドには以下のフィールド（データ型）が定義されています。

No.	フィールド（データ型）	型	説明
1	STRING	String	文字列として登録するためのフィールドです。
2	INT	Integer	整数値として登録するためのフィールドです。
3	LONG	Long	長整数値として登録するためのフィールドです。
4	DATE	Date	日付データとして登録するためのフィールドです。
5	BOOLEAN	Boolean	真偽値を登録するためのフィールドです。
6	NGRAM	N-gram テキスト	文字列に対してN-gram解析を行った結果を登録するためのフィールドです。
7	MORPH	形態素解析 テキスト	文字列に対して形態素解析を行った結果を登録するためのフィールドです。

No.	フィールド（データ型）	型	説明
8	WHITESPACE	Whitespace テキスト	文字列に対してWhitespace解析を行った結果を登録するためのフィールドです。
9	STRING_MLT	String配列	文字列の配列を登録するためのフィールドです。
10	INT_MLT	Integer配列	整数値の配列を登録するためのフィールドです。
11	LONG_MLT	Long配列	長整数値の配列を登録するためのフィールドです。
12	DATE_MLT	Date配列	日付データの配列を登録するためのフィールドです。
13	BOOLEAN_MLT	Boolean配列	真偽値の配列を登録するためのフィールドです。
14	NGRAM_MLT	N-gram テキスト配列	文字列の配列に対してN-gram解析を行った結果を登録するためのフィールドです。
15	MORPH_MLT	形態素解析 テキスト配列	文字列の配列に対して形態素解析を行った結果を登録するためのフィールドです。
16	WHITESPACE_MLT	Whitespace テキスト配列	文字列の配列に対してWhitespace解析を行った結果を登録するためのフィールドです。

登録用コンテンツ に動的フィールドの値を追加する場合は、 setValue および addValue メソッドを利用します。
検索結果コンテンツ から動的フィールドの値を取得する場合は、 getValue メソッドを利用します。

コンテンツの動的フィールドを操作するサンプル

```
// 登録用コンテンツ
InputContent inputContent = new StandardInputContent();
inputContent.setValue(Fields.STRING.toField("code"), "サンプル製品");
inputContent.setValue(Fields.INT.toField("price"), 100000);

// 検索結果用コンテンツ
ResultContent resultContent = // 検索処理にて取得します
String code = resultContent.getValue(Fields.STRING.toField("code"));
int price = resultContent.getValue(Fields.INT.toField("price"));
```

権限制御

IM-ContentsSearch ではコンテンツごとに閲覧可能権限の設定が可能です。
コンテンツに設定された権限を利用し、検索を行ったユーザが検索結果として表示されるコンテンツを制御します。

閲覧可能権限の制御は各APIの実装において以下のフローで実現されています。

コンテンツ作成時のフロー（権限の付与）

1. 作成した登録用コンテンツに対して、参照を許可する権限を作成する `ACIBuilder` を追加します。
2. 登録処理の実行時に投稿用のAPIがコンテンツに追加された、 `ACIBuilder` を利用して権限の一覧を作成します。
   作成された権限の一覧はインデックスの権限保持フィールドに保持されます。

全文検索実行時のフロー（権限によるフィルタリング）

1. 検索を行ったユーザが保持するすべての権限を取得します。
2. 取得したすべての権限を、登録済みのインデックスの権限保持フィールドに一致するかを検索条件に付与します。
3. 権限情報がマッチしたコンテンツのみがフィルタリングされ、検索結果として返却されます。

注意

コンテンツの元となるデータの権限に関する権限が変更された場合、その権限でコンテンツを再登録するまで変更前の権限でフィルタリングされます。

検索処理の実装

この章では、 IM-ContentsSearch の検索APIを利用した全文検索の実行方法について説明します。

項目

はじめに

全文検索を実行するために必要な処理は以下の通りです。

1. 検索するためのクエリを作成する。
2. クエリを指定して検索処理を実行し、検索結果を取得する。
3. 取得した検索結果を扱って任意の処理を行う。

検索クエリを作成する

検索クエリは **検索条件** と **取得条件** で構成されています。

- **検索条件**
  取得したいコンテンツの条件を設定します。
  条件にマッチしたコンテンツのみが返却されます。
- **取得条件**
  検索条件にマッチしたコンテンツを更に絞り込むための条件です。
  例えば、 **取得する最大件数** や **取得するフィールド** などを指定します。

まず、 `Condition` のAPIを利用して検索条件を作成します。
次に検索クエリのインスタンスを作成し、作成した検索条件と任意の取得条件を設定します。

検索条件を作成するサンプル

```
// 検索条件の作成
// 例：TYPEが"example_type"であり、テキストデータに"any_keyword"を含み、ID_ORIGINALフィールドに値が存在する
Searchable condition = Condition.type("example_type").
        keyword("any_keyword").
        exists(Fields.ID_ORIGINAL);

// 検索クエリ作成
SearchQuery query = new Query();

// 作成した検索条件の設定
query.setCondition(condition);
// 例：全フィールドのデータを保持するコンテンツを、1件目から開始して100件取得する
query.addFieldToOutput(Fields.ALL).
        setOffset(1).
        setRows(100);
```

> **コラム**
>
> Fields.ALL は取得条件として指定する専用のフィールドになります。
> 検索結果を単純な索引情報として利用する場合には、必要最低限のフィールド（ID_ORIGINAL など）を指定する事により検索処理の負荷を軽減することが可能です。

検索処理を実行する

作成した検索クエリを ContentsSearchManager#search(SearchQuery) に指定して検索を実行します。
検索結果として、 SearchResponse インスタンスが返却されます。

検索を実行するサンプル

```
try {
   // マネージャのインスタンスを取得
   ContentsSearchManager manager = new ContentsSearchManager();
   // 全文検索の実行
   SearchResponse response = manager.search(query);
} catch (InvalidSearchConditionException e) {
   // 検索条件が不正だった場合
} catch (ContentsSearchExecutionException e) {
   // 検索処理に失敗した場合
}
```

検索結果を利用する

検索処理で取得した検索結果（ SearchResponse ）を利用して任意の処理を実装します。

検索結果コンテンツの利用

取得した検索結果から、検索結果コンテンツを取得して利用する方法です。
SearchResponse#getResultContentList() を利用することで、検索結果コンテンツのリストを取得できます。

検索結果コンテンツの処理実装サンプル

```
// 検索結果コンテンツのリストを取得
List<ResultContent> contents = response.getResultContentList();

// 検索結果コンテンツごとに処理を実行
for (ResultContent content : contents) {
   // IDの取得
   String id = content.getId();
   // 動的フィールドの取得
   String anyString = content.getValue(Fields.STRING.toField("any"));

   // ... 任意の処理
}
```

件数情報の利用

検索結果からは検索結果コンテンツ以外に件数に関する情報が取得できます。

共通情報取得のサンプル

```
// 検索条件にヒットした総数（未取得のコンテンツも含む）
long found = response.getFound();

// 実際に取得した検索結果コンテンツの件数
long returned = response.getReturned();

// 検索結果コンテンツの取得開始位置
long offset = response.getOffset();
```

検索対象の作成・登録・削除

この章では、任意のコンテンツを作成、および、登録、削除のプログラミング方法についてサンプルを交えながら説明します。

項目

はじめに

全文検索対象を追加するためにはコンテンツを作成し、APIを利用して登録する必要があります。
登録用のコンテンツを作成する手順は以下の通りです。

1. コンテンツの各フィールドに格納するデータを設計する。
2. APIを利用してコンテンツを作成する。
3. APIを利用してコンテンツを登録する。

具体的な作成処理をイメージするため、次のようなRDBの**製品情報マスタ**テーブルを検索対象のコンテンツとして登録する処理を考えます。
テーブル内の各情報をどのフィールドに登録するか、事前に設計しておく必要があります。

表 製品情報マスタテーブル

No.	論理名	物理名	データ型	備考
1	製品コード	code	varcher (20)	必須, 主キー
2	製品名	name	varcher (200)	必須
3	分類	category	varchar (50)	必須
4	価格	price	number (13,0)	必須
5	説明	description	varcher (1024)	
6	参考ファイル	reference_file	varcher (200)	説明用に添付するファイルのパブリックストレージ上のパスを保存します。
7	作成日時	create_date	timestamp	必須

No.	論理名	物理名	データ型	備考
8	更新日時	record_date	timestamp	必須

標準フィールドの設計

登録用コンテンツの各フィールドに格納する情報を設計する必要があります。
サンプルでは標準フィールドとして製品情報マスタの情報を以下のように設定します。
まず、標準フィールドに設定するTYPE以外の設定値について検討します。

TYPEフィールド

標準フィールドにて必須に定義されている **TYPEフィールド** は、標準検索画面にて下記の3つの用途に使用されます。

- 検索時の絞り込み条件、 **コンテンツ種別** として利用
- 検索結果の表示に利用するテンプレート画面の切り替え
- 検索結果に表示される絞り込み条件 （画面左のツリー）

また、TYPEフィールドに複数の値を指定することでコンテンツ種別に階層情報を持たせることが可能です。
階層情報を持たせることで、検索結果に表示される絞り込み条件が階層化されます。

サンプルでは製品情報マスタに対するTYPEとして **“product_master”** という値を指定します。
さらに、製品情報マスタの分類（category）に応じて下記のTYPEフィールドを追加で指定します。

- 製品情報マスタの「分類」の値が”Base”である場合
  - **“product_master$Base”**
- 製品情報マスタの「分類」の値が”Product”である場合
  - **“product_master$Product”**
- 製品情報マスタの「分類」の値が”eBuilder”である場合
  - **“product_master$eBuilder”**

> **注意**
>
> 検索するコンテンツを分類するため、静的ファセットは検索対象ごとに一意となる値を指定する必要があります。
> *製品にて利用しているTYPE* を確認して、これらの値と重複しない文字列をTYPEフィールドに指定してください。

その他のフィールド

表 標準フィールド（製品情報マスタ）

No.	フィールド名	設定値	説明
1	ID	“product_master_” + %製品コード%	**製品情報マスタ** の主キーである **製品コード** を設定します。システム一意である必要があるため、商品マスタ検索であることを表すプレフィックスを付与します。

No.	フィールド名	設定値	説明
2	TYPE	“product_master” “product_master$Base” （「分類」の値が”Base”である場合） “product_master$Product” （「分類」の値が”Product”である場合） “product_master$eBuilder” （「分類」の値が”eBuilder”である場合）	
3	URL	“product_master/product”	検索結果のタイトル（リンク）をクリックした際にポップアップ表示するURLを設定します。
4	ID_ORIGINAL	%製品コード%	**製品情報マスタ** の主キーである **製品コード** を設定します。
5	TITLE	%製品名%	タイトルとして表示するため **製品名** を設定します。
6	TEXT	%説明%	**説明** に値が登録されている場合に設定します。
7	ATTACHMENT	%参考ファイル%	**参考ファイル** に登録されているパスのパブリックストレージ上に保存されたファイル内容を設定します。 ファイルの設定に関する詳細は、 *検索対象の作成・登録・削除* にて説明します。
8	RECORD_DATE	%更新日付%	

TYPEフィールドについて

標準フィールドにて必須に定義されている**TYPEフィールド**は、標準検索画面にて下記の3つの用途に使用されます。

- 検索時の絞り込み条件、**コンテンツ種別**として利用
- 検索結果の表示に利用するテンプレート画面の切り替え
- 検索結果に表示される絞り込み条件 （画面左のツリー）

また、TYPEフィールドに複数の値を指定することでコンテンツ種別に階層情報を持たせることが可能です。
階層情報を持たせることで、検索結果に表示される絞り込み条件が階層化されます。

サンプルでは製品情報マスタに対するTYPEとして**“product_master”**という値を指定します。
さらに、製品情報マスタの分類（category）に応じてTYPEフィールドを追加で指定しています。（「表 標準フィールド（製品情報マスタ）」参照）

> **注意**
>
> 検索するコンテンツを分類するため、静的ファセットは検索対象ごとに一意となる値を指定する必要があります。
> *製品にて利用しているTYPE* を確認して、これらの値と重複しない文字列をTYPEフィールドに指定してください。

動的フィールドの設計

動的フィールドに設定する設定値について検討します。設定した値は次のような用途に利用可能です。

- 標準の全文検索画面において検索結果に表示する
- 検索APIを用いた任意の検索処理にて検索条件に指定する

サンプルでは製品情報マスタの情報のうち、検索結果画面に表示させる以下の情報を設定します。

表 動的フィールド（製品情報マスタ）

No.	フィールド名	データ型	設定値	説明
1	category	STRING	%分類%	
2	price	INT	%価格%	

コンテンツを作成する

検索結果に独自のコンテンツを表示するためには、登録用コンテンツを作成し登録処理を行う必要があります。

まずは登録用コンテンツのインスタンスを作成し、値を設定していきます。

登録用コンテンツのインスタンス作成サンプル

```
// 登録用コンテンツをインスタンス化します
InputContent content = new StandardInputContent();
```

インスタンスを生成したら、設計した内容に沿って登録用コンテンツにそれぞれの値を設定していきます。
なお前提として、設定する値は事前にRDBから取得してモデル（ `product` ）に格納してあるものとします。

標準フィールドの設定

標準フィールドの設計 にて設計した通りに標準フィールドに値を設定していきます。

標準フィールドを設定するサンプル

```
// 標準フィールドを設定
content.setId("product_master_" + product.getCode()).
        setUrl("product_master/product").
        setOriginalId(product.getCode()).
        setTitle(product.getName()).
        setRecordDate(product.getRecordDate());

// 説明が存在した場合は追加
if (!StringUtil.isBlank(product.getDescription())) {
   content.addText(product.getDescription());
}
```

複数のTYPEフィールドを指定する

標準フィールドの設計 にて設計した複数のTYPEフィールドに値を設定していきます。

複数のTYPEフィールドを指定するサンプル

```
content.setTypes("product_master", "product_master$" + product.getCategory());
```

検索結果画面に表示する絞り込み条件（ファセット）の名称について

検索結果画面に表示する絞り込み条件の名称は、テンプレート設定ファイルにプロパティキーを設定することで表示されます。（詳しくは *テンプレート設定ファイルの詳細* を参照してください。）
テンプレート設定ファイルにプロパティキーを設定していない場合は、**TYPE**フィールドに設定した **“product_master”** に対応する表示名が設定されていないため、検索結果画面の**絞り込み条件**の欄が「未定義 (日本語ロケールの場合)」と表示されます。

テンプレート設定ファイルに設定せず、プログラム内で定義した値やRDBから取得した値などを絞り込み条件の名称として表示するには、動的プロパティ保存API（ DynamicPropertiesHolder ）を利用して、PublicStorageにJSON形式で絞り込み条件の名称を保存させる必要があります。
JSONファイルは下記のディレクトリに保存されます。

- %PUBLIC_STORAGE_PATH%/products/im_contents_search/store/%テナントID%/dynamic_property/type.json

絞り込み条件の名称（カテゴリ名）をPublicStorageに保存するサンプル

```
// ... 省略 ...

import jp.co.intra_mart.foundation.contentssearch.common.DynamicPropertiesHolder;
import jp.co.intra_mart.foundation.contentssearch.exception.ContentsSearchCrawlingException;
import jp.co.intra_mart.foundation.contentssearch.model.field.Fields;
import jp.co.intra_mart.foundation.contentssearch.web.model.DynamicFacetInputValue;
import jp.co.intra_mart.foundation.contentssearch.web.util.FacetUtil;
import jp.co.intra_mart.foundation.i18n.locale.LocaleInfo;
import jp.co.intra_mart.foundation.i18n.locale.SystemLocale;

// ... 省略 ...

   /** タイプ用の動的プロパティホルダー */
   private final DynamicPropertiesHolder dynamicPropertiesHolder =
DynamicPropertiesHolder.getHolder(Fields.TYPE.getName());

   /**
    * コンテンツのタイプを取得
    * @return タイプ
    */
   protected String getType() {
      return "product_master";
   }

// ... 省略 ...

   /**
    * 動的ファセットデータとしてカテゴリのキーを保存します。
    * @param category カテゴリ
    * @throws ContentsSearchCrawlingException
    */
   private void saveFacetType(final String category) throws ContentsSearchCrawlingException {

      // 動的ファセット一覧を作成
      final List<DynamicFacetInputValue> facets = new ArrayList<>();
      facets.add(new DynamicFacetInputValue(getType()));
      facets.add(createCategoryFacetValue(category));

      final Collection<List<DynamicFacetInputValue>> dynamicProperties = new
ArrayList<List<DynamicFacetInputValue>>();
      dynamicProperties.add(facets);

      // ファセット情報を保存
      try {
         dynamicPropertiesHolder.putLocalizedValues(FacetUtil.convertLocalizedValues(dynamicProperties,
Arrays.asList(new String[] { getType() })));
      } catch (final DynamicPropertiesException e) {
         throw new ContentsSearchCrawlingException("ファセット情報の保存に失敗しました。", e);
      }
   }

   /**
    * カテゴリの動的ファセットを作成します。
```

```
    * @param category カテゴリ
    * @return カテゴリの動的ファセット
    */
   private DynamicFacetInputValue createCategoryFacetValue(final String category) {
      // 動的ファセットをカテゴリで初期化
      final DynamicFacetInputValue facetValue = new DynamicFacetInputValue(category);

      // システムで利用可能な全ロケール分設定
      for (final LocaleInfo localeInfo : SystemLocale.getLocaleInfos()) {
         facetValue.addLocalizedValue(localeInfo.getLocale(), category);
      }

      return facetValue;
   }
```

動的フィールドの設定

動的フィールドの設計 にて設計した通りに標準フィールドに値を設定していきます。

動的フィールドを設定するサンプル

```
// 動的フィールドの設定
// 分類, 価格を設定
content.setValue(Fields.STRING.toField("code"), product.getCode()).
      setValue(Fields.INT.toField("price"), product.getPrice());
```

添付ファイルの設定

登録用コンテンツに対して任意のファイルを設定することができます。
登録用コンテンツに設定されたファイルは、登録処理の中でファイル内のテキストデータが抽出されます。
抽出されたテキストデータはファイル名と合わせて、全文検索の対象になります。

ここでは、**参考ファイル**にて指定されたパスにあるファイルを登録します。

添付ファイルを設定するサンプル

```
// 参照ファイルパスが存在した場合は追加
if (!StringUtil.isBlank(product.getReferenceFile())) {
   // パブリックストレージを取得
   PublicStorage storage = new PublicStorage(product.getReferenceFile());
   try {
      // ファイルだった場合のみ追加
      if (storage.isFile()) {
         content.addAttachment(new PublicStorageAttachment(storage));
      }
   } catch (IOException e) {
      // PublicStorageからファイルが取得できなかった場合
   }
}
```

権限情報の設定

登録するコンテンツに検索可能な権限を設定します。
登録用コンテンツの作成では権限を直接設定するのではなく、権限を生成するビルダーを設定します。

権限情報を設定するサンプル

```
// 認証済みユーザであれば参照可能とする権限を追加
content.addACIBuilder(new EveryoneACIBuilder());

// 特定ユーザ(青柳、円山、上田)を参照可能とする権限を追加
content.addACIBuilder(new StandardUserACIBuilder("aoyagi", "maruyama", "ueda"));

// 特定ロールの保持者を参照可能とする権限を追加
content.addACIBuilder(new StandardRoleACIBuilder("im_cs_user"));
```

利用可能な権限の一覧については、*登録用コンテンツに指定可能な権限* を参照してください。

> **注意**
>
> 権限情報が設定されていないコンテンツは検索できないため、必ずコンテンツに対して権限情報を設定してください。
> 適切な権限情報は検索対象により異なります。通常は元となるデータと同じ権限情報を設定します。

コンテンツを登録する

前項で作成した登録用コンテンツを、登録API（ `ContentsSearchManager#add(InputContent)` ）を利用して登録します。
その後 `ContentsSearchManager#commit()` を実行することで、登録したコンテンツを検索結果へ反映させています。

コンテンツを登録するサンプル

```
try {
  // マネージャのインスタンスを取得
  ContentsSearchManager manager = new ContentsSearchManager();

  // コンテンツを登録
  manager.add(content);

  // 登録内容の確定（検索結果への反映）
  manager.commit();

} catch (ContentsSearchExecutionException e) {
  // 登録、または、確定処理に失敗した場合
}
```

> **注意**
>
> IM-ContentsSearch の制限解除ライセンスが未登録の状態で登録済みのコンテンツ数が２万を超えた場合には、`ContentsSearchManager#add()`を実行したタイミングで **LicenseLimitReachedException** がスローされます。
>
> **LicenseLimitReachedException** は **ContentsSearchExecutionException** のサブクラスなため、上記サンプルの例外処理で捕捉可能です。

コンテンツを削除する

登録済みのコンテンツを削除したい場合、 `ContentsSearchManager` の `delete`系メソッドを利用します。

コンテンツを削除するサンプル

```
try {
    // マネージャのインスタンスを取得
    ContentsSearchManager manager = new ContentsSearchManager();

    // 任意の条件を指定してマッチしたコンテンツを削除（例：登録日が2ヶ月以上前のコンテンツ）
    Calendar maxDate = Env.getSystemDate();
    maxDate.add(Calendar.MONTH, -2);
    manager.delete(Condition.lessThan(Fields.RECORD_DATE, maxDate.getTime(), true));


    // TYPEフィールドを指定して削除
    manager.deleteByType("product_master");
    // 下記の条件を指定した場合と同じ
    manager.delete(Condition.type("product_master"));


    // すべてのコンテンツを削除
    manager.deleteAll();
    // 下記の条件を指定した場合と同じ
    manager.delete(Condition.all());


    // 削除内容の確定（検索結果への反映）
    manager.commit();

} catch (InvalidSearchConditionException e) {
    // 検索条件が不正だった場合
} catch (ContentsSearchExecutionException e) {
    // 削除、または、確定処理に失敗した場合
}
```

> **注意**
>
> RDBと違いロールバック処理が用意されていないため、削除したコンテンツを復元することはできません。
> `commit()` 処理は、削除したコンテンツを同じタイミングで検索結果に反映するためだけに利用します。

クローラを作成する

ここではコンテンツ登録処理をまとめて行うジョブである**クローラ**の作成方法について説明します。

次の理由から、コンテンツの作成処理はジョブによる非同期実行を推奨します。

- コンテンツの登録処理は大量のデータ件数を対象とする場合が予想される。
- 添付ファイルからテキストを抽出するにはファイルサイズやファイルの種類に応じて処理の負荷が大きくなる。
- 通常のリクエスト処理と比較して処理時間が長くなる傾向がある。

尚、作成したジョブはジョブ管理機能を利用して登録し、実行の設定を行う必要があります。
ジョブの設定について関する詳細は、「テナント管理者操作ガイド - ジョブを設定する」を参照してください。

クローリングの種類

IM-ContentsSearch ではクローリングを以下の３種類に分類しています。

1. **差分クローリング**
   新規追加、または、更新された検索対象データに対してコンテンツの作成、および、登録処理を行います。
   作成するには、何らかの方法で「新規追加された、または、更新された」という情報が取得可能である必要があります。

2. **削除クローリング**
   登録済みのコンテンツを必要に応じて削除します。
   検索結果として表示すべきでは無くなった過去のコンテンツを削除したり、一律ですべてのコンテンツ（通常はTYPEごと）を削除します。

3. **再作成クローリング**
   既存で登録されたコンテンツをすべて破棄し、すべてのコンテンツを再度作成します。
   通常は**削除クローリング処理**と**差分クローリング処理**を続けて呼ぶことで代替できますが、検索対象のデータ保持方法により実装は異なります。

IM-ContentsSearch には上記のクローリングジョブを登録するためのジョブネットが用意されています。
クローリング用のジョブネットに関する詳細は、「ジョブ・ジョブネット リファレンス - IM-ContentsSearch クローラ」を参照してください。

基底クラスを利用した実装

ここでは IM-ContentsSearch にて用意しているクローラジョブの基底クラス `BaseCrawlingJob` を利用した実装について説明します。

尚、 `BaseCrawlingJob` では以下の機能が提供されます。

- 削除クローリング(`executeDelete`)、再作成クローリング(`executeReindex`)の標準実装

  上記２つのメソッドは、必要に応じてオーバーライドしてください。

- 各クローリングメソッドで利用可能な `UpdateService` インスタンスの提供

  `UpdateService` は、 `ContentsSearchManager` の内部でも利用している、更新処理専用の機能を提供するAPIです。

- ジョブの実行パラメータによる、差分クローリング / 削除クローリング / 再作成クローリング の動作切替

  実行パラメータ **crawlingType** の値によって動作が切り替わるため、一つのジョブで３つのクローラを実装できます。

- ジョブの実行パラメータによる、コミット処理、および、最適化処理の実行指定

  実行パラメータ **withCommit** によってコミット処理実行を、 **withOptimize** によって最適化処理実行を制御可能です。

実装例

`BaseCrawlingJob` を利用したジョブのサンプル実装になります。処理対象の判別には更新日時を利用しています。

また、更新日時の検索条件として利用するために**クローラの最終実行日時**を扱うメソッドを使用しています。
クローラの最終実行日時と各メソッドの詳細については次項にて説明します。

クローラのサンプル

```
public class ProductMasterCrawlingJob extends BaseCrawlingJob {

  /**
   * コンテンツのタイプを取得（実装必須）
   * @return タイプ
   */
  @Override
  protected String getType() {
    return "product_master";
  }


  /**
```

```
    * 差分クローリング処理（実装必須）
    * @param  updateService ジョブが初期化した更新用サービス
    * @throws JobExecuteException クローリングに失敗した場合
    */
    @Override
    protected void executeDelta(UpdateService updateService) throws JobExecuteException {

        // クローラの最終実行日の取得（後述）
        Date lastCrawlingDate = getLastCrawlingDate();

        // クローラ実行日時を保存
        Date crawlingDate = Env.getSystemDate().getTime();

        // 作成した登録用コンテンツを保存
        List<InputContent> contents = new ArrayList<>();

        // 実行日時以降に更新された製品情報をRDBから取得
        try (
                Connection conn = new TenantDatabase().getConnection();
                PreparedStatement stmt = conn.prepareStatement("SELECT * FROM sample_product_master WHERE
record_date >= ?");
            ) {
            // パラメータの設定
            stmt.setTimestamp(1, new Timestamp(lastCrawlingDate.getTime()));

            // RDBから値を取得
            ResultSet result = stmt.executeQuery();

            while (result.next()) {
                contents.add(/* ResultSetからコンテンツを作成する処理 */);
            }
        } catch (SQLException e) {
            throw new JobExecuteException(e.getMessage(), e);
        }

        // コンテンツを登録
        try {
            updateService.add(contents);
        } catch (ContentsSearchExecutionException e) {
            throw new JobExecuteException(e.getMessage(), e);
        }

        // クローラの最終実行日を更新（後述）
        updateLastCrawlingDate(crawlingDate);
    }


    /**
    * 削除クローリング処理
    * @param  updateService ジョブが初期化した更新用サービス
    * @throws JobExecuteException クローリングに失敗した場合
    */
    @Override
    protected void executeDelete(UpdateService updateService) throws JobExecuteException {

        // 削除クローリング（基底実装の呼び出し）
        super.executeDelete(updateService);

        // クローラの最終実行日時をクリア（後述）
        clearLastCrawlingDate();
    }


    /**
```

```
    /
     * 再作成クローリング処理（基底実装と同じです）
     * @param  updateService ジョブが初期化した更新用サービス
     * @throws JobExecuteException クローリングに失敗した場合
     */
    @Override
    protected void executeReindex(UpdateService updateService) throws JobExecuteException {

        // 削除クローリング処理（基底実装の呼び出し）
        executeDelete(updateService);

        // 差分クローリング処理（基底実装の呼び出し）
        executeDelta(updateService);
    }

}
```

最終実行日時の保存方法

前項の**基底クラスを利用した実装**にて利用していた、**クローラの最終実行日時**に関するメソッドについて説明します。
クローラのサンプル実装において以下のメソッドを利用しています。

- `updateLastCrawlingDate(Date)`：クローラの最終実行日時を保存します
- `getLastCrawlingDate()`：クローラの最終実行日時を取得します
- `clearLastCrawlingDate()`：クローラの最終実行日時をクリアします

下記サンプルでは IM-ContentsSearch にて提供している、ストレージ上に永続化するAPI（`LastCrawlingDateHolder`）を利用しています。
また、 `BaseCrawlingJob` にて定義されている `getType()` メソッドも利用しています。

最終実行日時処理のサンプル

```
public class ProductMasterCrawlingJob extends BaseCrawlingJob {

  // ... 省略 ...

  /**
   * クローラの最終実行日時を取得します。
   * @return String クローラの最終実行日時
   * @throws JobExecuteException 取得に失敗した場合
   */
  protected Date getLastCrawlingDate() throws JobExecuteException {
    // ホルダーを取得
    LastCrawlingDateHolder crawlingDateHolder = LastCrawlingDateHolder.getHolder(getType());

    try {
      // 最終実行日時を取得
      return crawlingDateHolder.getLastCrawlingDate();
    } catch (ContentsSearchException e) {
      throw new JobExecuteException(e.getMessage(), e);
    }
  }

  /**
   * クローラの最終実行日時を更新します。
   * @param updateDate 最終実行日時
   * @throws JobExecuteException 更新に失敗した場合
   */
  protected void updateLastCrawlingDate(Date updateDate) throws JobExecuteException {
    // ホルダーを取得
    LastCrawlingDateHolder crawlingDateHolder = LastCrawlingDateHolder.getHolder(getType());

    try {
      // 最終実行日時を更新
      crawlingDateHolder.updateLastCrawlingDate(updateDate);
    } catch (ContentsSearchException e) {
      throw new JobExecuteException(e.getMessage(), e);
    }
  }

  /**
   * クローラの最終実行日時をクリアします。
   * @throws JobExecuteException クリアに失敗した場合
   */
  protected void clearLastCrawlingDate() throws JobExecuteException {
    // 最終実行日時を更新
    updateLastCrawlingDate(new Date(0L));
  }

}
```

標準の全文検索画面の拡張

この章では、標準の全文検索画面を拡張して任意の画面を検索結果画面へ表示する方法について説明します。

項目

はじめに

標準の全文検索結果画面にて拡張可能な項目は以下の通りです。

独自に定義したTYPEによる絞り込み条件の追加

- 検索時の絞り込み条件
- 検索結果の絞り込み条件一覧

TYPEごとの独自の検索結果テンプレートを用いた検索結果画面のカスタマイズ

- 画面のデザイン
- 表示する情報
- JavaScript（リンク押下時のアクションなど）

拡張を行う場合は、設定ファイルの作成が必要となります。

設定ファイルを作成する

作成が必要な設定ファイルは以下の通りです。

- テンプレート設定ファイル

  TYPEに対応する表示名や、実装したテンプレートの配置場所などを指定します。

- 表示用プロパティファイル

  テンプレート設定ファイルでは、画面に表示する名称はプロパティのキーで設定します。
  そのため、テンプレート設定ファイルに設定したキーに対応するプロパティファイルの値が必要になります。

テンプレート設定ファイルの詳細

検索結果のTYPEに対応するテンプレート情報を設定します。

テンプレート設定ファイルのサンプル

```
<?xml version="1.0" encoding="UTF-8"?>
<contentssearch-template-config>
 <template-pages>
  <!-- TYPEの指定 -->
  <template-page type="product_master" sort-key="20">

   <!-- TYPEの画面表示用プロパティキー -->
   <type-display-key>CAP.SAMPLE.PRODUCT.MASTER.CONTENTSSEARCH.CONTENTS.TYPE</type-display-key>

   <!-- 任意の検索結果画面 -->
   <template-path>im_contents_search/template/sample_product_master/product_master.jssp</template-path>

   <!-- テンプレートに渡す動的フィールド -->
   <require-dynamic-fields>
    <field type="string">code</field>
    <field type="int">price</field>
   </require-dynamic-fields>

  </template-page>

  <!-- "product_master"の子のTYPEを指定 -->
  <template-page type="Base" sort-key="1">

   <!-- 親のTYPEを指定 -->
   <parent-type>product_master</parent-type>

   <type-display-key>CAP.SAMPLE.PRODUCT.MASTER.CONTENTSSEARCH.CONTENTS.TYPE.BASE</type-display-key>
   <template-path>im_contents_search/template/sample_product_master/product_master.jssp</template-path>
   <require-dynamic-fields>
    <field type="string">code</field>
    <field type="int">price</field>
   </require-dynamic-fields>

  </template-page>

  <!-- 省略（Product, eBuilder） -->

 </template-pages>
</contentssearch-template-config>
```

それぞれの設定値について説明します。

No.	タグ (: 属性)	必須	説明
1	template-pages	○	テンプレートページ設定のルートタグです。
2	template-page	○	テンプレートページ設定を記述します。TYPEごとに指定する必要があります。
3	template-page : type	○	設定を適用するTYPEを指定します。
4	template-page : sort-key		設定したTYPEを一覧に表示する際のソート順を数値で指定します。 小さな値ほど上に表示され、省略した場合は最下部に表示されます。
5	type-display-key	○	TYPEを画面に表示する名称のプロパティキーを指定します。
6	template-path		テンプレート画面の実装があるパスを指定します。拡張子には **.jssp** を指定してください。 省略した場合は標準テンプレートが利用されます。

No.	タグ (: 属性)	必須	説明
7	required-dynamic-fields		テンプレート画面の実行パラメータに渡す検索結果コンテンツの動的フィールドを指定します。
8	field	○	動的フィールドの名称を指定します。
9	field : type	○	動的フィールドの型を指定します。

表示用プロパティファイルの詳細

表示用プロパティファイルに画面表示用の名称を設定します。
専用のプロパティファイルを用意しなくても、任意のプロパティファイルにキーを追加することで設定可能です。

表示用プロパティファイルのサンプル：日本語、英語、中国語（中華人民共和国）

```
# conf/message/path/to/sample/sample_message_ja.properties
CAP.SAMPLE.PRODUCT.MASTER.CONTENTSSEARCH.CONTENTS.TYPE=製品情報
```

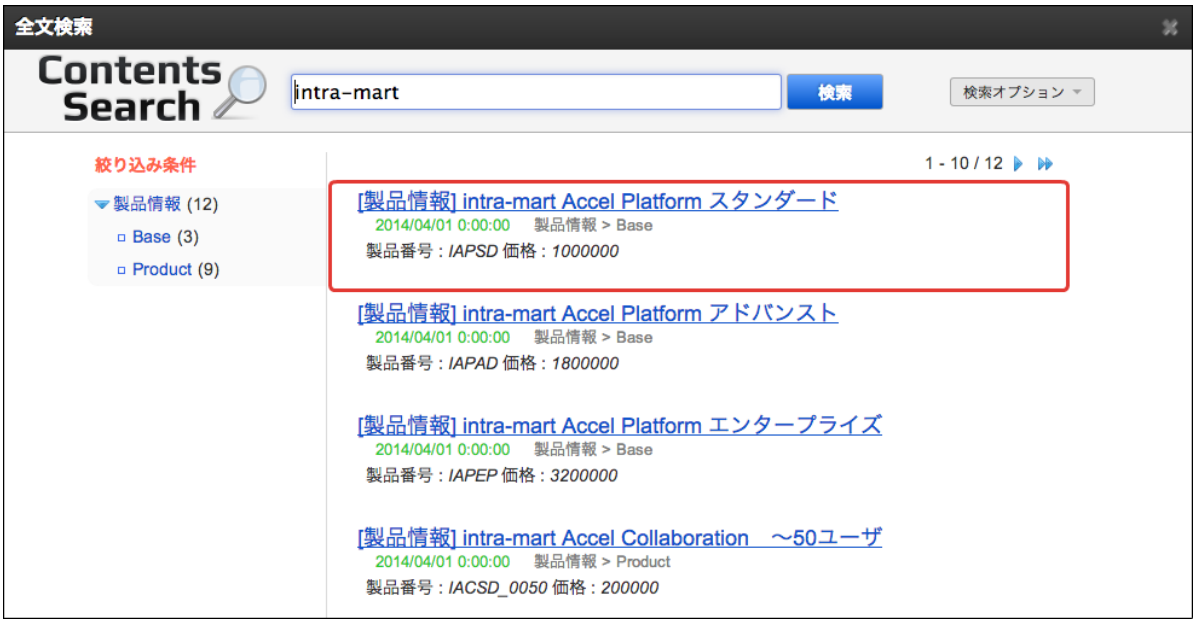
```
# conf/message/path/to/sample/sample_message_en.properties
CAP.SAMPLE.PRODUCT.MASTER.CONTENTSSEARCH.CONTENTS.TYPE=Product Information
```

```
# conf/message/path/to/sample/sample_message_zh_CN.properties
CAP.SAMPLE.PRODUCT.MASTER.CONTENTSSEARCH.CONTENTS.TYPE=□品信息
```

独自の検索結果テンプレートを実装する

独自の検索結果テンプレートを実装する方法について説明します。
検索結果テンプレートの実装は、 スクリプト開発モデル を利用します。

検索結果画面における赤枠部分が 1 つの検索結果テンプレートが表示する範囲になります。



尚、独自の検索結果テンプレートが設定されていない場合には標準で用意されているテンプレートが利用されます。
標準で用意されている検索結果テンプレートは簡素なため、独自のテンプレートを用意することを推奨します。

標準テンプレートの検索結果サンプル

検索結果テンプレートの実装

独自の検索結果テンプレートの実装方法をサンプルを用いて解説します。

JavaScriptファイル

はじめに表示用データを用意する処理（JavaScriptファイル）を作成します。
テンプレートでは `init` 関数に、引数 `content` が与えられます。
通常の スクリプト開発モデル における実装と違い、リクエストパラメータは格納されません。

`content` には以下のパラメータが設定されています。

- 標準フィールド
  id, type, url, id_original, title, text, attachment, record_date
- テンプレート設定ファイルで指定された動的フィールド
  サンプルの場合：code, price
- スニペット（検索文字にマッチした文字列のハイライト）
  snippets
- TYPEのパンくずリスト
  typeBreadcrumbs

以下は、引数に与えられた `content` を用いて、画面表示用の `$data` を作成するサンプルになります。

JavaScriptファイルの実装サンプル

```
var $data = {};
var $format = {};

function init(content) {

 // 日時のフォーマットIDを設定
 $format.datetimeFormatId = ["IM_DATETIME_FORMAT_DATE_STANDARD",
"IM_DATETIME_FORMAT_TIME_TIMESTAMP"];

 // 標準フィールドデータ格納
 // ID
 $data.id = content.id;
 // タイトルの接頭詞
 var titlePrefix =
MessageManager.getMessage("CAP.SAMPLE.PRODUCT.MASTER.CONTENTSSEARCH.TITLE.PREFIX");
 // タイトル
 $data.title = titlePrefix + " " + content.title;

 // 更新日時
 $data.recordDate = content.record_date;

 // 詳細画面URL
 var idOriginal = content.id_original;
 $data.url = content.url + "/" + idOriginal;

 // 動的フィールドデータ格納
 // 製品番号
 $data.code = content.code;
 // 価格
 $data.price = content.price;

 // 表示用データ格納
 // スニペット
 $data.snippets = content.snippets;

 // パンくずリスト
 $data.breadcrumbs = content.typeBreadcrumbs;

}
```

HTMLファイル

次に画面（HTMLファイル）を作成します。
JavaScriptファイルで作成した `$data` を利用して、画面を作成します。

検索結果テンプレート画面は全文検索結果画面のdiv要素として埋め込まれるため、divタグから開始する必要があります。
また、標準のデザインと合わせる事ができるよう、独自のCSSを定義しています。

以下のサンプルを参考に、画面を作成してください。

画面の実装サンプル

```
<!-- div要素から開始します -->
<div>

 <!-- タイトル（リンククリックで詳細画面を表示） -->
 <h3 class="imcs-content-detail-title">
  <a target="_blank" id="<imart type="string" value=$data.id escapeXml="true" escapeJs="true" />"
href="javascript:void(0);">
    <imart type="string" value=$data.title escapeXml="true" escapeJs="true" />
  </a>
 </h3>
```

```
  <!-- サブタイトル（RECOED_DATE および TYPE） -->
  <div class="imcs-content-detail-subtitle">
    <span class="imcs-content-detail-subtitle-date">
      <imart type="imuiAccountDateTimeFormatter" value=$data.recordDate formatId=$format.datetimeFormatId
/>
    </span>
    <span>
      <imart type="string" value=$data.breadcrumbs escapeXml="true" escapeJs="true" />
    </span>
  </div>

  <div>
    <!-- 任意のデータ（動的フィールドなど）の表示 -->
    <div class="imcs-content-detail-option">
      <div class="imcs-content-detail-option-row">
        <div class="imcs-iac-infomation-content-detail-option-cell" >
          <span class="imcs-content-detail-option-label">
            <imart type="message" id="CAP.SAMPLE.PRODUCT.MASTER.CONTENTSSEARCH.CODE" escapeXml="true"
escapeJs="false" />
          </span>
          <span class="imcs-content-detail-option-value">
            <imart type="string" value=$data.code escapeXml="true" escapeJs="false" />
          </span>
          <span class="imcs-content-detail-option-label imcs-iac-infomation-content-detail-option-label">
            <imart type="message" id="CAP.SAMPLE.PRODUCT.MASTER.CONTENTSSEARCH.PRICE" escapeXml="true"
escapeJs="false" />
          </span>
          <span class="imcs-content-detail-option-value">
            <imart type="string" value=$data.price escapeXml="true" escapeJs="false" />
          </span>
        </div>
      </div>
    </div>

    <!-- スニペットの表示 -->
    <div class="imcs-content-detail-snippets">
      <imart type="repeat" list=$data.snippets item="snippet">
        <imart type="string" value=snippet escapeXml="false" escapeJs="false" />
      </imart>
    </div>
  </div>

  <!-- 詳細画面をポップアップで開くためのJavaScript -->
  <script type="text/javascript">
      jQuery(function($) {
        var target = "PRODUCT_INFO_DETAIL";
        $("#<imart type="string" value=$data.id />").click(function() {
            window.open("", target, "status=yes,resizable=yes,scrollbars=yes,height=600,width=998");
            $("<form/>", {
              "action" : "<imart type="string" value=$data.url escapeXml="false" />",
              "method" : "post",
              "target" : target
            }).appendTo('body').submit();
            return false;
        });
      });
    </script>
</div>
```

作成したファイルの配置

作成した JavaScriptファイル と HTMLファイル を配置します。

ファイルの配置場所は、テンプレート設定ファイルの **template-path** に指定したパスと合わせる必要があります。

ファイル配置場所の例

- JavaScriptファイル - `{project_path}/src/main/jssp/src/im_contents_search/template/sample_product_master/product_master.js`
- HTMLファイル - `{project_path}/src/main/jssp/src/im_contents_search/template/sample_product_master/product_master.html`

付録

注意事項

実装時や環境構築時、運用時における注意事項について記載します。

- **コンテンツの登録処理では、添付ファイルの種類およびサイズが処理時間に大きく影響します。**

  添付ファイルからテキストを抽出する処理は負荷が高く、またファイルの種類によっても負荷の増減があります。

  例えば、単純なテキストファイルから抽出する処理の負荷は比較的軽くなりますが、複雑な計算式と大量データを持ったExcelファイルなどからテキストを抽出する場合には、抽出処理にて式の計算を行う必要があるため非常に高負荷になる傾向があります。
  また、画像などが含まれていてファイルサイズが大きなPDFやWordファイルなどを処理する場合には、ファイルの転送処理や展開処理において高い負荷が発生する傾向があります。

- **大量のテキストデータを設定したコンテンツにおいて、一定サイズ以上のテキストが検索対象にならない場合があります。**

  Solrでは文字列の最大サイズに制限があるために発生します。Solrの設定を変更することで回避可能です。
  特に、大量のテキストデータを持った添付ファイルを設定したコンテンツで発生する可能性が高くなります。

- **登録済みのコンテンツ数が増えるほど、最適化処理の実行時間が増加します。**

  Solrにおける最適化処理では、一時的に登録されている全データを複製します。
  そのため、登録済みのコンテンツ数に比例して最適化処理の実行時間が増加します。

  標準では、最適化処理専用のジョブをクローラジョブネットの最後に実行するように設定されています。
  各クローラジョブ内で最適化処理は行わずに、最適化ジョブを利用することで最適化処理の実行回数を減らしてください。

- **可能な限り複数のクローラジョブを同時実行しないでください。**

  登録処理が並列で実行されると、Solrサーバのディスク I/Oの負荷が高くなります。
  また分散トランザクションに対応していないため、意図しないタイミングで検索結果がユーザに公開される可能性があります。

各種一覧

IM-ContentsSearch を利用したプログラミングにおいて参考となる情報の一覧です。

製品にて利用しているTYPE

当社製品において利用されているTYPEフィールドの値（親階層のみ）の一覧を記載します。
TYPEフィールドは登録用コンテンツの標準フィールドに設定されます。
詳細は *標準フィールドの設計* を参照してください。

No.	機能・製品名	TYPE	ソート
1	IM-Workflow	workflow	1
2	IMBox	imbox	3
3	intra-mart Accel Collaboration	iac	4
4	ドキュメントワークフロー (BPW)	bpw	6
5	intra-mart Accel Documents	acceldocuments	1
6	intra-mart Accel Archiver	wdc	1
7	intra-mart Accel GroupMail	iag	20
8	IM-Knowledge	imkb	7

注意

上記の一覧に存在しないTYPEについても、今後製品が追加された場合に追加される可能性があります。

登録用コンテンツに指定可能な権限

登録用コンテンツに対して指定可能な権限（ビルダー）の一覧を記載します。
各ビルダーの詳細については、クラス名のリンクからAPIドキュメントを参照してください。

No.	権限	クラス名
1	未認証ユーザ	AnonymousACIBuilder
2	認証ユーザ	EveryoneACIBuilder
3	ユーザ	StandardUserACIBuilder
4	ロール	StandardRoleACIBuilder
5	パブリックグループ	StandardPublicGroupACIBuilder
6	パブリックグループ役割	StandardPublicGroupRoleACIBuilder
7	会社	StandardCompanyACIBuilder
8	組織	StandardDepartmentACIBuilder
9	役職	StandardPostACIBuilder