IBM Security Directory Integrator
バージョン 7.2.0.1

ユーザーズ・ガイド

SC88-8416-03
(英文原典：SC27-2706-03)

IBM Security Directory Integrator
バージョン 7.2.0.1

ユーザーズ・ガイド

SC88-8416-03
(英文原典：SC27-2706-03)

お願い

本書および本書で紹介する製品をご使用になる前に、 255 ページの『特記事項』に記載されている情報をお読みください。

注: 本書は、*IBM Security Directory Integrator* バージョン 7.2.0.1 ライセンス・プログラム (5724-K74)、および新しい版で明記されていない限り、以降のすべてのリリースおよびモディフィケーションに適用されます。

お客様の環境によっては、資料中の円記号がバックスラッシュと表示されたり、バックスラッシュが円記号と表示されたりする場合があります。

原典： SC27-2706-03
IBM Security Directory Integrator
Version 7.2.0.1
Users Guide

発行： 日本アイ・ビー・エム株式会社

担当： トランスレーション・サービス・センター

目次

本書について

この資料には、IBM® Security Directory Integrator に含まれるコンポーネントを使用したソリューション開発に必要な情報が記載されています。

IBM Security Directory Integrator の各コンポーネントは、ユーザー・ディレクトリーおよびその他のリソースの管理を担当するネットワーク管理者用に設計されています。IBM Security Directory Integrator と IBM Security Directory Server の両方のインストールおよび使用に関する実務経験を持っていることを想定しています。

本書は、IBM Security Directory Integrator を使用したソリューションの開発、インストール、および管理を担当するユーザーも対象としています。読者は、開発したソリューションの接続先となるシステムの概念や管理方法について習熟している必要があります。そのようなシステムには、ソリューションに応じて、以下の製品、システム、概念のうち 1 つ以上が含まれます (これらに限定されてはいません)。

- IBM Security Directory Server
- IBM Security Identity Manager
- IBM Java™ ランタイム環境 (JRE) または Oracle Java ランタイム環境
- Microsoft Active Directory
- Windows および UNIX オペレーティング・システム
- セキュリティー管理
- Hypertext Transfer Protocol (HTTP)、HyperText Transfer Protocol Secure (HTTPS)、および Transmission Control Protocol/Internet Protocol (TCP/IP) を含むインターネット・プロトコル (IP)
- Lightweight Directory Access Protocol (LDAP) およびディレクトリー・サービス
- サポートされるユーザー・レジストリー
- 認証と許可の概念
- SAP ABAP アプリケーション・サーバー

資料および用語集へのアクセス

オンラインでアクセス可能な IBM Security Directory Integrator バージョン 7.2.0.1 ライブラリーおよび関連資料の説明をお読みください。

このセクションには、以下が含まれています。

- 『IBM Security Directory Integrator ライブラリー』の資料のリスト。
- ix ページの『オンライン資料』へのリンク。
- ix ページの『IBM Terminology Web サイト』へのリンク

IBM Security Directory Integrator ライブラリー

IBM Security Directory Integrator ライブラリーでは、以下の資料を入手できます。

- *IBM Security Directory Integrator バージョン 7.2.0.1 Federated Directory Server 管理ガイド*

  Federated Directory Server コンソールを使用したデータ統合ソリューションの設計、実装、および管理について説明しています。System for Cross-Domain Identity Management (SCIM) プロトコルおよびインターフェースを使用した ID 管理についても説明しています。

- *IBM Security Directory Integrator バージョン 7.2.0.1 スタートアップ・ガイド*

  IBM Security Directory Integrator の簡単なチュートリアルおよび概要が記載されています。対話の作成の例と、IBM Security Directory Integrator の実践学習を含んでいます。

- *IBM Security Directory Integrator バージョン 7.2.0.1 ユーザーズ・ガイド*

  IBM Security Directory Integrator の使用法についての情報が記載されています。Security Directory Integrator 設計ツール (構成エディター) を使用したソリューション設計、またはコマンド行からの既成ソリューションの実行に関する説明が記載されています。また、インターフェース、概念、および AssemblyLine の作成に関する情報も記載されています。

- *IBM Security Directory Integrator バージョン 7.2.0.1 インストールおよび管理者ガイド*

  インストール、旧バージョンからのマイグレーション、ロギング機能の構成、および IBM Security Directory Integrator のリモート・サーバー API の基礎となるセキュリティー・モデルに関する詳細情報が記載されています。ソリューションのデプロイおよび管理の方法についての情報が記載されています。

- *IBM Security Directory Integrator バージョン 7.2.0.1 リファレンス・ガイド*

  IBM Security Directory Integrator の個々のコンポーネント (コネクター、関数コンポーネント、パーサー、オブジェクトなど、AssemblyLine のビルディング・ブロック) に関する詳細情報が記載されています。

- *IBM Security Directory Integrator バージョン 7.2.0.1 Problem Determination Guide*

  問題の識別および解決に役立つ IBM Security Directory Integrator ツール、リソース、および技法について説明されています。

- *IBM Security Directory Integrator バージョン 7.2.0.1 Message Guide*

  IBM Security Directory Integrator に関連するすべての通知メッセージ、警告メッセージ、およびエラー・メッセージのリストが記載されています。

- *IBM Security Directory Integrator バージョン 7.2.0.1 Password Synchronization Plug-ins ガイド*

  5 つの IBM Password Synchronization Plug-ins (Windows 用 Password Synchronizer、Sun Directory Server 用 Password Synchronizer、IBM Security Directory Server 用 Password Synchronizer、IBM Domino 用 Password Synchronizer、UNIX および Linux 用 Password Synchronizer) それぞれのインストールおよび構成について詳細に説明されています。また、LDAP パスワード・ストアと JMS パスワード・ストアの構成手順についても説明します。

- *IBM Security Directory Integrator バージョン 7.2.0.1 リリース情報*

資料に記載されていない IBM Security Directory Integrator の新機能および最新情報を記載しています。

オンライン資料

IBM では、製品のリリース時および資料の更新時に、以下の場所に製品資料を掲載しています。

IBM Security Directory Integrator ライブラリー
製品資料サイト (http://www-01.ibm.com/support/knowledgecenter/SSCQGF/welcome) には、ライブラリーのウェルカム・ページとナビゲーションが表示されます。

IBM Security Systems Documentation Central
IBM Security Systems Documentation Central には、すべての IBM Security Systems 製品ライブラリーのアルファベット順のリストと、各製品の特定バージョンのオンライン資料へのリンクが掲載されています。

IBM Publications Center
IBM Publications Center サイト (http://www-05.ibm.com/e-business/linkweb/publications/servlet/pbi.wss) には、必要なすべての IBM 資料を見つけるのに役立つカスタマイズ検索機能が用意されています。

関連情報

IBM Security Directory Integrator の関連情報は以下の場所で入手できます。

- IBM Security Directory Integrator では、Oracle の JNDI クライアントを使用しています。JNDI クライアントについては、「*Java Naming and Directory Interface™ Specification*」(http://download.oracle.com/javase/7/docs/technotes/guides/jndi/index.html) を参照してください。
- IBM Security Directory Integrator に関する疑問点を解決するために有用な情報が https://www-947.ibm.com/support/entry/myportal/over-accesspubsview/software/security_systems/tivoli_directory_integrator に記載されています。

IBM Terminology Web サイト

IBM Terminology Web サイトは、製品ライブラリーの用語を 1 つのロケーションに統合したものです。Terminology Web サイトには、http://www.ibm.com/software/globalization/terminology からアクセスできます。

アクセシビリティー

アクセシビリティー機能は、運動障害または視覚障害など身体に障害を持つユーザーがソフトウェア・プロダクトを快適に使用できるようにサポートします。この製品では、インターフェースの読み上げおよびナビゲートを行うための支援技術を使用できます。また、マウスの代わりにキーボードを使用して、グラフィカル・ユーザー・インターフェースのすべての機能を操作できます。

詳しくは、「*Directory Integrator の構成*」のアクセシビリティーに関する付録を参照してください。

技術研修

以下は英語のみの対応となります。技術研修の情報については、IBM Education Web サイト (http://www.ibm.com/software/tivoli/education) を参照してください。

サポート情報

IBM サポートは、コード関連の問題、およびインストールまたは使用方法に関する短時間の定型質問に対する支援を提供します。IBM ソフトウェア・サポート・サイトには、http://www.ibm.com/software/support/probsub.html から直接アクセスできます。

トラブルシューティング では、以下が詳細に説明されています。

- IBM サポートに連絡する前に収集する情報。
- IBM サポートに連絡するためのさまざまな方法。
- IBM Support Assistant の使用方法。
- 自分で問題を切り分け、修正するための説明および問題判別の資料。

適切なセキュリティーの実践に関する注意事項

IT システムのセキュリティーでは、企業内および企業外からの不適切なアクセスの防止、検出、およびそれらのアクセスへの対応により、システムおよび情報を保護する必要があります。不適切なアクセスにより、情報が改ざん、破壊、盗用、または悪用されたり、ご使用のシステムの損傷または他のシステムへの攻撃のための利用を含む悪用につながる可能性があります。完全に安全と見なすことができる IT システムまたは IT 製品は存在せず、また単一の製品、サービス、またはセキュリティー対策が、不適切な使用またはアクセスを防止する上で、完全に有効となることもありません。 IBM のシステム、製品およびサービスは、包括的なセキュリティーの取り組みの一部となるように設計されており、これらには必ず追加の運用手順が伴います。また、最高の効果を得るために、他のシステム、製品、またはサービスを必要とする場合があります。IBM は、システム、製品、またはサービスが、悪意のある行為または不正な行為から影響を受けないこと、またはこれらの行為がお客様の企業に影響を与えないことを保証しません。

第 1 章 一般概念

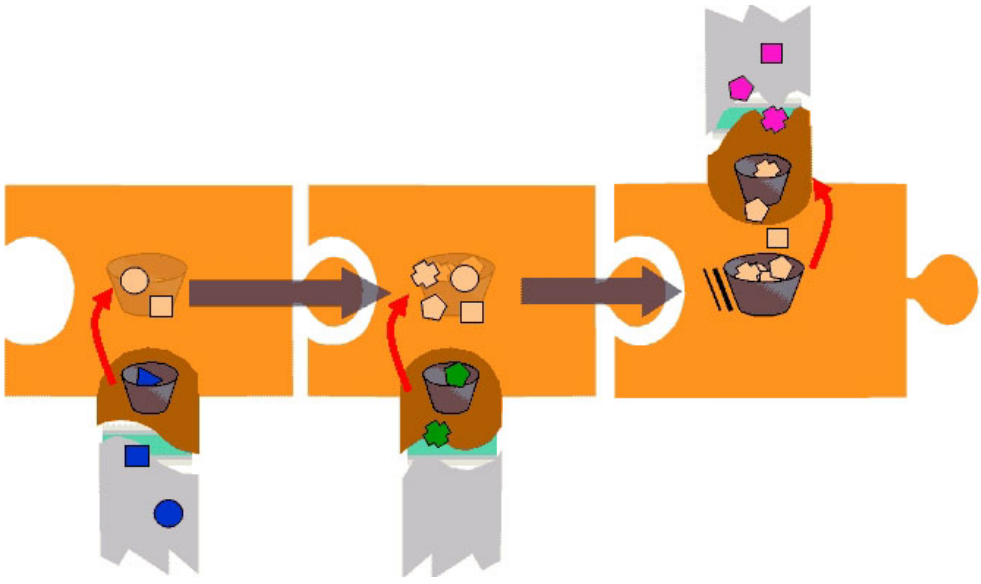
このセクションでは、IBM Security Directory Integrator の基本概念の一部と共に、ソリューションの構築に役立つアーキテクチャーの各種エレメントや、それらの特徴および動作について説明します。

AssemblyLine

AssemblyLine (AL) は、データを移動および変換するために組み合わされた一連のコンポーネントであり、データが通過する「ルート」を記述します。このルートで処理されるデータは*項目* オブジェクトとして表現されます。AssemblyLine は、AssemblyLine の各サイクルで一度に 1 つの項目を処理します。これは IBM Security Directory Integrator における作業単位であり、一般に 1 つ以上のデータ・ソースから 1 つ以上のターゲットへの情報のフローを表します。

概説

AssemblyLine を構成するコンポーネントの一部は、1 つ以上の*接続*システム からデータを取得します。このように取得されたデータは、AL への「フィード」と呼ばれます。処理されるデータは 1 回に 1 つの*項目* ずつ AL に送られます。これらの項目は、ディレクトリー項目、データベース行、E メール、*IBM Notes* 文書、レコードまたは類似データ・オブジェクトからの値とともに、*属性* を伝送します。各項目は、ソース・システム内のフィールドまたは列から読み取ったデータ値を保持する*属性* を伝送します。これらの属性は、AL 内のあるコンポーネントから次のコンポーネントへのフローの処理により、名前変更、再フォーマット、または計算されます。他のソースからの新規情報を「結合」できます。変換後のデータのすべてまたは一部は、目的のターゲット・ストアに書き込むことも、ターゲット・システムに送信することもできます。これを次の図に示します。



このダイアグラムに示す大きなジグソー・パズルのピースの集合が AssemblyLine であり、灰色のストリームの左端にある青色の円形と正方形が、下から入力ストリームのロー・データとして入力され、右上の紫色のパーツが出力ストリームのデータ出力となります。暗いオレンジ色のエレメント (中にバケツが示されている) が、ジグソー・パズル・ピースと一部重なっています。これはパーサー を示します。パーサーはロー・データを構造化データ (バケツの中の明るい色のエレメント) に変換します。変換された構造化データは AssemblyLine 内を進むことができます。中央のジグソー・パズル・ピースは、データベースなどから既に構造化されているデータを読み取るコネクターを示します。

データは何らかの入力モード の 4 ページの『コネクター』を使用して接続システムから AssemblyLine に入力され、その後何らかの出力モードのコネクターを使用して 1 つ以上の接続システムに出力されます。

データをレコード指向システム (データベースやメッセージ・キューなど) から読み取ることができます。この場合、入力の各種列が結果の作業項目 (左のパズル・ピースに「バケツ」として示されている) の属性に適切にマップされます。あるいは、データをデータ・ストリーム (ファイル・システムのテキスト・ファイル、ネットワーク接続など) から読み取ることもできます。この場合、入力ストリームを分析して認識できるようにするため、コネクターの前にパーサーを追加し、入力ストリームを分割して作業項目の属性に割り当てることができます。

1 番目のコネクターの処理が完了すると、情報のバケツ (「作業項目」つまり*作業*) が AssemblyLine に沿って次のコンポーネント (この図では別のコネクター) に渡されます。1 番目のコネクターから出力されたデータが使用可能であるため、2 番目の接続システムのデータの検索、つまりルックアップのためのキー情報として、このデータを使用できます。検出された関連データを*作業* にマージできます。これにより、1 番目のコネクターからのデータが完全なものになります。

最後に、マージされたデータが AssemblyLine に沿って 3 番目のパズル・ピース (何らかの出力モードのコネクター) に渡されます。このコンポーネントは、接続システムへのデータ出力を処理します。接続システムがレコード指向の場合、*作業* の各種属性がレコードの列にマップされます。接続システムがストリーム指向の場合、パーサーが必要なフォーマット処理を実行できます。

作業 のデータに対する操作を実行するため、その他のコンポーネント (24 ページの『スクリプト・コンポーネント』や 23 ページの『関数』など) を任意で AssemblyLine に挿入できます。

AssemblyLine は一度に 1 つの項目のみを処理するように設計および最適化されている点に留意することが重要です。しかし、複数の更新や削除を行う必要がある場合 (同時に複数の項目を処理する場合など) は、そのような処理を実行できるように AssemblyLine のスクリプトを記述しなければなりません。必要に応じて、JavaScript、Java ライブラリー、および IBM Security Directory Integrator の標準機能を使用して、そのような処理をインプリメントすることができます。例えば、JDBC コネクターなどを使用してデータをソート済みのデータ・ストアにプールした後、そのデータを読み戻して別の AssemblyLine で処理することができます。

AssemblyLine の作成、構成、およびテストには IBM Security Directory Integrator 構成エディター (CE) を使用します。詳しくは 85 ページの『第 3 章 構成エディ

ター』を参照してください。構成エディターでは、AssemblyLine に「データ・フロー」タブがあります。ここに、AL を構成するコンポーネントのリストが保持されます。

AL のすべてのコンポーネントは、スクリプト変数として自動的に登録されます。そのため、ReadHRdump というコネクターを保持している場合、*ReadHRdump* 変数を使用してスクリプトからコネクターとそのメソッドに直接アクセスできます。つまり、スクリプト変数を命名するように AL コンポーネントを命名する必要があります。名前には英数字のみを使用し、先頭の文字に数字は使用しないでください。また、特殊な国別文字 (å、ä など)、分離文字 (下線「_」は除く)、空白文字なども使用しないでください。

AL コンポーネントにアクセスするための代替方法 (`task.getConnector()` 関数など) が常にありますが、命名規則を常に意識することが推奨されます。

IBM Security Directory Integrator で AssemblyLine を開始する際、新規 Javaスレッドを作成し、通常は 1 つ以上のデータ・ソースに対して接続を設定するため、非常に手間のかかる操作になります。 使用するソリューション設計で、処理する個別の AssemblyLine の数を可能な限り少なくできるかどうかを、慎重に検討してください。つまり、ブランチや Switch を使用して、1 つの AL で処理する複数の操作を定義するなど、それぞれの AssemblyLine がより多くの作業を処理するようにします。各操作は個別の AssemblyLine としてインプリメント可能ですが、操作を「即座に使用可能」なものとして 1 つの AL に組み込むことができる点に注意してください。この場合、AL が AL コネクターまたは AL 関数を使用して作業を操作にディスパッチします。これにより、グローバル・コネクター・プールのような機能を利用して、リソースの使用量を管理したり、パフォーマンスやスケーラビリティーを高めたりすることもできます。

コンポーネント

AssemblyLine には以下のコンポーネントを組み込むことができます。

- 4 ページの『コネクター』
- 23 ページの『関数』
- 24 ページの『スクリプト・コンポーネント』
- 25 ページの『AttributeMap』
- 29 ページの『ブランチ・コンポーネント』

また、コネクターではパーサー 32 ページの『パーサー』を構成できます。システム、構成、AssemblyLine、属性マップと属性のレベルでは、 26 ページの『NULL 動作』を構成するためのオプションがあります。

AssemblyLine 内部での AL コンポーネントへのアクセス

各 AL コンポーネントは、そのコンポーネントに対して指定した名前を使用した事前登録済みのスクリプト変数として使用できます。

system.getConnector() のような、関数へのスクリプト記述呼び出しによりコンポーネントを動的にロードできます。ただし、この方法は、経験の浅いユーザーの方にはお勧めしません。[1]

AssemblyLine でのパラメーターの引き渡し

AssemblyLine にデータを取り込む方法としては、次の 3 つがあります。

- 独自の初期項目を AssemblyLine 内で生成する (プロローグ・スクリプトなど)。
- 1 つ以上のイテレーターからフィードする[2]。
- AL コネクター、AL 関数コンポーネント、または API 呼び出しを使用して、別の AssemblyLine のパラメーターを指定して AssemblyLine を開始する。

AssemblyLine の開始時に別の AssemblyLine のパラメーターを指定する場合は、いくつかのオプションがあります。

- *タスク呼び出しブロック* (TCB) を使用する (推奨方法)。詳しくは、 72 ページの『タスク呼び出しブロック (TCB)』を参照してください。このセクションでは、AssemblyLine コンポーネントを動的に使用可能または使用不可に設定する方法についても説明しています。
- 初期作業項目を直接指定する。詳しくは、 8 ページの『初期作業項目 (IWE) の提供』を参照してください。

  注: これらのオプションは、以前のバージョンとの互換性を維持する目的で用意されています。

コネクター

コネクターは情報ソースへのアクセスと更新に使用されます。コネクターは、作業環境を整備する役割を果たします。これにより、ユーザーは、さまざまなデータ・ストア、システム、サービス、またはトランスポートを処理するための技術的な詳細を考慮する必要がなくなります。そのため、各タイプのコネクターは、特定のプロトコルまたは API を使用してデータ・ソース・アクセスの詳細を処理するように設計されています。これにより、ユーザーは、データの操作、データの関係、およびフィルター処理や整合性制御などのカスタム処理に集中することができます。

概説

コネクターは、一部のシステムまたはストアの詳細を抽象化するために使用され、同一のアクセス機能セットを提供します。これにより、非常に広範で多様なテクノロジーおよびフォーマットを一貫性のある予測可能な方法で処理できます。標準的な AssemblyLine (AL) には、入力データを提供する 1 つのコネクターと、データを書き出す 1 つ以上のコネクターが含まれます。

---

1. この呼び出しによって取得できるコネクター・オブジェクトは、コネクター・インターフェース・オブジェクトです。これは、AssemblyLine コネクターのデータ・ソースに固有の部分です。コネクターのタイプを変更すると、そのコネクターのデータ・ソース・インテリジェンス(特定のシステム、サービス、またはデータ・ストアのデータにアクセスするための機能を提供している部分で、コネクター・インターフェースとも呼ばれます) がスワップアウト(交換) されることになります。AssemblyLine コネクターの機能の大部分(属性マップ、リンク基準、フックなど) はカーネルによって提供されており、コネクターのタイプを変更してもそのまま保持されます。

2. *イテレーター* は、「イテレーター・モードのコネクター」の省略表現です。

コネクターには次の 2 つのカテゴリーがあります。

- 第 1 のカテゴリーは、トランスポートとデータ内容の構造が両方ともコネクターにとって既知である (つまり、JDBC や LDAP などの既知の API を使用してデータ・ソースのスキーマを照会または検出できる) 場合です。
- 第 2 のカテゴリーは、トランスポートのメカニズムは既知であるが、内容の構造 (通常は、データ・ストリームのような構造) が不明である場合です。このカテゴリーでは、AssemblyLine が適切に機能するように、パーサー (32 ページの『パーサー』または「リファレンス」の『パーサー』セクションを参照) が内容の構造を解釈または生成する必要があります。

充実したコネクター・ライブラリーを備えている点が、IBM Security Directory Integrator の特長の 1 つです。IBM Security Directory Integrator に組み込まれているすべてのコネクターのリストについては、「*リファレンス*」を参照してください。ただし、独自のコネクターを JavaScript や Java で作成することもできます。「*リファレンス*」の『独自のコンポーネントのインプリメント』を参照してください。

各コネクターは、特定のプロトコル、API、またはトランスポートを対象として設計されており、接続されたシステムのネイティブ・タイプと Java オブジェクトの間におけるデータのマーシャルを処理します。 他のコンポーネントとは異なり、コネクターには、接続システムへのアクセス方法を決定するモード設定があります。詳しくは、 6 ページの『コネクター・モード』を参照してください。各コネクターは、それが接続されたシステムに対して適切なモードのサブセットのみサポートします。例えば、ファイル・システム・コネクターは 1 つの出力モード (AddOnly) のみをサポートし、更新、削除、および CallReply をサポートしません。コネクターを使用するときは、最初にそのコンポーネントの資料を調べて、サポートされるモードのリストを確認する必要があります。

注: AssemblyLine を構成するコネクター (およびその他のコンポーネント) の数は、特定のデータ・フローを実現するために必要な数にすることができます。コネクターの数に関するシステム上の制限はありません。ただし、ベスト・プラクティスは、AssemblyLine をできるだけ単純にして、保守容易性を最大に高めるようにすることです。

AssemblyLine のコネクターを選択すると、ダイアログ・ボックスが表示されるため、ここで*継承元* のコネクターのタイプを選択できます。継承は、IBM Security Directory Integrator の作業で使用される重要な概念です。ソリューションに組み込まれたすべてのコンポーネントは、その特性のすべてまたは一部を、別のコンポーネントから継承します。継承元のコンポーネントは、基本タイプの 1 つであるか、またはライブラリー内の事前構成済みのコンポーネント(ワークスペースの「リソース」セクションの「コネクター」、「パーサー」、「関数」など) です。

コネクターは、AL で使用された場合、コンポーネントのセットアップ時 (接続の作成時、リソースのバインド時など) に、制御する「初期化」オプションを提供します。デフォルトでは、AssemblyLine の開始時 (*AL 開始フェーズ*) にすべてのコネクターが初期化されます。

サーバー・モードまたはイテレーター・モードのコネクターは、AL が作成する各サイクルの新規作業項目を AL にフィードする役割を担います。作業項目は、フロ

ーの終端に達するまで (インプリメントした任意のブランチ・ロジックに従って)、「フロー」セクションでコンポーネントからコンポーネントへと渡されます。この時点で、サイクルの終わりの動作が開始されます。例えば、イテレーターがソースから次の項目を取得し、新規サイクルのために「フロー」セクションに渡します。

「供給」セクションのイテレーターは実際はフローを主導しますが、「フロー」セクションのイテレーターは単に次の項目を取得し、入力マッピング用にそのデータ属性を作業項目に提供します。

注: イテレーター・モードのコネクターを「フロー」セクションに含めることができます。その場合、イテレーターは「供給」セクションの場合と同様に機能します。つまり、AL の始動時に初期化され (`selectEntries` 呼び出しによる結果セットの作成を含む)、AL の各サイクルで 1 つの項目を取得します (`getNextEntry`)。ただし、「フロー」セクション内のイテレーターは、「供給」セクションの場合と異なり、AL 自体を駆動することはありません。

「供給」セクションの動作は、サーバー・モードとイテレーター・モードで異なります。イテレーターは AL 内で実行される最初のコンポーネントとして期待され、作業項目がまだ存在しない場合は次の項目の読み取りのみを行います。AL が初期作業項目を渡していた場合、イテレーターはこの最初のサイクルについていずれのデータも読み取りません。 また、これは一連のイテレーターが連動することを意味します。最初のイテレーターがデータの終わりに到達して何も戻さなかった場合 (null)、2 番目のイテレーターが項目を戻し始めます。

一方、サーバー・モードでは、コネクターが (例えば IP ポートまたはイベント通知コールバックに対して) サーバー・リスナー・スレッドを開始し、次の供給コネクターに制御を渡します。

AssemblyLine 関数コンポーネント (AL FC) を使用して AL を呼び出す際に、手動サイクル・モードを使用する場合、FC が呼び出しを実行するたびに「フロー」セクションのコンポーネントのみが使用されます。

IBM Security Directory Integrator ナビゲーターのワークスペースに **Connectors** フォルダーがあります。このフォルダーには、構成済みコネクターのライブラリーを維持できます。これは、コネクター・プールが定義される場所でもあります。

コネクター・モード

AssemblyLine コネクターのモードは、データ・フロー内でそのコネクターが果たす役割を定義し、AssemblyLine の自動化動作がコンポーネントをどのように実行するかを制御します。コネクターのモードによって、入力ソースに対して読み取りを行うか、書き込みを行うか、あるいはその両方を行うかが決まります。

コネクターは、以下の標準モードのうちの 1 つを設定できます。

- イテレーター
- ルックアップ
- AddOnly
- 更新
- 削除
- CallReply

- サーバー
- デルタ

これらのモードについては、以下の各セクションで説明します。コネクター・モードの動作および AssemblyLine の全般的な動作について詳しくは、「リファレンス」の『AssemblyLine およびコネクター・モードのフローチャート』を参照してください。

イテレーター・モード:

イテレーター・モードのコネクターは、データ・ソースを走査してそのデータを抽出するために使用されます。イテレーター・モードのコネクターは、データ・ソース項目を実際に反復処理し、その属性値を読み取り、各項目を 1 つずつ AssemblyLine の「フロー」セクションのコンポーネントに渡します。一般に、イテレーター・モードのコネクターをイテレーター・コネクター、あるいは単にイテレーターと呼びます。

一般に、AssemblyLine (IWE で呼び出されるものを除く。8 ページの『初期作業項目 (IWE) の提供』を参照) には、イテレーター・モードのコネクターが少なくとも 1 つ含まれています。イテレーターは、作業項目を作成して AL の「フロー」セクションに渡すことで、AssemblyLine にデータを提供します。

「フロー」セクションのコンポーネントは、フロー・リストの最上部から順次有効になります。フロー処理が完了すると、制御はイテレーターに戻され、次の項目が検索されます。

AssemblyLine 内の複数のイテレーター: イテレーター・モードのコネクターが複数ある場合、それらのコネクターは、構成 (および「フィード」セクション内の構成エディターのコネクター・リスト) に表示される順序でスタックされ、1 つずつ処理されます。したがって、2 つのイテレーターを使用している場合は、第 1 のイテレーターがデータ・ソースを読み取り、結果の作業項目をイテレーター以外の最初のコネクターに渡します。この処理は、イテレーターがデータ・セットの終わりに到達するまで反復されます。 第 1 のイテレーターが入力ソースを読み終えると、第 2 のイテレーターがデータの読み取りを開始します。

初期作業項目は、他のどのイテレーターよりも前に処理された目に見えないイテレーターからのデータと見なされます。つまり、IWE は AssemblyLine 内のイテレーター以外の最初のコネクターに渡され、最初のサイクル中にはすべてのイテレーターがスキップされます。この動作については、「リファレンス」の『AssemblyLine およびコネクター・モードのフローチャート』の『AssemblyLine フロー』ページを参照してください。

2 つのイテレーターを持つ AssemblyLine があるとします。**a** は **b** の前にあります。第 1 のイテレーター **a** が項目を戻さなくなるまで **a** が使用されます。次に、AssemblyLine は **b** に切り替えます (**a** を無視します)。この AssemblyLine に初期作業項目 (IWE) が渡された場合は、最初のサイクルでは 2 つのイテレーターが無視され、その後で AssemblyLine は **a** の呼び出しを開始します。

IWE を使用して AssemblyLine に構成パラメーターを渡すことはできますが、データを渡すことはできません。ただし、IWE が存在すると、最初のサイクル中に AssemblyLine 内のイテレーターがスキップされます。これを回避したい場合は、プ

ロローグ・スクリプト内で task.setWork(null) 関数を呼び出して、作業項目オブジェクトを空にする必要があります。これにより、第 1 のイテレーターが正常に作動します。

イテレーター・モードの使用:

イテレーターによって AssemblyLine を開始することは非常に一般的です。

イテレーター・モードのコネクターを使用するための最も一般的なパターンは次のとおりです。

1. 構成エディターで、イテレーター・モードのコネクターをワークスペースに追加します。 126 ページの『コネクターの作成』を参照してください。
2. このコネクターのモード (イテレーター) およびその他の接続パラメーターを「**接続**」タブで設定します。必須パラメーターはアスタリスク (*) で示されます。コネクターの種類によっては、さらに「**パーサー**」タブでパーサーの構成が必要になることがあります。
3. 属性マップをセットアップします。 123 ページの『入力属性マッピング』を参照してください。

これらのマップされた属性は、データ・ソースから取得され、*作業* 項目内に配置され、AssemblyLine の「フロー」セクションのコネクターに渡されます。

AssemblyLine にコネクターを直接作成していない場合は、AssemblyLine でこのコネクターを使用するため、コネクターを <workspace>/Resources/Connectors 内の位置から AssemblyLine の「**供給**」セクションにドラッグします。

初期作業項目 (IWE) の提供: 代わりに、この方法で TCB を使用してパラメーターを渡すことができます。この方法は、前のバージョンとの互換性を維持するためにサポートされています。

スクリプトで system.startAL() 呼び出しを実行して AssemblyLine を開始する場合、AssemblyLine にパラメーターを渡すには、初期作業項目に属性値またはプロパティー値を設定します。初期作業項目にアクセスするには、*work* 変数を使用します。その後、これらの値を適用して、コネクター・パラメーターを設定します。例えば、AssemblyLine の「**プロローグ - 開始**」フック内で *connectorName*.setParam() 関数を使用します。

注: task.setWork(null) 呼び出しを使用して作業項目をクリアする必要があります。クリアしないと、AssemblyLine 内のイテレーターが最初のサイクルでパススルーします。

getResult() 関数を使用して、AssemblyLine の結果 (AssemblyLine が停止した時点の**作業項目**) を確認できます。「リファレンス」の『ランタイム提供コネクター (Runtime provided Connector)』も参照してください。

IWE を使用してコネクターのパラメーター値を渡す例を次に示します。

```
var entry = system.newEntry();
entry.setAttribute ("userNameForLookup", "John Doe");

// Here we start the AssemblyLine
var al = main.startAL ( "EmailLookupAL", entry );
```

```
// wait for al to finish
al.join();

var result = al.getResult();

// assume al sets the mail attribute in its working entry
task.logmsg ("Returned email = " + result.getString("mail"));
```

ルックアップ・モード:

ルックアップ・モードを使用すると、システム内の属性間の関係を使用して、異なるデータ・ソースからのデータを結合できます。ルックアップ・モードのコネクターをルックアップ・コネクターと呼ぶこともあります。

構成エディターでルックアップ・コネクターをセットアップするには、コネクターに対して、AssemblyLine 内の既存のデータと接続されたシステム内で検出されたデータとの間の一致をどのように定義するかを指定する必要があります。これをコネクターのリンク基準 と呼びます。各ルックアップ・コネクターには、関連付けられた「**リンク基準**」タブがあるため、ここで一致する項目を検索するためのルールを定義します。詳しくは、21 ページの『リンク基準』を参照してください。

ルックアップ・モードの使用: ルックアップ・モードのコネクターを使用するための最も一般的なパターンは次のとおりです。

1. 構成エディターで、ルックアップ・モードのコネクターをワークスペースに追加します。126 ページの『コネクターの作成』を参照してください。
2. このコネクターのモード (ルックアップ) およびその他の接続パラメーターを「**接続**」タブで設定します。必須パラメーターはアスタリスク (*) で示されます。コネクターの種類によっては、さらに「**パーサー**」タブでパーサーの構成が必要になることがあります。
3. 属性マップをセットアップします。123 ページの『入力属性マッピング』を参照してください。
4. コネクター構成ウィンドウの「**リンク基準**」タブを開き、属性一致ルールを設定します。このプロセスの結果として、接続システムから取得する項目が判別されます。いくつかのオプションがあります。
   a. 「**追加**」をクリックして新規リンク基準を追加し、接続システムの属性、比較演算子 (Equals、 Begins With など)、および比較する作業項目属性を選択します。コネクターは、ルックアップを実行するときに、指定されたリンク基準に基づいて、基礎となる API またはプロトコル構文を作成します。ソリューションは、使用するシステムのタイプから独立しています。ブール演算子 AND で結合された複数のリンク基準を追加して、検索呼び出しを作成することもできます。
   b. 「**カスタム・スクリプトで基準を作成**」を選択することもできます。これにより、スクリプト・エディター・ウィンドウが開きます。このウィンドウで独自の検索ストリングを作成し、`ret.filter` オブジェクトを使用してコネクターにこのストリングを戻すことができます。例を示します。

      ```
      ret.filter = "uid=" + work.getString("uid");
      ```

   式は、リンク基準に対して動的に属性または値を指定する目的にも使用できます。詳しくは、 40 ページの『式』を参照してください。リンク基準について詳しくは、21 ページの『リンク基準』も参照してください。

入力マップ内で読み取る (および計算する) 属性は、作業項目オブジェクトを使用するほかの下流のコネクターおよびスクリプト・ロジックでも使用できます。

AssemblyLine 内にコネクターを直接作成しなかった場合、AssemblyLine でこのコネクターを使用するには、<workspace>/Resources/Connectors 内の位置から AssemblyLine の「**フロー**」セクションにコネクターをドラッグします。

AddOnly モード:

AddOnly モードのコネクター (通常 AddOnly コネクターと呼ばれる) は、データ・ソースに新規のデータ項目を追加するために使用されます。

このコネクター・モードで必要な構成はほとんどありません。接続パラメーターを設定し、「作業項目」から書き込む属性を選択 (マップ) します。

***AddOnly* モードの使用:** AddOnly モードのコネクターを使用するための最も一般的で簡単なパターンは次のとおりです。

1. 構成エディターで、AddOnly モードのコネクターをワークスペースに追加します。 126 ページの『コネクターの作成』を参照してください。
2. このコネクターのモード (AddOnly) およびその他の接続パラメーターを「**接続**」タブで設定します。必須パラメーターはアスタリスク (*) で示されます。コネクターの種類によっては、さらに「**パーサー**」タブでパーサーの構成が必要になることがあります。
3. 属性マップをセットアップします。 124 ページの『出力属性マッピング』を参照してください。

AssemblyLine 内にコネクターを直接作成しなかった場合、AssemblyLine でこのコネクターを使用するには、<workspace>/Resources/Connectors 内の位置から AssemblyLine の「**フロー**」セクションにコネクターをドラッグします。

AssemblyLine によりコネクターが呼び出されると、マップした作業項目属性が接続システムに出力されます。

更新モード:

更新モードのコネクター (通常、更新コネクターと呼ばれる) は、データ・ソースのデータを追加および変更するために使用されます。 AssemblyLine から渡される項目ごとに、更新コネクターはデータ・ソース内で一致する項目を検索して、受け取った項目属性の値でデータ・ソース内の項目の更新を試行します。一致する項目が見つからない場合、更新モード・コネクターは新規項目を追加します。

ルックアップ・コネクターの場合と同様に、コネクターに対して、AssemblyLine 内に既に存在するデータと接続されたシステム内で検出されたデータとの間の比較をどのように定義するかを指示する必要があります。これをコネクターのリンク基準と呼びます。各更新コネクターには、関連付けられた「リンク基準」タブ (21 ページの『リンク基準』を参照) があります。このタブでは、一致する項目を検索するためのルールを定義します。一致する項目が検出されない場合は、新規項目がデータ・ソースに追加されます。ただし、一致する項目が検出された場合は、その項目が変更されます。複数の項目がリンク基準と一致する場合は、「複数項目時」フッ

クが呼び出されます。また、出力マップを構成して、追加操作または変更操作中に使用する属性を指定することもできます。

変更操作中には、出力マップ内で「変更 (Mod)」とマークされた属性のみがデータ・ソース内で変更されます。AssemblyLine から渡された項目の 1 つの属性に値がない場合は、その属性の NULL 動作が有効になります。この動作が「削除」に設定されている場合は、変更される項目内に属性が存在しないため、データ・ソースの属性を変更することはできません。この動作が「*NULL*」に設定されている場合は、変更される項目内に属性が存在しますが、NULL 値が設定されています。この場合は、データ・ソースの属性が削除されます。

更新コネクターが提供する重要な機能の 1 つは、「変更の計算」オプションです。このオプションをオンにすると、コネクターは最初に新規の値を既存の値と照合して検査し、必要な場合にのみ値を更新します。これにより、不必要な更新をスキップできます。この機能は、更新対象の特定のデータ・ソースに対する更新操作で大量の処理が必要な場合は便利です。

一部の更新コネクターでは、更新実行時に不要なルックアップをスキップするオプションがあります。コネクターでこのオプションがサポートされている場合、「**変更の計算**」チェック・ボックスの隣に「**ルックアップのスキップ**」チェック・ボックスが表示されます。このオプションを選択すると、コネクターの動作が変更され、リンク基準に対応する項目を検索するためのルックアップは実行されなくなります。このため、「ルックアップ前」フックも「ルックアップ後」フックも呼び出されません。また、「複数項目時」フックを呼び出すこともできません。このオプションがオンになっていると、検索条件のみが作成され、変更操作は直接呼び出されます。リンク基準に一致する項目が検出されない場合、新規項目は追加されないため、これは更新モードのコネクターのデフォルトの動作とは異なります。

更新モードの使用: 更新モードのコネクターを使用するための最も一般的で簡単なパターンは次のとおりです。

1. 構成エディターで、更新モードのコネクターをワークスペースに追加します。126 ページの『コネクターの作成』を参照してください。
2. このコネクターのモード (更新) およびその他の接続パラメーターを「**接続**」タブで設定します。必須パラメーターはアスタリスク (*) で示されます。コネクターの種類によっては、さらに「**パーサー**」タブでパーサーの構成が必要になることがあります。
3. 属性マップをセットアップします。124 ページの『出力属性マッピング』を参照してください。
4. コネクター構成ウィンドウの「**リンク基準**」タブを開き、属性一致ルールを設定します。ここでいくつかのオプションを選択できます。
   a. 「**追加**」をクリックして新規リンク基準を追加し、接続システムの属性、比較演算子 (Equals、 Begins With など)、および比較する作業項目属性を選択します。コネクターは、ルックアップを実行するときに、指定されたリンク基準に基づいて、基礎となる API またはプロトコル構文を作成します。ソリューションは、使用するシステムのタイプから独立しています。ブール演算子 AND で結合された複数のリンク基準を追加して、検索呼び出しを作成することもできます。

b. 「**カスタム・スクリプトで基準を作成**」を選択することもできます。これにより、スクリプト・エディター・ウィンドウが開きます。このウィンドウで独自の検索ストリングを作成し、`ret.filter` オブジェクトを使用してコネクターにこのストリングを戻すことができます。例を示します。

```
ret.filter = "uid=" + work.getString("uid");
```

式は、リンク基準に対して動的に属性または値を指定する目的にも使用できます。詳しくは、40 ページの『式』を参照してください。リンク基準について詳しくは、21 ページの『リンク基準』も参照してください。

入力時にマップ対象として選択された属性を持つ項目が、AssemblyLine の実行中にデータ・ソースに追加されます。

AssemblyLine でコネクターを使用するには、`<workspace>/Resources/Connectors` 内の位置から AssemblyLine の「**フロー**」セクションにコネクターをドラッグします。作業項目属性を出力にマップするには、以前にマップした属性を、AssemblyLine の「**属性マップ**」ウィンドウの更新コネクターにドラッグします。

また、属性を新規に作成することもできます。新規に作成するには、このウィンドウで「コネクター」を右クリックし、「**属性マップ項目の追加 (Add attribute map item)**」を選択します。

注: 更新モードでは、複数の項目を更新できます。「リファレンス」の『AssemblyLine およびコネクター・モードのフローチャート』を参照してください。

削除モード:

削除モードのコネクター(削除コネクターと呼ばれることが多い) は、データ・ソースからデータを削除するために使用されます。

削除コネクターに項目が渡されるたび、削除コネクターは、接続されたシステム内で一致するデータを探し出そうとします。一致する項目が 1 つだけ検出された場合は、その項目が削除されます。1 つも検出されなかった場合は、「一致なしの場合」フックが呼び出されます。複数の一致が検出された場合は、「複数項目時」フックが呼び出されます。ルックアップ・モードや更新モードの場合と同様に、削除モードでも、削除対象の一致項目を検索するためのルールを定義する必要があります。これはコネクターの「リンク基準」タブで構成します。

削除コネクターの中には、削除を実行する際に不要なルックアップをスキップするオプションを備えたものがあります。コネクターでこのオプションがサポートされている場合、「変更の計算」チェック・ボックスの隣に「ルックアップのスキップ」チェック・ボックスが表示されます。このオプションを選択すると、コネクターの動作が変更され、リンク基準に対応する項目を検索するためのルックアップは実行されなくなります。このため、「ルックアップ前」フックも「ルックアップ後」フックも呼び出されません。また、「複数項目時」フックを呼び出すこともできません。このオプションをオンにすると、検索基準だけが作成され、削除は直接呼び出されます。

削除モードの使用: 削除モードのコネクターを使用するための最も一般的で簡単なパターンは次のとおりです。

1. 構成エディターで、削除モードのコネクターをワークスペースに追加します。126 ページの『コネクターの作成』を参照してください。
2. このコネクターのモード (削除) およびその他の接続パラメーターを「**接続**」タブで設定します。必須パラメーターはアスタリスク (*) で示されます。コネクターの種類によっては、さらに「**パーサー**」タブでパーサーの構成が必要になることがあります。
3. 属性マップをセットアップします。 123 ページの『入力属性マッピング』を参照してください。
4. 「入力属性マップ」タブでは、「コネクター属性」リストから属性を選択し、入力マップへドラッグすることができます。属性を手動で追加また除去することもできます。

   注: 削除モードの入力マップは、データ・ソース内で検出された一致項目を *conn* 項目オブジェクトに読み込むために使用されます。この項目オブジェクトをスクリプト内で (例えば項目を実際に削除するかどうかを判別するために) 使用できます。
5. コネクター構成ウィンドウの「**リンク基準**」タブを開き、属性一致ルールを設定します。このプロセスの結果として、接続システムから取得する項目が判別されます。いくつかのオプションがあります。
   a. 「**追加**」をクリックして新規リンク基準を追加し、接続システムの属性、比較演算子 (Equals、 Begins With など)、および比較する作業項目属性を選択します。コネクターは、ルックアップを実行するときに、指定されたリンク基準に基づいて、基礎となる API またはプロトコル構文を作成します。ソリューションは、使用するシステムのタイプから独立しています。ブール演算子 AND で結合された複数のリンク基準を追加して、検索呼び出しを作成することもできます。
   b. 「**カスタム・スクリプトで基準を作成**」を選択することもできます。これにより、スクリプト・エディター・ウィンドウが開きます。このウィンドウで独自の検索ストリングを作成し、`ret.filter` オブジェクトを使用してコネクターにこのストリングを戻すことができます。例を示します。

   ```
   ret.filter = "uid=" + work.getString("uid");
   ```

   リンク基準について詳しくは、 21 ページの『リンク基準』を参照してください。

AssemblyLine 内にコネクターを直接作成しなかった場合、AssemblyLine でこのコネクターを使用するには、`<workspace>/Resources/Connectors` 内の位置から AssemblyLine の「**フロー**」セクションにコネクターをドラッグします。

CallReply モード:

CallReply モードは、入力パラメーターの送信を要求し、戻り値を含む応答を受信するデータ・ソース・サービス (Web サービスなど) に対する要求を作成するために使用されます。

他のモードとは異なり、CallReply モードでは、入力属性マップと出力属性マップの両方にアクセスできます。

コネクターは最初に出力マップ操作を実行します。そうすることで、出力マップ操作によって提供されるパラメーターを使用して、外部システムを呼び出します。その後すぐに入力マップ操作を実行するため、外部システムから応答を受け取ります。

***CallReply* モードの使用:** CallReply モードのコネクターを使用するための最も一般的で簡単なパターンは次のとおりです。

1. 構成エディターで、CallReply モードのコネクターをワークスペースに追加します。 126 ページの『コネクターの作成』を参照してください。ほとんどのコネクターではこのモードはサポートされていません。
2. このコネクターのモード (CallReply) およびその他の接続パラメーターを「**接続**」タブで設定します。必須パラメーターはアスタリスク (*) で示されます。コネクターの種類によっては、さらに「**パーサー**」タブでパーサーの構成が必要になることがあります。
3. 出力属性マップをセットアップします。 124 ページの『出力属性マッピング』を参照してください。
4. 入力属性マップをセットアップします。 123 ページの『入力属性マッピング』を参照してください。

出力マップの属性は、接続システムに入力パラメーターとして提供される点に注意してください。入力属性マップを介してマップされた属性には、接続システムからの応答が含まれています。

サーバー・モード:

サーバー・モードは一部のコネクターで使用可能です。サーバー・モードでは、着信イベント待機機能が提供され、着信イベントを処理するスレッドがディスパッチされ、応答がイベントの発信元に送り返されます。

現時点では、すべてのサーバー・モード・コネクターは接続ベースです。したがって、「供給」セクションでサーバー・モード・コネクターを使用しているすべての AssemblyLine (AL) は初期化を実行してから、着信接続 (TCP、HTTP、LDAP、Web サービス、SNMP 接続経由) を待機します。接続が開始されると、サーバー・モード・コネクターは、コネクター自体が含まれている AL を複製し、再び次のイベント (新規接続の開始) を待機します。複製された作業 AL ではその間、サーバー・モード・コネクターがイテレーター・モードに移行し、接続からのデータの読み取りを開始します。接続から読み込まれたデータは、通常のイテレーター方式の処理が行われる AL の残りの部分に送られます。つまり、標準のイテレーター・フック・フローに従い、一度に 1 つずつイベント項目が読み込まれ、他のフロー・コンポーネントに渡され処理されます。この処理は、読み取るデータが存在する限り続きます。各サイクル (サイクルは通常 1 つのみ) の最後で、サーバー・モード・コネクターを含む AL は、例えば `system.skipEntry();` などで応答フェーズをスキップする場合を除いて、応答をクライアントに送り返します。

AL へのデータのフィードが完了すると (つまり、データ・ソースのデータをすべて読み込むと)、そのスレッドは終了します。つまり、この時点で作業 AL はクリアされ、必要に応じて、この AL インスタンスが再び使用可能であることがプール・マネージャーに通知されます。

元のサーバー・モード・コネクターは、この時点でもアクティブでその他の接続の開始を listen しています。

注: ほとんど発生することがない特定の条件下 (zOS などのオペレーティング・システムを実行するサーバーに、5 を超えるクライアント・リクエストが同時に発行された場合など) では、SNMP クライアントは終了し、好ましくない Protocol Data Unit (PDU) 例外がスローされます。ただし、現実的な実際の状況では、エージェント (SNMP サーバー・コネクターなど) に複数のマネージャー (SNMP コネクターなど) からクエリーが送られることはほとんどありません。

SNMP コネクターには、資料に記載されていない *snmpWalkTimeout* という名前の構成パラメーターがあります。このパラメーターのデフォルト値 (5000 ミリ秒) は指定変更できます。ただし、このパラメーターには構成エディターからアクセスすることはできません。このパラメーターの値は、JavaScript を使用して指定変更できます。この snmpWalkTimeout パラメーターには、次の形式で値を設定します。

```
thisConnector.connector.setParam("snmpWalkTimeout", "100000");
```

サーバー・モードと ALPool: クローン AL を作成するプロセスは、AssemblyLine プール (ALPool) を使用して最適化できます。イベントの検出時には、サーバー・モードのコネクターは、この AL の「フロー」セクションに進むか、ALPool がこの AL に構成されている場合、プール・マネージャー・プロセスに通知して、このイベントを処理する使用可能な AL インスタンスを要求します。

サーバー・モードのコネクターを含む AssemblyLine で ALPool が使用されると、ALPool は最初から最後まで AL インスタンスを実行します。ALPool の AL インスタンスがフロー・コネクターをクローズする前に、ALPool は、これらのコネクターをプール済みのコネクターのセットに取り込みます。これらのコネクターは、ALPool によって作成される次の AL インスタンスで再利用されます。つまり、ALPool は `tcb.setRuntimeConnector()` メソッドを使用します。

コネクターのプールの動作は、次の 2 つのシステム・プロパティーによって制御されます。

com.ibm.di.server.connectorpooltimeout
このプロパティーでは、プールされているコネクター・セットを解放するまでのタイムアウトを秒単位で定義します。

表 1. コネクター・プールの「タイムアウト」プロパティー値の表

値	意味
< 0	コネクター・プールを使用不可にする。
0	タイムアウトを使用不可にする。プール・コネクターはタイムアウトしない。
> 0	プールされたコネクターがタイムアウトするまでの秒数。

com.ibm.di.server.connectorpoolexclude
このプロパティーでは、プールから除外されるコネクター・タイプを定義します。コネクターのクラス名がこのコンマ区切りリストに表示されている場合、コネクター・プール・セットには含まれていません。

新規 AssemblyLine (AL) インスタンスが ALPool によって作成されると、プールされている使用可能なコネクター・セットを検索し、存在する場合は、実行時に提供

されるコネクターとして新規 AL インスタンスに提供されます。 これにより、属性マッピングやフック実行などの点で、一般に AL のフローが適切なものになります。

コネクターが共有されることはありません。コネクターが使用される場合は、1 つの AL インスタンスのみに割り当てられます。

サーバー・モードの使用: サーバー・モードのコネクターを使用する最も一般的なパターンは次のとおりです。

1. 構成エディターで、サーバー・モードのコネクターをワークスペースに追加します。 126 ページの『コネクターの作成』を参照してください。
2. このコネクターのモード (サーバー) およびその他の接続パラメーターを「**接続**」タブで設定します。必須パラメーターはアスタリスク (*) で示されます。コネクターの種類によっては、さらに「**パーサー**」タブでパーサーの構成が必要になることがあります。
3. 属性マップをセットアップします。 123 ページの『入力属性マッピング』を参照してください。

これらのマップされた属性は、データ・ソースから取得され、*作業* 項目内に配置され、AssemblyLine の「フロー」セクションのコネクターに渡されます。

注:

1. サーバー・モードのコネクターは、通常、接続先のクライアントに情報を戻す必要があるという点で特別です。その特質上、「**属性のディスカバー**」をクリックして属性マップをセットアップしても、多くの場合は有用なスキーマが作成されません。したがって、多くの場合は「**新規属性の追加 (Add new attribute)**」を選択し、属性マップのセットアップを完全に手動で行う必要があります。あるいは、不要なマッピングを削除するため、マッピングを選択して削除します。この方法でマップされたデータは、AL のサイクルごとにクライアントに送り返されます。先に説明したように、操作の通常の要求または応答では、一般的にサイクルは 1 つのみです。
2. TCP プロトコルに基づいたサーバー・モード・コネクターには、着信接続におけるキューの長さを制御する接続バックログと呼ばれるパラメーターが追加されました。
3. AssemblyLine コンポーネントのリストで、「供給」セクションと「フロー」セクションを監視します。サーバー・モード・コネクターは、AssemblyLine 処理の 3 番目のフェーズ (*応答* フェーズ) を監視します。出力マップと応答フックは、画面上ではサーバー・モード・コネクター自体に含まれますが、応答動作は、「フロー」セクションの処理が完了した後に実行されます。

   `system.skipEntry();` 呼び出しは、応答動作をスキップすることを AssemblyLine に対して指示します。このためサーバー・モード・コネクターから応答はありません。「フロー」セクションの残りのコンポーネントを単にスキップして、応答を送信する場合は、代わりに `system.exitBranch("Flow");` 呼び出しを使用します。

AssemblyLine にコネクターを直接作成していない場合は、AssemblyLine でこのコネクターを使用するため、コネクターを `<workspace>/Resources/Connectors` 内の位置から AssemblyLine の「**供給**」セクションにドラッグします。

デルタ・モード:

デルタ・モードは、接続されたシステムへの増分変更を提供することで、データへの変更適用を簡略化できるように設計されています。これは、デルタ命令コードに基づいています。

デルタ命令コードは、イテレーター・デルタ・エンジン機能 (イテレーターの「**デルタ**」タブ)、変更検出コネクター (IBM Security Directory Server、LDAP、Active Directory (AD) コネクターなど)、または RDBMS および Domino 変更のコネクターのいずれかにより設定されるか、あるいはこのデルタ情報を Lightweight Directory Interchange Format (LDIF) または Directory Services Markup Language (DSML) パーサーで構文解析することにより設定されます。

IBM Security Directory Integrator の V6.1 より前のバージョンでは、デルタ・エンジンの処理中にデルタ・ストア (システム・ストアの一機能) に書き込まれたスナップショットが即座にコミットされていました。その結果、デルタ・エンジンは、ALフロー・セクションの処理が失敗した場合でも、変更された項目を処理されたものと見なしていました。この制限は、コネクターのデルタ・タブの *Commit* パラメーターで解決されます。このパラメーターの設定により、デルタ・エンジンが着信データのスナップショットをシステム・ストアにコミットするタイミングを制御できます。

デルタ・モードは、LDAP コネクターおよび JDBC コネクターでのみ使用可能です。

注: デルタ・モードのコネクターは、デルタ情報を提供する別のコネクターを組み合わせて使用する必要があります。そうでない場合、デルタ・モードはデルタ命令コードを処理できません。

IBM Security Directory Integrator のデルタ機能 (221 ページの『第 7 章 デルタ』を参照) は、同期ソリューションを円滑に実行するように設計されています。システムのデルタ機能は、2 つのセクション、「*検出*」と「*アプリケーション*」に分割されていることがわかります。

デルタ検出: IBM Security Directory Integrator には、変更 (デルタ) 検出の手段とツールがいくつか用意されています。

デルタ・エンジン

これは、イテレーター・モードのコネクターで使用できる機能です。イテレーターの「デルタ」タブで使用可能にすると、デルタ・エンジン機能がシステム・ストアを使用して、繰り返されるデータの「スナップショット」を取ります。継続的に実行すると、繰り返される各項目がスナップショット・データベース (デルタ・ストア) と比較され、変更されているかどうかが確認されます。

変更検出コネクター

このコンポーネントは、接続されたシステムの情報を利用して変更を検出します。コネクターのタイプによって、イテレーター・モードまたはサーバー・モードのいずれかで使用されます。例えば、イテレーター・モードは、

多くの変更検出コネクター (LDAP 用、IBM Security Directory Server の変更ログ、RDBMS、Active Directory、Notes/Domino の変更検出コネクターなど) に使用されます。

変更検出コネクターは、それぞれが同様に動作し、共通の設定に対して同じパラメーター・ラベルが使用されるように設計されています。次のコネクターがあります。

- IBM Security Directory Server 変更ログ
- AD 変更検出 (Active Directory)
- Domino 変更検出
- Sun Directory 変更検出 (openLDAP、SunOne、iPlanet など)
- RDBMS 変更検出 (DB2®、Oracle、SQL サーバーなど)
- z/OS® LDAP 変更ログ

注: IBM Security Directory Integrator バージョン 7.2 以降では z/OS オペレーティング・システムはサポートされません。

これらのコネクターについて詳しくは、「*リファレンス*」の『コネクター』セクションを参照してください。

デルタ・エンジン機能は、特定の変更について、属性の個々の値に至るまで詳細に報告します。こうした詳細な変更点まで検出する機能は、LDIF ファイルを解析する際にも適用可能です。他のコンポーネントは、項目全体が追加、変更、または削除された場合のみ簡単に報告されます。

イテレーターの「デルタ」タブで「**重複デルタ鍵を許可**」チェック・ボックスを選択すると、長期間実行する AssemblyLine で同一項目を複数回処理する必要がある場合に重複デルタ鍵が許可されます。つまり、項目が既に更新されている場合に、変更ログまたは変更検出コネクター、デルタ・モード・コネクター、およびデルタ・モード・ストアを使用する AssemblyLine で重複項目を処理することができます。

重要: 例えば、AssemblyLine に複数の変更ログ・コネクターおよびデルタ・モード・コネクターが組み込まれている可能性があります。この場合、デルタ・モード・コネクターが変更ログ・コネクターと同じ基盤システムをポイントすると、デルタ操作によって変更ログを再度起動することができます。新しく受信した変更とデルタ・エンジンが起動した変更を区別する方法はないので、無限ループに落ちないようにするためには、適用するシナリオを慎重に検討する必要があります。

デルタ・エンジンで計算されるデルタ情報は、作業項目オブジェクトに保管され、使用される変更検出コンポーネントまたは機能によって、*項目レベルの命令コード*として、*属性レベル* または*属性値レベル* で保管できます。

例として、デルタ・エンジン機能を有効にしてファイル・システム・コネクターをセットアップします。テキスト・エディター内で簡単に変更可能な単純な XML 文書を繰り返してください。例を示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<DocRoot>
    <Entry>
        <Telephone>
            <ValueTag>111-1111</ValueTag>
```

```
            <ValueTag>222-2222</ValueTag>
            <ValueTag>333-3333</ValueTag>
        </Telephone>
        <Birthdate>1958-12-24</Birthdate>
        <Title>Full-Time SDI Specialist</Title>
        <uid>jdoe</uid>
        <FullName>John Doe</FullName>
    </Entry>
</DocRoot>
```

属性マップのすべての文字をマップする特殊文字であるアスタリスク (*) を必ず使用してください。これは、戻されるすべての属性が作業項目オブジェクトに必ずマップされるようにするために必要な唯一の属性です。

ここで次のコードを使用してスクリプト・コンポーネントを追加します。

```
// Get the names of all Attributes in work as a String array
var attName = work.getAttributeNames();
// Print the Entry-level delta op code
task.logmsg(" Entry (" +
    work.getString( "FullName" ) + ") : " +
    work.getOperation() );
// Loop through all the Attributes in work
for (i = 0; i < attName.length; i++) {
  // Grab an Attribute and print the Attribute-level op code
 att = work.getAttribute( attName[ i ] );
 task.logmsg("    Att ( " + attName[i] + ") : " + att.getOperation() );
  // Now loop through all the Attribute's values and print their op codes
 for (j = 0; j < att.size(); j++) {
  task.logmsg( "      Val (" +
                     att.getValue( j ) + ") : " +
                     att.getValueOperation( j ) );
 }
}
```

この AL を初めて実行する際にスクリプト・コンポーネント・コードによって、次のログ出力が作成されます。

```
12:46:31   Entry (John Doe) : add
12:46:31      Att ( Telephone) : replace
12:46:31        Val (111-1111) :
12:46:31        Val (222-2222) :
12:46:31        Val (333-3333) :
12:46:31      Att ( Birthdate) : replace
12:46:31        Val (1958-12-24) :
12:46:31      Att ( Title) : replace
12:46:31        Val (Full-Time SDI Specialist) :
12:46:31      Att ( uid) : replace
12:46:31        Val (jdoe) :
12:46:31      Att ( FullName) : replace
12:46:31        Val (John Doe) :
```

この項目が以前に空だったデルタ・ストアで見つからなかったため、項目レベルで *new* というタグが付けられます。また、その各属性には、すべての値が変更されていることを意味する *replace* コードが付きます (デルタによりこれが新規データであることが通知されるので意味があります)。

XML ファイルに次の変更を加えます。

1. 最後の Telephone (電話番号) の値を 333-3334 に変更します。
2. Birthdate を削除します。
3. 新規の Address 属性を追加します。

構成の結果は次のようになります。

```
<?xml version="1.0" encoding="UTF-8"?>
<DocRoot>
    <Entry>
        <Telephone>
            <ValueTag>111-1111</ValueTag>
            <ValueTag>222-2222</ValueTag>
            <ValueTag>333-3334</ValueTag>
        </Telephone>
        <Title>Full-Time SDI Specialist</Title>
        <uid>jdoe</uid>
        <FullName>John Doe</FullName>
        <Address>123 Willowby Lane</Address>
    </Entry>
</DocRoot>
```

AL を再実行します。今回のログの出力は、次のようになります。

```
13:53:22   Entry (John Doe) : modify
13:53:22      Att ( Telephone) : modify
13:53:22        Val (111-1111) : unchanged
13:53:22        Val (222-2222) : unchanged
13:53:22        Val (333-3334) : add
13:53:22        Val (333-3333) : delete
13:53:22      Att ( Birthdate) : delete
13:53:22        Val (1958-12-24) : delete
13:53:22      Att ( uid) : unchanged
13:53:22        Val (jdoe) : unchanged
13:53:22      Att ( Title) : unchanged
13:53:22        Val (Full-Time SDI Specialist) : unchanged
13:53:22      Att ( Address) : add
13:53:22        Val (123 Willowby Lane) : add
13:53:22      Att ( FullName) : unchanged
13:53:22        Val (John Doe) : unchanged
```

ここでは、項目に *modify* というタグが付けられ、変更が属性に反映されています。ご覧のように、Birthdate 属性には *delete*、Address には *add* のマークが付いています。そのため、入力マップでは、すべての文字をマップするための特殊文字を使用しました。XML 文書の最初のバージョンに存在する属性のみをマップした場合、住所が入力に表示されても取得されません。

modify としてマークされている Telephone 属性の下にある値項目のうち、最後の 2 つに特に注目してください。 この属性値の 1 つを変更すると、2 つのデルタ項目 (値 *delete*、次いで *add*) が生じます。

IBM Security Directory Integrator の以前のバージョンでデータ同期 AssemblyLine を構築するには、フロー制御を処理するためにスクリプトを記述する必要がありました。変更コンポーネントまたは機能から *add*、*modify*、および *delete* を受け取ることはできますが、コネクターについては、要求された更新と削除の 2 つの出力モードのうち、いずれか一方にのみ設定が可能でした。そのため、命令コードがこのコンポーネントのモードと一致しない場合は、同じターゲット・システムを指すコネクターを 2 つ含め、それぞれの「実行前」フックにスクリプトを配置して項目を無視するか、単一コネクター (更新または削除モード) の状態を*受動* にして、命令コードを確認したスクリプト・コードから実行を制御するかのいずれかでした。つまり、項目が変更されている場合に何が変更されたかがわかっていても、その変更をデータ・ソースに書き戻す前なら、更新モードのコネクターは元のデータを読み込むことがまだ可能でした。これは、数千個の値を含む、多値のグループに関連し

た属性で、1 つの値だけを変更する場合に、ネットワークまたはデータ・ソースに不要なトラフィックを発生させる可能性があります。

コネクターのデルタ・モードを入力します。

デルタ・アプリケーション (コネクターのデルタ・モード): デルタ・モードは、デルタ情報の適用を簡素化するように設計されています (つまり、さまざまな方法で実際の変更を行います)。

第 1 に、デルタ・モードはすべてのデルタ・タイプ (追加、変更、削除) を処理します。これにより、データ同期 AL の数は、2 つのコネクター (変更を選出する「フィード」セクションのデルタ検出コネクターと、ターゲット・システムに変更を適用するデルタ・モードのコネクター) に削減されます。

また、デルタ・モードでは、ターゲット・システム自体がサポートする最下位のレベルでデルタ情報を適用します。 これは、コネクター・インターフェースの最初のチェックによって行われ、増分変更のどのレベルがデータ・ソースによってサポートされるのかを確認されます[3]。LDAP ディレクトリーで作業中の場合、デルタ・モードが属性値の追加と削除を実行します。従来の RDBMS (JDBC) の場合、列の値の削除と追加を行うことは意味がありませんでした。そのため、その属性値の置換として処理されています。

これは、この機能をサポートするデータ・ソースに対して、デルタ・モードにより自動的に処理されます[4]。データ・ソースが最適化された呼び出しにより増分変更を処理し、この呼び出しがコネクター・インターフェースによってサポートされる場合は、デルタ・モードがこの呼び出しを使用します。一方、接続されたシステムが「インテリジェント」デルタの更新メカニズムを提供しない場合、デルタ・モードは可能な限りシミュレートし、事前更新のルックアップ (更新モードに類似)、計算の変更、および削除した変更に続くアプリケーションを実行します。

リンク基準:

リンク基準は、更新、ルックアップ、および削除モードのコネクターに対して、AssemblyLine 内に存在するデータ属性と接続されたシステム内で検出されたデータ属性との間の一致をどのように定義するかを指示するために使用されます。

リンク基準には、構成エディターの「**リンク基準**」タブを使用してアクセスできます。リンク基準は、更新、ルックアップ、および削除コネクターのモードに対してのみ使用可能になります。

リンク基準には、**単純**と**拡張**の 2 つのタイプがあります。

単純リンク基準: 各単純リンク基準では、コネクター属性 (コネクター・スキーマ内で定義されている属性)、使用する演算子 (Contains、Equals など)、および演算で使用する値を指定します。使用する値を直接入力することも、AssemblyLine フローのこの時点で使用可能な作業項目内の属性値を参照することもできます。コネクターがルックアップ操作 (ルックアップ、更新、または削除モードの場合) を実行する

3. 増分変更をサポートするコネクターのみが LDAP および JDBC コネクターです。LDAP ディレクトリーがこの機能を提供します。

4. こうした組み込み機能の動作は、構成パラメーターとフック・コードを使用して制御することもできます。

場合、コネクターはリンク基準をデータ・ソース固有の呼び出しに変換するため、基礎となる技術にソリューションが依存することはありません。

作業項目内の属性値を使用してリンク基準を作成する場合は、リンク基準の「値」フィールドにその属性の名前を使用します。このとき、属性名の前にドル記号 ($) を付けます。したがって、*cn* という名前の属性と、作業項目内の FullName という名前の属性を比較する場合は、リンク基準を次のように指定します。

```
cn EQUALS $FullName
```

特定の個人を直接検索する場合は、リンク基準にリテラル定数値を設定します。

```
cn EQUALS Joe Smith
```

注: ドル記号 ($) を指定すると、多値属性の最初の値のみが比較されます。データ・ソース内の属性を、作業項目属性に格納されている複数の値のすべてに対して比較する場合は、アットマーク 記号 (@) を使用します。例を示します。

```
dn EQUALS @members
```

この例では、接続システム内の dn 属性と、作業項目内の *members* という名前の多値属性の値のすべてを比較します。

1 つのコネクターに対して複数のリンク基準を定義できます。通常、これらのリンク基準はブール演算子 AND を使用して結合され、一致の検索に使用されます。

ただし、「**いずれかと一致**」を選択している場合、OR 演算と同じように、リンク基準のいずれか 1 つが一致する必要があります。

比較する属性の名前は、式として指定できます (詳しくは 40 ページの『式』を参照)。単純リンク基準の「値」フィールドで使用できる形式は次のとおりです。

テキスト・ストリング
その値を持つ定数にマップされます。

$Name 属性 *Name* の最初の値である `work.getString("Name")` に対応します。

@Name
多値属性 *Name* の値の 1 つに一致します。

IBM Security Directory Integrator 式
詳細が 40 ページの『式』に記述されています。

拡張リンク基準: 「**カスタム・スクリプトで基準を作成**」チェック・ボックスをオンにして、独自のカスタム検索基準を作成することもできます。これをオンにすると、独自のリンク基準式を記述するためのスクリプト・エディターが表示されます。拡張リンク基準をサポートしていないコネクターもあります。拡張リンク基準のサポートの有無は、コネクターの資料に記載されています。「リファレンス」の『コネクター』を参照してください。

ユーザーが作成する検索式は、基礎となるシステムが要求する構文に準拠している必要があります。検索式をコネクターに渡すには、*ret.filter* オブジェクトにストリング式を取り込む必要があります。

SQL コネクターの単純な JavaScript の例を次に示します。

```
ret.filter = " ID LIKE '" + work.getString("Name") + "'";
```

このカスタム・リンク基準で想定されている例では、データ・ソースに *ID* (通常は列名) という名前の属性があり、この属性を作業項目内の *Name* 属性と比較します。

注:

1. SQL 式の最初の部分 `Select * from Table Where` は、IBM Security Directory Integrator によって提供されます。
2. 単一引用符が追加されたのは、`work.getString()` はストリングを戻し、SQL 構文ではストリング定数を単一引用符で囲む必要があるからです。
3. **$** と **@** を使用した特殊な構文は、ここでは使用されません。

リンク基準エラー

リンク基準を使用するときに発生する最も一般的なエラーは、次のとおりです。

```
ERROR> AssemblyLine x failed because
No criteria can be built from input (no link criteria specified)
```

このエラーは、リンク基準が参照する属性をルックアップ中に検出できなかった場合に発生します。 例えば、次のリンク基準の場合です。

```
uid equals $w_uid
```

作業項目内に *w_uid* が存在しない場合、リンク基準のセットアップは失敗します。この原因としては、入力ソースから読み取られていない (例えば、入力マップに含まれていない、または入力ソース内に存在しない) こと、またはスクリプト内の作業項目から削除されたことが考えられます。この場合、関数呼び出し `work.getAttribute("w_uid")` は、NULL を戻します。

これを回避する方法の 1 つは、ルックアップ、削除、または更新モードのコネクターの「実行前」フックでコードを記述して、属性が検出されないためにリンク基準を解決できない場合に操作をスキップさせることです。例を示します。

```
if (work.getAttribute("w_uid") == null)
  system.ignoreEntry();
```

ビジネス・ルールによっては、`ignoreEntry()` の代わりに `skipEntry()` を呼び出すなど、他の処理が必要になることがあります。skipEntry() の場合、AssemblyLine は現在の項目の処理を停止し、新規の反復の先頭から処理を開始します。ignoreEntry() 関数は現在のコネクターを単にスキップし、AssemblyLine の残りの部分の処理を継続します。

関数

関数 (関数コンポーネント (FC) とも呼ばれる) はコネクターに非常に類似したコンポーネントですが、モード設定がありません。コネクターは接続されたシステムに対する標準的なアクセス用の動詞 (ルックアップ、削除、更新など) を提供しますが、一方で関数は、パーサーを介したデータのプッシュ、他の AssemblyLine への作業のディスパッチ、Web サービス呼び出しなどの単一の操作のみ実行します。

概説

関数は AL の「フロー」セクションのいずれの場所でも使用可能です。構成ブラウザーの関数ライブラリー・フォルダーを使用して、ご使用の関数コンポーネントのライブラリーを管理できます。

コネクターと同様に、AssemblyLine における関数は、このコンポーネントを開始するタイミングを決定する「初期化」オプションを提供します。デフォルトでは、関数は AL の開始中に初期化されます。

スクリプト・コンポーネント

スクリプト・コンポーネント (SC) は、コネクターや関数コンポーネントとともに、AssemblyLine のデータ・フロー・リストのどこにでも配置可能なユーザー定義の JavaScript コード・ブロックであり、AL ワークフロー内の各サイクルのこの時点でスクリプト・コードを実行します。

概説

フックとは異なり、スクリプト・コンポーネントは AssemblyLine フロー内を容易に移動でき、独自のフロー・ロジックのデバッグから、プロトタイピングやインプリメントまですべてに対して強力なツールとなります。他のタイプの AL コンポーネントとは異なり、スクリプトには定義済みの振る舞いやモードがありません。したがって、スクリプトでは振る舞いやモードをインプリメントできます。スクリプトのライブラリーは、構成ブラウザー内の「スクリプト・ライブラリー」フォルダーに保管できます。

使用法

例えば、AssemblyLine の一部のみをテストおよびデバッグする場合は、次のコードを SC に置いて、AL フローを制限および制御できます。

```
task.dumpEntry( work );
system.skipEntry();
```

AssemblyLine のフローの適切な位置に配置された SC は、作業オブジェクトの内容をログ出力に書き込んで表示し、このサイクルで残りの AL をスキップします。コンポーネント・リスト内で SC を上下に移動すると、実際に実行される AL が制御できます。`system.skipTo("ALComponentName")` を使用して `system.skipEntry()` 呼び出しをスワップアウトすると、制御が特定の AL コンポーネントに直接渡されます。

SC を使用して他のコンポーネントを実行することもできます。 ディレクトリーやデータベースを同期させる際の典型的なシナリオは、更新された情報と削除された情報の両方を処理していることです。組み込み AL ワークフローから開始するコネクターは、1 回に 1 つのモード (更新または削除) のみを操作できるため、独自のコードを使用してこのロジックを少し拡張する必要があります。 1 つの方法として、更新モードと削除モードの 2 つのコネクターを追加し、各コネクターの「実行前」フックにコードを挿入して、処理する必要のない変更操作をスキップするように指定できます。例えば、更新コネクターの「実行前」フックに、次のように記述するとします。

```
// The LDAP change log contains an attribute called "changeType"
if (work.getString("changeType").equals("delete"))
 system.ignoreEntry();
```

すると、更新モードのコネクターは、削除済みの項目をスキップすることになります。削除モードのコネクターの「実行前」フックに補完コードを使用して、削除以外のすべてをスキップすることができます。

ただし、複数のターゲットに対する更新を同期化している場合、それぞれデータ・ソースごとに 2 つのコネクターを持つ必要があります。他に、スクリプトから開始する受動状態の単一コネクターを持つ方法があります。例として、*synchIDS* という受動状態の AL コネクターがあるとします。次のコードを使用して、SC を追加し実行できます。[5]

```
if (work.getString("changeType").equals("delete"))
 synchIDS.deleteEntry( work )
else
   synchIDS.update( work );
```

SC が受動状態のコネクターを使用していることを明確に指定する限り、この方法は、AssemblyLine の記述をより短くし、読み取りや保守を容易にします。これは、AL を短く保持し、カスタム・スクリプトに対して組み込みロジックを使用するという 2 つのベスト・プラクティスから選択する例になります。ただし、この場合、読みやすさと簡素化の目的は、スクリプト記述を少なくすることによって達成されます。

スクリプト・コンポーネントは、ロジックやスクリプト・コードをテストする場合にも役に立ちます。データ・フロー・リストの 1 つのスクリプト・コンポーネントを使用して AssemblyLine を作成し、コードに組み込んで実行します。

関連情報

49 ページの『第 2 章 IBM Security Directory Integrator のスクリプト記述』

AttributeMap

属性マップは、AssemblyLine 内部または AssemblyLine 外部へのデータ・フローの経路を示します。属性マップは、入力マップおよび出力マップとしてコネクターと関数に表示されます。また、AssemblyLine でスタンドアロン・コンポーネントとしても使用可能です。

概説

1 ページの『AssemblyLine』の冒頭の図では、3 つの属性マップが曲線の矢印として描かれています。これらのマップは、データを AssemblyLine に取り込む 2 つの入力マップと、データを書き込むためにコネクターのキャッシュ (*Conn* 項目) に渡す出力マップです。

各属性マップには、作業項目または Conn 項目に属性を作成するための一連のルールがあります。マッピング・ルールでは次の 2 つが指定されます。

5. 受動状態のコネクターは AL ロジックで実行されないため、データ・フロー・リスト内での指定は任意の場所で問題ありません。

1. ターゲット項目に作成される (または上書きされる) 属性の名前。入力マップと独立属性マップ・コンポーネントの場合、これは作業項目です。出力マップのターゲットは Conn 項目です。
2. 1 つ以上の値を属性に取り込むために使用される割り当て。割り当てとして割り当てスクリプトを使用できます。次に例を示します。

   ```
   work.Title
   ```

   または、オプションのトークン置換を含むリテラル・テキスト (改行文字を含む) にすることもできます。

   ```
   <html>
      <header>
         <title>{work.Title}</title>
      </header>
      <body>
         <p>Look for the "Title" Attribute value in the title of this page</p>
      </body>
   </html>
   ```

属性マップを作成するには、構成エディターを使用します。 118 ページの『属性マッピングおよびスキーマ』を参照してください。

属性マップでは、マップ・レベルと個々のマッピング・ルールの両方で継承がサポートされています。属性マップにスクリプトをドラッグして、継承される JavaScript マッピング・ルールを作成できる点に注意してください。

属性マップには、欠落データの処理方法を制御するための 『NULL 動作』という機能もあります。

属性マップについて詳しくは、 50 ページの『内部データ・モデル: 項目、属性、値』を参照してください。

NULL 動作

システムは、欠落している属性のマッピングを試行することがあります。例えば、入力データ・ソース内にオプションの電話番号がない場合や、出力マップ内の属性が作業項目から除去されている場合です。その他には、データベース表内のヌル可能列のように、属性があっても値がない場合があります。

欠落している値の処理方法 (NULL 値、空ストリングなど) は、データ・ソースごとに異なります。このセクションで説明する機能によって、欠落している属性 (または属性値) の処理のカスタマイズ方法も提供されます。この機能は *NULL 動作* と呼ばれ、これにより、「NULL 値」とその処理方法について定義できます。

注: JDBC コネクターには、`jdbcExposeNullValues` パラメーター設定があり、欠落している属性に NULL 値をマップできます (「*リファレンス*」の『JDBC コネクター』を参照してください)。

NULL 動作はさまざまなレベル (システム、構成、AssemblyLine、AttributeMap、および属性) で指定できます。ただし、NULL 動作はデータ・ソース固有の傾向が強いため、システム・プロパティーを属性マップ・レベルで設定することをお勧めします (コネクターの入力または出力マップにより処理されるすべての属性など)。可能なレベルを以下に詳述します。

システム・レベル

システム・レベルの NULL 動作を指定するには、global.properties ファイル内で、rsadmin.attribute.nullBehavior プロパティーおよび rsadmin.attribute.nullDefinition プロパティーの値をこのセクションで後述する 1 つに設定します。

構成レベル

この設定はシステム・レベルの NULL 動作を指定変更します。これを構成するには、プロパティー・ストア内で、rsadmin.attribute.nullBehavior プロパティーまたは rsadmin.attribute.nullDefinition プロパティーの値をこのセクションで後述する 1 つに設定します。

AssemblyLine レベル

AssemblyLine の NULL 動作を指定する場合は、AssemblyLine ツールバーの「**オプション...**」ボタンをクリックし、「**AssemblyLine 設定**」を選択してから、表示されたダイアログ・ボックスの「**NULL 値の動作**」をクリックします。

属性マップ・レベル

属性マップ・リストの上部にある「**その他...**」ボタンをクリックして、「**NULL 動作**」を選択することで、マップ内のすべての属性について NULL 動作が定義されます。

属性レベル

特定の属性の NULL 動作を構成するには、属性マップ内でその属性を右クリックし、「**NULL 動作**」を選択します。属性についてこの項目が定義されている場合には、マップされた項目に青丸記号が付いています。

以下に示すように、NULL 動作は NULL 値を定義するための 5 つの異なる設定をサポートします (各設定の括弧内のテキストは、システム・レベルの NULL 動作定義の設定に使用される値であり、通常はソリューションまたはグローバル・プロパティー・ストアに定義されることに注意してください)。各設定の括弧には実際のプロパティー値を示します。これらの定義は包括的な順序でリストされているため、2 つ目は 1 つ目を含み、3 つ目はそれ以前の 2 つを含みます。

属性が欠落している (AbsentAttribute)

属性マップにおいて 1 つ以上の値のソースとして参照された属性が欠落しています。

属性に値が存在しない (EmptyAttribute)

属性内に 1 つ以上の値のソースとして使用される属性が検出され、そこに値が存在していません。 1 つ目の場合も確認されます。

属性に空ストリング値が含まれる (EmptyString)

属性が検出され、そこに単一のストリング値のみ存在します。

値 (value)

指定した値が属性に含まれます。AssemblyLine、属性マップ、および属性レベルの NULL 値の定義の場合、この値は「NULL 動作」ダイアログの「**値**」フィールドに設定されます。ここで複数の属性値を指定する場合は、1 行に 1 つずつ値を入力します。システム・レベルおよび構成レベルの設

定で rsadmin.attribute.nullDefinition を使用する場合は、rsadmin.attribute.nullDefinitionValue プロパティーも設定する必要があります。

注: HTTP サーバー・コネクターに対していくつかの機能拡張が行われました。 HTTP サーバー・コネクターなどの TCP ベースのコンポーネントの構成画面には、 TCP ヘッダーを属性値として戻すためのスイッチがあります。このフラグがクリアされると、TCP ヘッダーは戻された項目オブジェクトにプロパティーとして保管されます。

デフォルトの動作 (Default Behavior)
NULL 値の定義は、上位のレベルから継承する必要があります。例えば、属性は属性マップ設定からその NULL 値の定義を継承し、属性マップ設定は AssemblyLine からそれを継承します。

注: 構成レベルの NULL 動作は、すべてのシステム・レベル設定を指定変更します。また、システム・レベルのデフォルトの動作設定は **delete** を指定するのと同じであり、構成レベルのデフォルトの動作設定は **value** を指定するのと同じです。

NULL 動作機能を使用して、NULL 値が検出された場合に実行されるアクションも定義します。

空ストリング (empty string)
欠落している属性には、空のストリング値 ("") を持つ単一値がマップされます。

NULL (null)
欠落している属性には、値がマップされません。つまり、att.getValue() 呼び出しは **null** を戻します。

削除 (delete)
属性はマップから除去されます。

値 (value)
欠落している属性には、指定された値がマップされます。AssemblyLine、属性マップ、および属性レベルの NULL 動作の場合、値は「NULL 動作」ダイアログの「**値**」編集フィールドに設定されます。ここで複数の属性値を指定する場合は、1 行に 1 つずつ値を入力します。システム・レベルおよび構成レベルの設定で rsadmin.attribute.nullBehavior を使用する場合は、rsadmin.attribute.nullBehaviorValue プロパティーも設定する必要があります。

デフォルトの動作 (Default Behavior)
NULL 動作は、上位のレベルから継承する必要があります。例えば、属性レベルは AttributeMap から継承し、AttributeMap は AssemblyLine 設定から継承します。

注: 構成レベルの NULL 動作は、すべてのシステム・レベル設定を指定変更します。また、システム・レベルのデフォルトの動作設定は **delete** を指定するのと同じであり、構成レベルのデフォルトの動作設定は **value** を指定するのと同じです。

ブランチ・コンポーネント

ブランチ・コンポーネントは、AssemblyLine のフローでのその他のコンポーネント (コネクター、スクリプト、関数、AttributeMap、および他のブランチ・コンポーネント) の実行順序に影響します。

概説

ブランチ・コンポーネントには次の 3 つがあります。

- 単純 (単にブランチとも呼ばれる)
- ループ (ループするブランチ)
- Switch (同じ式を共用するブランチ)

ブランチは「フロー」セクションの任意の場所で使用できますが、ワークスペースの「リソース」セクションには、ブランチのライブラリー・フォルダーはありません。

ブランチは、完了するまで実行する必要はありません。すべてのタイプのブランチをプログラマチックに終了させるため、同一のスクリプト呼び出し (`system.exitBranch()`) が使用されます。詳しくは、31 ページの『ブランチ (またはループ、AL フロー) の終了』を参照してください。

3 つのブランチ・コンポーネントのタイプは以下のとおりです。

ブランチ

各タイプのブランチでは AssemblyLine 処理のための代替経路を定義します。これは、「このシチュエーションが発生したらアクションを実行する」というように、最も単純な形式でケースのアクションを定義するものです。「シチュエーションの発生」を定義するため、満たす必要がある条件を設定します。例えば、データ値の比較や、ある操作の結果の検証などです。条件が True の場合、このブランチに接続されたコンポーネントが実行されます。

ブランチにより、IBM Security Directory Integrator サーバー内部のデータに基づいて条件を定義できます。このようなデータには、属性値、パラメーター設定、外部的にアクセス可能なプロパティー、および JavaScript を使用して利用できる情報 (ディスクに対するオペレーティング・システム呼び出しやメモリー使用量など) があります。複数の条件は、「**いずれかと一致**」チェック・ボックスの設定に応じて、ANDed または ORed になります。

ブランチ・コンポーネントのこの最も単純なフォームは、選択可能な 3 つのサブタイプ設定 (IF、ELSE-IF、ELSE) をサポートします。

IF AssemblyLine フロー内のいずれの場所でも使用できます。IF ブランチでは、条件が True の場合に処理する代替トラックが指定されます。ブランチに属するコンポーネントが実行されると、制御はこのブランチの後 の最初のコンポーネントに渡されます。 これを回避するには、ELSE ブランチまたは ELSE IF ブランチを追加するか、または `system.exitBranch()` のスクリプト呼び出しによりブランチを終了する必要があります。

ELSE-IF

IF ブランチまたは ELSE-IF ブランチの直後にのみ使用される点を除き、IF ブランチと同じです。

ELSE

IF ブランチまたは ELSE-IF ブランチの直後でのみ使用できます。ELSE ブランチには条件はありません。ここに属するコンポーネントは、先行する IF ブランチまたは ELSE-IF ブランチが True でない場合にのみ処理されます。また、ELSE インスタンスは常に true と評価されるため、サイクル中に評価される条件はありません。

先に説明したように、ブランチを未完了の段階で終了するには、スクリプト (`system.exitBranch()`) を使用します。

ループ

ループ・コンポーネントは、AssemblyLine にサイクル・ロジックを追加する機能を提供します。ループは、3 種類の操作モード (条件に基づく操作、コネクターに基づく操作、または属性値に基づく操作) に対して構成できます。

条件付き

単純ブランチの場合のように、ループの動作を制御する条件を定義できます。ループは、条件が満たされている限り反復し、満たされなくなると即座に停止します。ループの詳細ウィンドウは、前のセクションで説明した単純ブランチのものと同じです。

コネクター

このメソッドでは、イテレーター・モードまたはルックアップ・モードのコネクターを設定します。項目が戻されるたびにループ・フローを反復します。このループ・タイプの詳細ウィンドウには、コネクターの構成、接続、属性のディスカバー、および入力マップの設定のために必要なタブが表示されます。

「**Init オプション**」というパラメーターを使用して AL に以下のいずれかを指定できます。

- 「**何もしない**」を指定すると、AL サイクル間でコネクターが準備されません。
- 「**初期化および選択/ルックアップ**」を選択すると、AL サイクルごとにコネクターが再初期化されます。
- 「**選択/ルックアップのみ**」では、コネクターの初期化が続行されますが、モードの設定によって、イテレーターの選択またはルックアップをやり直します。

出力マップに類似した機能の「**コネクター・パラメーター**」タブもあり、**work** 属性値から設定されるコネクター・パラメーターを選択できます。

これにより、イテレーター・モードでのループとルックアップ・モードでのループの違いについて認識できます。いずれのオプションも *検索* を実行し、ループに戻される結果セットを作成します。 イテレーター・モードの場合、結果セットはこのコンポーネントのパラメーター設定により排他的に制御されます。ルックアップ・モー

ドの場合、リンク基準を使用して検索ルールまたは一致ルールを定義します。「一致なしの場合」または「複数項目時」などのフックをコーディングする必要がないため、必ずしも 1 つの一致項目のみが戻されるわけではない検索には、この方法が推奨されます。

属性値 作業項目に使用可能な属性を選択すると、その値ごとにループ・フローが実行されます。各値は、2 番目のパラメーターで指定されている新規の作業項目属性のループに渡されます。 このオプションを選択すると、グループのメンバーシップ・リストや電子メールなど、多値の属性の使用が容易になります。

ブランチを未完了の状態で終了するには、スクリプト (`system.exitBranch()`) を使用してループを終了します。

Switch

ブランチおよびループの条件で使用される式とは異なり、Switch 式の結果は、単なる `True` または `False` ではなく複数の値になります。例えば、属性値で Switch を使用したり、他の AssemblyLine または処理から AL が呼び出された場合に要求された操作で Switch を使用したりできます。 Switch コンポーネントにおいて、処理する Switch 式の各定数値に Case を追加します。例えば、作業項目のデルタ命令コードを使用するように Switch をセットアップする場合、使用する Case は「add」、「delete」、および「modify」などの値に対するものとなります。

AL の Switch-Case 構文では、複数の Case を同時にアクティブにできます。IBM Security Directory Integrator は、一連の標準 IF ブランチを検査するのと同様に、各 Case を検査します。以下の例は、複数の Case がどのように動作するかを示しています。

```
work.setAttribute("test","abc");

Switch work.test
  Case startsWith("a"): this is true
  Case contains ("bc"): this is true
  Case length=3: this is true
```

3 つの Switch work.test 式が True となり、Switch の実行がトリガーされます。

Switch-Case を未完了の状態で終了するには、スクリプト (`system.exitBranch()`) を使用します。

ブランチ (またはループ、AL フロー) の終了

ブランチ、ループ、スイッチ、または AL の「フロー」セクションのような組み込みブランチを終了する場合、フック、場合によってはスクリプト・コンポーネントなど、スクリプト記述が可能な場所で、`system.exitBranch()` メソッドを使用します。 パラメーターなし (または空のストリング) で `system.exitBranch()` を呼び出すと、このメソッドを含むブランチが終了し、ブランチ後の最初のコンポーネントでフローが継続します。

次のいずれかを含むストリング・パラメーターを設定してメソッドも指定できます。

予約キーワード (Branch、Loop、Flow、Cycle、AssemblyLine) のいずれか 1 つ (大/小文字を区別しない)

指定すると、このタイプの最初のブランチが切断され、AssemblyLine が逆方向にトレースされます。スクリプト・コードがループ内のブランチにある場合に、`system.exitBranch("Loop")` 呼び出しを実行すると、このメソッドを含むブランチとループの両方が終了されます。予約語である `Flow` を使用すると、フローは AssemblyLine の「フロー」セクションを終了し、応答動作 (サーバー・モードのコネクターの場合)、次の項目で読み取るアクティブ・イテレーター、あるいは AL シャットダウン (エピローグ、...) のいずれかを継続します。 `Cycle` キーワードを使用すると、現行の AL サイクルの終端に制御を渡し、サーバー・モードのコネクターで応答動作を呼び出しません。`AssemblyLine` キーワードを使用すると、AL が停止してシャットダウンします。

`system.exitBranch()` 呼び出しで使用されるその他すべての値は、指定した名前のブランチ/ループを中断します。 そのため、例えば `system.exitBranch("IF_LookupOk")` を呼び出すと、組み込まれている「IF_LookupOk」というブランチまたはループの後にフローが送信されます。任意の名前の AL コンポーネントに制御を渡す `system.skipTo()` とは異なり、`system.exitBranch()` は指定したループ/ブランチの後で処理を続行させます。

ブランチまたはループの名前 (大文字小文字を区別)

スクリプトの呼び出しがネストされるブランチまたはループの名前を渡すと、制御は AL 内の次のコンポーネントに渡されます。この名前のブランチまたはループが見つからない場合は、呼び出し時点から逆方向にトレースすると、エラーが発生します。

ループ・コンポーネントには、*継続* 機能もあります。システム・オブジェクト内の以下のメソッドが使用可能です。

```
system.continueLoop();
system.continueLoop(name);
```

ここで、*name* はループ名を表すストリングです (大/小文字を区別します)。ループ名が指定されている場合、プログラムのフローは、その名前の LOOP コンポーネントに転送されます。

パーサー

パーサーは、読み取りまたは書き込み中の内容の構造を解釈または生成するために、バイト・ストリーム・コンポーネント (ファイル・システム・コネクターなど) とともに使用されます。

構文解析するバイト・ストリームと選択されたパーサーの組み合わせが不適切である場合は、`sun.io.MalformedInputException` が発行される点に注意してください。例えば、「**入力マップ**」タブを使用してファイルをブラウズすると、エラー・メッセージが表示されることがあります。

構成エディターでは、次の 2 つの場所でパーサーを選択できます。

1. バイト・ストリーム・コネクターの「**パーサー**」タブ。
2. 独自に作成したスクリプト (フックやスクリプト・コンポーネントなど)。

個別のパーサーについて詳しくは、「リファレンス」の『パーサー』を参照してください。

文字エンコード変換

Java2 は、内部文字エンコードとして Unicode を使用します。Unicode は、2 バイト文字セットです。AssemblyLine およびコネクター内で処理されるストリングおよび文字は、常に Unicode であると想定されます。ほとんどのコネクターは、文字エンコードを変換するための何らかの手段を備えています。ローカル・システム上のテキスト・ファイルからデータを読み取るときには、稼働しているプラットフォームに依存するデフォルトの文字エンコードの変換方法が、Java2 によって事前に設定されています。

IBM Security Directory Integrator サーバーには、構成ファイルの文字セットを指定する **-n** コマンド行オプションがあり、構成ファイルを新規に作成するときにこの文字セットを使用します。また、この文字セットの指定子をファイルに組み込むため、後でファイルを読み込むときに正しく解釈できます。

一方、テキスト・ファイルからのデータの読み取り、またはテキスト・ファイルへのデータの書き込みを行うときに、ファイルによって情報の文字エンコードの方式が異なる場合がしばしばあります。例えば、パーサーを必要とするコネクターは、通常パーサー構成内の「**文字セット**」パラメーターを受け入れます。このパラメーターは、IANA Charset Registry (http://www.iana.org/assignments/character-sets) によって指定された受け入れ済みの変換テーブルの 1 つに設定されている必要があります。

一部のファイルには、UTF-8、UTF-16、または UTF-32 エンコードの場合、ファイルの最初に Byte Order Marker (BOM) が含まれています。BOM は、`0xFEFF` という文字のエンコードです。これは、使用されるエンコードのシグニチャーとして使用できます。ただし、IBM Security Directory Integrator ファイルのコネクターは、BOM を認識しません。

BOM が含まれるファイルを読み込む場合は、以下のコードをコネクターの「選択前」フックなどに追加する必要があります。

```
var bom = thisConnector.connector.getParser().getReader().read(); // skip the BOM = 65279
```

このコードは、BOM を読み込んでスキップし、パーサーに適切な文字セットを指定したと想定します。

HTTP プロトコルについては、注意が必要です。詳しくは、「リファレンス」の HTTP パーサーの説明にある文字セットのエンコードに関するセクションを参照してください。

独自の Java クラスへのアクセス

ユーザー独自のカスタム Javaクラスには、IBM Security Directory Integrator フレームワーク内部からアクセスできます。ただし、これらが *public* クラスおよびメソッドであることが必要です。これらのライブラリーは、.jar または .zip ファイルとしてパッケージされ、`TDI_install/jars` ディレクトリー内 (ユーザー独自のサブディレクトリー内が望ましい) に格納される必要があります。CLASSPATH 環境変数または Java ランタイム環境拡張フォルダーを使用することもできますが、この 2 つ

の方法はお勧めできません。この 2 つの方法では、ユーザー独自のクラスの内部からクラスを呼び出すことができるのは、ローダーがユーザー独自のクラスよりも前にそのクラスをロードした場合のみです。

構成エディターからサーバーを実行している場合は、構成エディターに TDI_install/jars ディレクトリーおよびサブディレクトリー内の新しいクラスを認識させるために、構成エディターを再始動する必要があります。

jars サブディレクトリー内に .jar ファイルを格納した後で、IBM Security Directory Integrator の内部で参照されるクラスのインスタンスを作成できます。 Java 関数コンポーネントにより、.jar ファイルを開き、.jar ファイルに含まれるオブジェクトをそのメソッドと同様に参照できる点に注意してください。関数を一度呼び出すと、FC は入力スキーマと出力スキーマを準備し、Java 関数が必要とするパラメーターと突き合わせます。

スクリプトからの Java クラスの呼び出しについて詳しくは、 81 ページの『Java クラスのインスタンス化』を参照してください。

構成エディターを使用したクラスのインスタンス化

構成エディターの「**ソリューションのロギングおよび設定**」ウィンドウ内の「Java ライブラリー」フォルダーを使用して、使用するクラスを宣言します。これは、使用するクラスに引数を保持しないコンストラクターがある場合にのみ有効です。このコンストラクターは、通常はデフォルト・コンストラクターですが、そうでない場合もあります。

クラス・オブジェクトを追加する場合は、「**追加...**」をクリックし、スクリプト・オブジェクト名 (対象の Java クラスのインスタンスであるスクリプト変数の名前) と Java クラス名の 2 つのパラメーターを指定してください。例えば、スクリプト・オブジェクト名 *mycls* と Java クラス *my.java.classname* を指定できます。*mycls* オブジェクトは、グローバル・プロローグの実行前に定義されている任意の AssemblyLines で使用できます。

注: これにより、AssemblyLine が実行されるごとにこのオブジェクトがインスタンス化されます。この方法ではなく要求時のインスタンス化を希望する場合は、次のセクションを参照してください。

クラスのランタイム・インスタンス化

クラスを特定の実行ポイントでインスタンス化する場合、または引数を保持しないコンストラクターを持たないクラスの場合は、ランタイム中にクラスをインスタンス化する必要があります。 例を示します。

```
crytoLib = new com.acme.myCryptoLib();
```

AssemblyLine フローおよびフック

AssemblyLine には、データ・フローの作成とデプロイメントの迅速化に役立つ、組み込みの自動動作が用意されています。この自動動作は、「リファレンス」のフロー・ダイアグラムに詳述されています。また、コネクターおよび関数にも独自の動作があり、それらもフロー・ダイアグラムに示されています。コネクターの動作はモード設定によって異なることに注意してください。

これらの組み込みロジック・フローの全体は、多数の*中間地点* となります。ここで独自のスクリプト・ロジックを追加して組み込み動作を拡張したり、完全に指定変更できます。これらの中間地点は「フック」と呼ばれ、AssemblyLine 自身のものと同様にすべてのコネクターおよび関数の「フック」タブでカスタマイズ可能です。

実行中の AssemblyLine に特定のフックが適用可能かどうかに従い、フックを使用可能または使用不可にできます。フックを使用不可にしても、そのフックがパーツとなっているコネクターではフックの継承は中断されません。

AssemblyLine が起動されると、3 つのフェーズ (開始、データ・フロー、シャットダウン) が実行されます。 開始フェーズでは、コンポーネントを初期化前に再構成するための「プロローグ」フックを使用できます。 データ・フロー・フェーズでは、AssemblyLine に送られた各作業項目が処理のためにフロー・コンポーネントに渡されます。

最後に、シャットダウン・フェーズでは「エピローグ」フックを使用して、エラー状況の確認およびレポート、次回 AL 開始時のための状態データの保管など、ジョブ終了作業を実行できます。

開始フェーズ

この時点では、サーバーは AssemblyLine をロードして実行するよう指定されています。サーバーは、構成ファイルに保管された青写真を使用して AL をセットアップします。 TaskCallBlock (TCB) が AssemblyLine に渡されていた場合、その内容が評価されます (結果として AL コンポーネント・パラメーターの変更が生じる場合があります)。 この時点で、「プロローグ」フック・フローが開始されます。

グローバル・プロローグ

最初に、グローバル・プロローグが評価されます (グローバル・プロローグが定義されている場合)。グローバル・プロローグは、プロジェクト内のリソース・フォルダー内にあるスクリプトで、AssemblyLine (「**ソリューションのロギングおよび設定**」ウィンドウ) に組み込まれています。通常、これは AL の開始時に実行するスクリプトを選択することで、AssemblyLine の「AssemblyLine 設定」ウィンドウで行われます。すべてのグローバル・プロローグの完了後、AL の「プロローグ」フックが呼び出されます。

AL の「プロローグ」フック (初期化前)

最初に、AssemblyLine の「プロローグ - 初期化前」フックが呼び出されます。その後、「開始時」に初期化するように構成されたすべてのコネクターおよび関数は、それぞれの初期化フェーズを実行し、各「プロローグ」フックの呼び出しも行います (次の説明を参照)。

コネクター/関数の初期化

初期化シーケンスは、初期化が「開始時」に設定されたコネクターおよび関数ごとに実行されます。これらの開始順序は、AssemblyLine における順序での定義によって決まります。各コネクターまたは関数について、フローは以下のようになります。

1. コンポーネントの「**プロローグ - 初期化前**」フックが呼び出されます。

2. コンポーネントが開始されます。例えば、その基盤となるデータ・ソース、ターゲット、または API への接続です。
3. イテレーター・モードのコネクターの場合、「**プロローグ - 選択前**」フックが処理されます。コネクターは項目の選択を実行します。 SQL SELECT または LDAP 検索の実行など、データ・ソースの特定の呼び出しをトリガーします。
4. イテレーターの場合、「**プロローグ - 選択後**」フックが評価されます。
5. 「**プロローグ - 初期化後**」フックが呼び出され、シーケンスが終了します。

コネクターの初期化に失敗した場合、AssemblyLine フローは「**プロローグ - エラー発生時**」フックに移ります。このフックでは、初期化エラーを処理できます。

再接続機能により、セットアップまたはデータ・アクセス中にエラーが発生した場合にコネクターが自動的に接続の再確立を試行するよう構成できます。これらの設定は、コネクターの「**接続障害 (Connection Failure)**」タブにあります。

注: スクリプト・コネクター (JavaScript を使用してインプリメントされたコネクター) については、この段階で評価されます。この結果、要求されるコネクター関数が登録され、初期化コードが実行されます。

AL の「プロローグ」フック (初期化後)
AssemblyLine の「**プロローグ - 初期化後**」フックが実行されます。このフックの完了は、開始フェーズの終了およびデータ・フロー・フェーズの開始を意味します。

データ・フロー・フェーズ

AssemblyLine の「サイクル開始 (Start of Cycle)」フック
このフックは、供給コンポーネントまたはフロー・コンポーネントの前のすべてのサイクル開始時に呼び出されます。

AssemblyLine サイクル
制御はフロー内の最初のコンポーネントに渡されます。通常は、「フィード」セクションのサーバーまたはイテレーター・モードのコネクターです。

AssemblyLine に 1 つ以上のイテレーターがある場合、最初のイテレーターがその結果セットから次の項目を検索し、属性を作業項目にマップすることでサイクルを開始します。その結果となる作業項目は「フロー」セクションのコンポーネントに渡され、CE で表示されるとおりリストの最上部から開始します。

サーバー・モードのコネクターの場合、「リスナー」処理が開始され、着信するクライアント接続を待機します。接続要求が検出されると、コネクターは自身のクローンを作成し、接続を受け入れて自身 (クローン) をイテレーター・モードに切り替え、処理のためにデータをクライアントから「フロー」セクションに送ります。いずれの場合でも、イテレーターを使用して作業項目をフロー・コンポ

ーネントに送ります(この間、元のサーバー・モードのコネクターは、追加の着信接続要求を待機します)。

AssemblyLine にイテレーター・モードのコネクターとサーバー・モードのコネクターのどちらもない場合、別の呼び出しプロセスにより送られる初期作業項目 (IWE) を処理するために通常は使用する、1 回限りの AssemblyLine が必要になります。

サイクルの終わり

最後のフロー・コンポーネントが実行されると、以下の 3 つのうち 1 つが発生します。

- 現在の作業項目がイテレーターからの項目である場合、制御はイテレーターに戻されて、そのソースから次の項目が取得されます。
- サーバー・モードのコネクターの場合、クライアントに対して応答が通知されます。
- 手動サイクル・モードで呼び出された AL の場合、スレッドが呼び出し元に戻され、結果へのアクセスが可能になります。

最後にスクリプトを挿入することで特定のフックを AssemblyLine に追加できますが、この時点では特定のフックはありません。

データの終わり

データの終わりはイテレーター・モードのフックで、入力データ・セットの終わりに達した場合に呼び出されます。 この時点で、制御は次の供給コネクターに渡されるか、または AssemblyLine がシャットダウン・フェーズに移行します。

シャットダウン・フェーズ

この時点で、AL 処理は正常に完了しているか、エラーにより異常終了しています。

AssemblyLine の「エピローグ - クローズ前」フック

「**エピローグ - クローズ前**」と呼ばれる AssemblyLine フックが処理されます。

コネクター/関数のクローズ・フロー

各コネクターおよび関数の「エピローグ」フックが、次のように構成エディターでの表示順に従って呼び出されます。

1. 「**クローズ前**」フックが呼び出されます。
2. クローズ操作が実行されます。例えば、接続のクローズ、または API コールバックのリリースです。
3. 「**クローズ後**」フックが呼び出されます。

AssemblyLine の「エピローグ: クローズ後」フック

最後に、AssemblyLine の「**エピローグ - クローズ後**」フックが実行されます。

サーバー・モードのコネクターのセットアップ

サーバー・モードのコネクターは開始すると、イベントの listen モードに移ります。イベントを受信すると、AL のクローンを作成してから、その他のイベントの待機を再開します。クローンでは、その間、コネクターが自身をイテレーター・モードに切り替え、フィード・リストにある次のコンポーネ

ントに制御を渡します。このプロセスにより、サーバー・モードの複数のコネクターをアクティブにしておくことが可能になり、同時にデータをフローに送ることが可能になります。例えば、それぞれ異なるポートを listen し、1 つの AL にデータを送る (フィードする) サーバー・モードの HTTP サーバー・コネクターが複数存在する場合があります。サーバー・モードのコネクターは、AssemblyLine 構成に含まれますが、別個のプロセス (スレッド) として実行されます。

このモードのコネクターを評価する追加のフック・セットがあります。着信接続を処理するサーバー・モード機能に固有のフックには、次のものがあります。

接続受け入れ前
このフックは、コネクターが listen モードに移行する前に呼び出されます。

接続受け入れ後
接続が受け取られると、このフックが呼び出されます。この時点で使用可能なデータはありません。 着信イベント情報を確認するには、「**GetNext 後**」または「**GetNext 成功**」などのイテレーター・フックを使用します。

接続受け入れエラー
このフックは、サーバー・モードのフックのいずれかにエラーが発生した場合、またはイベントの listen 中にデータ・ソースからエラーを受信した場合に実行されます。

- 既に説明したとおり、イテレーター・モード (7 ページの『AssemblyLine 内の複数のイテレーター』を参照) のコネクターが複数ある場合、それらのコネクターは、構成に表示される順序 (上から下) でスタックされます。例えば、AssemblyLine に **a** と **b** の 2 つのイテレーターがある場合、まず **a** が、項目を戻さなくなるまで呼び出されます。その後、呼び出し先が **b** に切り替えられます。
- イテレーター・モードのコネクターが存在せず、初期作業項目 (IWE) が AssemblyLine の開始時に (別の AssemblyLine からの呼び出しなどで) 提供されず、 AssemblyLine またはコネクターの「プロローグ」フックによっても作業項目が作成されなくても、 AssemblyLine は 1 回実行されます。

最後に、「**シャットダウン要求**」フックについて説明します。このフックには、外部からのシャットダウン要求によって AssemblyLine が正常にクローズされる場合 (クラッシュしたのではない場合) に実行されるコードを指定することができ、これによって安全にシャットダウンが実行されるようにすることができます。

システム・オブジェクトからは、特殊な関数を使用して、現行の作業項目のスキップや再試行のほか、コネクターのスキップオーバーなどを行うことができます。詳しくは、 39 ページの『AssemblyLine のフローの制御』を参照してください。

クリティカル・エラーによる終了およびクリーンアップの処理

多くのメソッドでは、 IBM Security Directory Integrator の内部エラーのほか、IBM Security Directory Integrator コネクター、パーサー、関数コンポーネントなどで発生したエラーを検出および処理できます。このようなメソッドには、以下のものがあります。

- エラー・フック。これには、エラー処理のための JavaScript コードを書き込むことができます。IBM Security Directory Integrator ユーザーは、このメソッドにアクセスできます。『AssemblyLine のフローの制御』も参照してください。
- Java の try-catch-finally ブロック。これは、小規模の障害がサーバーを中断しないように、またすべてのエラーが適切に処理されるようにします。このようなブロックは、IBM Security Directory Integrator のコア・サーバー・クラスに既に組み込まれています。

JVM シャットダウン・フック機能は、サーバーの信頼性を向上させます。 Java シャットダウン・フックを使用すると、Control-C を押した後、または System.exit など、その他の理由で JVM がシャットダウンしているときに、1 つのコードで何らかの処理を実行できます。

ユーザーは、JVM がシャットダウンしているときに始動する外部プログラムを指定できます。この外部プログラムは、JVM シャットダウン・フック内から開始されます。この外部プログラムは、`global.properties` ファイルまたは `solution.properties` ファイル内のオプション・プロパティーを使用して構成します。

```
jvm.shutdown.hook=<external application executable>
```

シェル・スクリプトおよびバッチ・ファイルも、このプロパティーの値として指定できます。

JVM シャットダウン・フックが呼び出された場合、JVM の終了を防ぐことはできません。 ただし、外部プログラムを実行すれば、カスタマイズ可能な操作を実行できます。例えば、 IBM Security Directory Integrator サーバーが終了したというメッセージの送信や、クリーンアップ操作の実行です。必要であれば、新規サーバーの再始動も可能です。

AssemblyLine のフローの制御

フックは、IBM Security Directory Integrator の組み込み自動化動作におけるプログラマブルな中継点であり、ここでユーザー独自のロジックを実行させることができます。

フックは、AssemblyLine、コネクター、および関数コンポーネント内にあります。例えば、AssemblyLine のいくつかの部分を完全にスキップまたは再始動する場合、通常はコネクターのフックの内部からこれを実行します。

注: 次の構成を使用して、ブランチ・コンポーネントまたはループを終了することもできます。

system.ignoreEntry()
現在のコネクターを無視し、次のコネクターで既存のデータ項目の処理を続行します。

system.skipEntry()
項目を完全にスキップ (ドロップ) し (現在のサイクルを打ち切り)、AssemblyLine の先頭に制御を戻し、現在のイテレーターから次の項目を取得します。

system.exitFlow()
現在の項目の以降の処理をドロップし、サイクルの終わりのロジックを実行します。例えば、イテレーター状態キーを保存するようにコネクターが構成されている場合は保存し、AssemblyLine の先頭に制御を戻し、現在のイテレーターから次の項目を取得します。

system.restartEntry()
AssemblyLine の先頭から再始動し、現在のイテレーターに現在の項目を強制的に再使用させます。

system.skipTo(String name)
指定されたコネクターまでスキップします。

system.abortAssemblyLine(String reason)
指定されたエラー・メッセージを表示し、AssemblyLine 全体を異常終了します。

注: エラー・フック内に何らかのコードを入れ、現在の AssemblyLine または EventHandler を終了しない場合は、エラー・フックに到達するまでの方法とは無関係に処理が続行されます。つまり、スクリプト内に構文エラーがあっても無視されます。したがって、エラーの原因を確認する場合は、エラー・オブジェクトをチェックしてください。

フック内で他のコードが実行されないため、上記の各メソッドは、goto 文であると見なすことができます。 例を示します。

```
system.skipEntry(); // Causes the flow to change
// This next line is never executed.
task.logmsg("This will never be reached");
```

注: エラー・フックが存在しない場合と空の場合を構成エディターで簡単に見分けられるとは限りませんが、この 2 つの間には違いがあります。空の エラー・フックがあると、システムは、フックが呼び出される原因となったエラー条件をリセットし、その後でサーバーが処理を続行しますが、*一方存在しない* フック、つまり未定義のフックがあると、システムは、デフォルトのエラー処理 (通常は AssemblyLine の打ち切り) を実行します。

式

IBM Security Directory Integrator には、v.6 と互換性のある式機能が用意されています。これを使用すると、実行時にパラメーターやその他の設定を計算でき、ソリューションを動的に構成することができます。この機能は、以前のバージョンにあったプロパティー処理が拡張されたものです。

単純な外部プロパティー参照のサポート (旧バージョンとの完全な互換性) に加え、式は、AssemblyLine またはコンポーネントの初期化中および実行中に AL およびコンポーネントの構成設定を行うことをさらに強力に支援します。式は、条件およびリンク基準と同様に属性マップにも使用可能で、これまでソリューションの動的な構成に必要であったスクリプトを大幅に軽減します。 IBM Security Directory Integrator は、これらの式の作成を容易にする式エディターを提供します。

式機能は、標準的な Java `java.text.MessageFormat` クラスにより提供されるサービス上に構築されています。 MessageFormat クラスは、強力な置換およびフォーマッ

トの機能を提供します。このクラスとその機能の概要を説明するオンライン・ページへのリンクは、http://docs.oracle.com/javase/1.6.0/docs/api/java/text/MessageFormat.html です。

注: このセクションで説明する MessageFormat ベースの式は、IBM Security Directory Integrator v.6 では、パラメーター置換の基本でした。しかし v.7 のベスト・プラクティスは、代わりに拡張 (JavaScript) 式を使用することです。

上記のクラスで説明した機能に加えて、IBM Security Directory Integrator では、式で使用できるさまざまなランタイム・オブジェクトが用意されています。ただし、一部のオブジェクトの可用性はランタイムの状態 (例えば、*conn* または *current* の定義の有無、*error* 項目) によって異なります。式の構文は、指定された項目オブジェクト内の属性またはコンポーネントの特定のパラメーターのように、これらのオブジェクトの情報にアクセスするための省略表現を使用します。

表 2. スクリプト・オブジェクト、その使用法および可用性

IBM Security Directory Integrator の参照	**値**	**可用性**

work.*attrname[.index]*	現行の AssemblyLine の *work* 項目です。
	
	オプションの *index* は、属性の *n* 番目の値を参照します。指定されない場合は最初の値が使用されます。
	
	以下の拡張属性マップは、
	
	`ret.value = work.getString ("givenName") + " " + work.getString("sn");`
	
	次のように簡潔に表現できます。
	
	`{work.givenName} {work.sn}`
		AssemblyLine

conn.*attrname[.index]*	現行の AssemblyLine の *conn* 項目です。 オプションの *index* は、属性の *n* 番目の値を参照します。指定されない場合は最初の値が使用されます。	属性マッピング中の AssemblyLine
current.*attrname[.index]*	現行の AssemblyLine の *current* 項目です。 オプションの *index* は、属性の *n* 番目の値を参照します。指定されない場合は最初の値が使用されます。	変更のために属性マッピング中の AssemblyLine

表 2. スクリプト・オブジェクト、その使用法および可用性 (続き)

IBM Security Directory Integrator の参照	**値**	**可用性**
config.*param*	現在のコンポーネント AL の構成オブジェクト。また、*config* がコネクター、パーサー、または関数のパラメーターで使用される場合は、そのコンポーネントの構成オブジェクト・インターフェース (JDBC コネクター、XML パーサーなど) を参照します。 getParam() または setParam() への呼び出しのように、*param* はパラメーター自体の名前です。例えば、JDBC コネクターの場合は以下のように参照できます。 `{config.jdbcSource]`	AssemblyLine EventHandler コネクター パーサー 関数コンポーネント

alcomponent.*name.param*	指定された AssemblyLine コンポーネントのコンポーネント・インターフェース・パラメーター値です。
	
	name は AssemblyLine コンポーネントの名前です。
	
	param は、*name* オブジェクトのパラメーター名です。
	
	そのため、以下の式は、
	`{alcomponent.DB2conn.jdbcSource}`
	
	次のスクリプト呼び出しと同等です。
	`DB2conn.connector.getParam ("jdbcSource");`
		AssemblyLine

property[:storename].name property[:storename/bidi].name	*TDI プロパティー* 参照。 オプションの storename は、特定のプロパティー・ストアを対象とします。storename が指定されない場合、デフォルト・ストアが使用されます。 *name* はプロパティー名です。 *bidi* が使用された場合は、パラメーター値が設定され、参照されたプロパティー・ストアに呼び出しが転送されます。*bidi* が使用された場合は、他の置換パターンまたはテキストは許可されません。	常時

表 2. スクリプト・オブジェクト、その使用法および可用性 (続き)

IBM Security Directory Integrator の参照	値	可用性
JavaScript<<EOF script code ... // Must contain “return” EOF **注:** v.6 の構文。代わりに、式エディターの「拡張 (JavaScript)」オプションを使用。	式について値を生成するために使用される組み込み スクリプト・コードです。このスクリプトは、値を戻す必要があります。 ここで使用される「EOF」テキストは、JavaScript スニペットを終了する任意のストリングです。 JavaScript は、EOF ストリングを含む単一行を検出するまで収集されます (EOF フラグが設定されない場合については以下の説明を参照)。 組み込み JavaScript は AssemblyLine のスクリプト・エンジン・インスタンスを使用して評価されるため、そうでない場合にはスクリプト記述に使用される変数にもすべてアクセスできます。 **注:** 複数行 (リンク基準、マップ内の属性名など) をサポートしないため、必要な EOF 行を使用できない入力フィールドに対しては、簡略形式で JavaScript を追加することができます。 `{JavaScript return work.givenName + " " + work.surName}`	常時

式に組み込まれている JavaScript で、AssemblyLine のスクリプト・エンジンにアクセスできます。そのため、AssemblyLine 内の他の場所に定義されたスクリプト変数にもアクセス可能です。このセクションの表に明確にリストされていない変数またはオブジェクトを参照する場合、式エバリュエーターは AL のスクリプト・エンジンを使用して、その変数またはオブジェクトがそこに定義されているかどうかを検査します。

コンポーネント・パラメーターにおける式

コンポーネント・パラメーターに使用された場合、以下のオブジェクトは特殊な意味を持ちます。

表 3. 式に使用できる特殊オブジェクト

オブジェクト(O)	値
config	コンポーネントのインターフェース構成オブジェクト。
mc	構成インスタンスの MetamergeConfig オブジェクト (`config.getMetamergeConfig()`)。
work	AssemblyLine の作業項目。
task	AssemblyLine オブジェクト。

表名のパラメーターが「Accounts」に設定された JDBC コネクターを例として使用します。この場合、**SQL SELECT** パラメーター・ラベルをクリックするか、このフィールドの隣の「**パラメーター値のダイアログを開きます**」ボタンをクリックして、

「**テキストの置換**」を選択し、これをダイアログの下部にある大きなテキスト・フィールドに入力します。

```
select * from {config.jdbcTable}
```

これは表名のパラメーターを取得し、次の SQL SELECT ステートメントを作成します。

```
select * from Accounts
```

また、さらに上級の使用法として、SQL SELECT パラメーターに対して以下を試行してください。

```
SELECT {JavaScript<<EOF

 var str = new Array();
 str[0] = "A";
 str[1] = "B";
 return str.join(",");
EOF

} FROM {property:mystore.tablename} WHERE A = '{work.uniqueID}'
```

組み込み JavaScript は、「A,B」の値を戻します。この値は、残りの式を完了するために使用されます。 *tablename* プロパティーが「Accounts」に設定されたプロパティー・ストア *mystore* があり、作業項目の *uniqueID* 属性の値が「42」である場合、最終的な結果は次のようになります。

```
SELECT A,B FROM Accounts WHERE A = '42'
```

これを評価した結果は、CE には表示されません。単に中括弧を入力しても、パラメーター値について式の評価は実行されません。代わりに、パラメーターに式を結合する方法は 2 つあります。

1. パラメーター入力フィールドで、フィールドの隣の「**パラメーター値のダイアログを開きます**」ボタンを押し (または「パラメーター」ラベルをクリックし)、「**テキストの置換**」を選択して、「式」ダイアログを開きます。このダイアログの大きなテキスト・フィールドに式を入力します。「**OK**」をクリックすると、式が入力されます。
2. パラメーター入力フィールドに特殊なプリアンブル (@SUBSTITUTE) を手動で入力し、続けて式を入力する。例を示します。

   ```
   @SUBSTITUTEhttp://{property.myProperties:HTTP.Host}/
   ```

   注:

   - 式を直接入力するこの最後の方法は非推奨です。代わりに、式エディターを使用してください。
   - ファイル・コネクターのファイル・パスの値として「**テキストの置換**」を選択し、{work.fullPath} を入力した場合、次のエラーが発生します。「CTGDIC114E パラメーター 'ファイル・パス' は必須です」。構成エディターは、ユーザーが指定したテキストに置換を適用した結果を表示する必要があるため、これは想定される結果です。この場合、work オブジェクトは実行中の AssemblyLine にしか定義されていないため、work オブジェクトが存在しません。したがって、結果は空ストリングになります。このパラメーターでは空ストリングはエラーのため、エラー・メッセージが表示されます。ただし、

AssemblyLine が実行中の場合、work オブジェクトが存在する可能性があるため、必要なパラメーター・ストリングが評価により作成される場合があります。

リンク基準における式

リンク基準における式は、事前定義のオブジェクトの類似リストを提供します。繰り返しますが、AssemblyLine のスクリプト・エンジンに現在定義されたほかのオブジェクトまたは変数にアクセスすることも可能です。

表 4. リンク基準における式で使用する事前定義オブジェクト

オブジェクト(O)	意味
config	コンポーネントのインターフェース構成オブジェクト
mc	構成インスタンスの MetamergeConfig オブジェクト (config.getMetamergeConfig())
work	AssemblyLine の作業項目
task	コンポーネント自体、または指定されたコンポーネント
alcomponent	コネクターまたは関数コンポーネント

例えば、コネクターに対してリンク基準をセットアップして、突き合わせに使用する属性が実行時に判別されるようにします。 作業項目の標準データ属性に加えて、ストリング値が「uid」である *matchAtt* 属性もあります。この場合、リンク基準で使用される以下の式、

```
{work.matchAtt}  EQUALS {work.uid}
```

は、次の式と同等です。

```
uid  EQUALS $uid
```

ブランチ、ループ、およびスイッチ/ケースにおける式

ここで示す式オブジェクトのリストは、リンク基準のものと類似しています。

表 5. ブランチ・コンポーネントにおける式で使用する事前定義オブジェクト

オブジェクト(O)	意味
config	コンポーネントのインターフェース構成オブジェクト
mc	構成インスタンスの MetamergeConfig オブジェクト (config.getMetamergeConfig())
work	AssemblyLine の作業項目
task	AssemblyLine
alcomponent	コネクターまたは関数コンポーネント

式は、属性名および条件のオペランドの両方に使用できます。また、Switch コンポーネントおよび Case コンポーネントの構成にも式を使用できます。

式を使用したスクリプト

式は、JavaScript コードから直接使用することもできます。新規 ParameterSubstitution クラスを使用して式を作成する例を以下に示します。

```
var ps = new com.ibm.di.util.ParameterSubstitution("{work.FullName} -> {work.uid}");

map = new java.util.HashMap();
```

```
map.put("mc", main.getMetamergeConfig());
map.put("work", work);

task.logmsg(ps.substitute(map));
```

JavaScript コードから作成される式は、テスト AssemblyLine で何度か繰り返して実行されると、以下のログ・メッセージを発行します。

```
14:35:29  Patty S Duggan -> duggan
14:35:29  Nicholas P Butler -> butler
14:35:29  Henri T Deutch -> deutch
14:35:29  Ivan L Rodriguez -> rodriguez
14:35:29  Akhbar S Kahn -> sahmad
14:35:29  Manoj M Gupta -> gupta
```

項目オブジェクト

IBM Security Directory Integrator を理解するには、データがシステム内で保管およびトランスポートされる仕組みを知ることが重要です。これは、*項目* と呼ばれるオブジェクトを使用して実行されます。項目オブジェクトは、任意の数の属性 (なし、1 つ、または複数) を保持できる「Java バケット」と考えることができます。

IBM Security Directory Integrator では、属性もバケットのようなオブジェクトです。各属性にはゼロ個以上の*値* を組み込むことができます。この値は、接続されたシステムから読み取られる (または接続されたシステムに書き込まれる) 実際のデータ値です。属性値は Java オブジェクトでもあり、ストリング、整数、タイム・スタンプなどが使用できます。そのいずれも、このデータ値のネイティブ・タイプと一致する必要があります。 1 つの属性は、異なるタイプの値を容易に保持できます。ただし、多くの場合、1 つの属性値はほとんどのデータ・ソースにおいて同じタイプとなります。

この*項目/属性/値* パラダイムは、Lightweight Directory Access Protocol (LDAP) ディレクトリー項目の概念と見事に一致しますが、これは、ファイル (IBM Lotus® Notes 文書やネットワーク上の HTTP ページ) 内のレコードと同様、データベース内の行が IBM Security Directory Integrator 内部でどのように表現されるかを示すものでもあります。IBM Security Directory Integrator が処理する任意のソースからのデータはすべて、属性およびその値とともに項目オブジェクトとして内部的に保管されます。

IBM Security Directory Integrator の以前のバージョンとは対照的に、v7.0 からは、AssemblyLine でも、AssemblyLine を構成する一部のコンポーネントでも、階層項目オブジェクトがサポートされています。項目オブジェクトが拡張され、階層データを処理するために便利な方法が複数用意されていますが、デフォルトでは非表示になっており、明示的に有効にするか、階層機能を必要とするコンポーネントとともに使用する場合にのみ、表示されます。また、org.w3c.dom.Document が実装され、階層の最上位ノードになっています。これについて詳しくは、52 ページの『階層項目オブジェクトの操作』を参照してください。

一部の項目オブジェクトは、IBM Security Directory Integrator により作成および保守されます。最も目立つインスタンスは*作業項目* と呼ばれ、AssemblyLine (AL) に

おいて主要なデータ・キャリアとして動作します。これは AL へのデータ・トランスポートに使用されるバケットで、あるコンポーネントから次のコンポーネントへと引き渡されます。

作業項目は、事前登録の変数 *work* を使用してスクリプト記述する際に利用可能で、これによって、AssemblyLine (およびその値) により処理されている属性に直接アクセスすることができます。さらに、作業項目により伝送されるすべての属性は、構成エディターの「**作業属性**」ヘッダーの下に表示されます。このヘッダーは、AssemblyLine の「AssemblyLine エディター」ウィンドウの「属性マップ」域にあります。

項目タイプ

多くのデータ・オブジェクトは、項目データ・モデルに従う AssemblyLine に配置されます。これには、以下のものがあります。

Work 前述のとおり、AssemblyLine のコンポーネント間を移動し、データを運ぶ項目です。事前登録の変数名は *work* で、ほとんどの場所でスクリプト記述に使用できます。[6].

Conn 項目に似たオブジェクトで、コネクターは、接続されたシステムと AssemblyLine の間の仲介としてこれを使用します。データとそのサブセットは、この後で作業項目で使用できるようになります。データを Conn と Work の間で移動するプロセスは、属性マッピングと呼ばれます。その事前登録の変数名は *conn* で、コネクターおよび関数コンポーネントに含まれる多くのフック内部でのスクリプト記述に使用できます。

Current®
この項目に似たオブジェクトは、更新モードのコネクターに含まれる特定のフック内で使用でき、接続されたシステムのデータを保持してから、そのシステムに更新を適用します。その事前登録の変数名は *current* です。

Error 項目に似たこのオブジェクトは、一定のエラー条件が発生してそれに関連するエラー・フックが呼び出された場合に、コンポーネント内の特定のフックにのみ存在します。ここには、スローされた実際の例外に関する情報が、場合によっては追加の変数やデータとともに含まれ、それによってエラーの原因を正確に特定することができます。その事前登録の変数名は *error* です。

「リファレンス」のコネクター・フロー・ダイアグラムでは、上記のオブジェクトのうち、どの環境でどのオブジェクトを使用できるかが示されています。

---

6. 処理の対象となる AssemblyLine が初期作業項目で呼び出されない場合、*work* オブジェクトは「プロローグ」フックの後 でしか使用できません。「プロローグ」フックでは、次のようなコードが使用できます。

```
if (work != null) {
 // An Initial work Entry has been provided, we can get values from there
 .... some code
} else {
 // No initial work Entry has been provided
 ... some other code
}
```

関連情報

50 ページの『内部データ・モデル: 項目、属性、値』

第 2 章 IBM Security Directory Integrator のスクリプト記述

IBM Security Directory Integrator は、柔軟性の高いエンジンをユーザーに提供します。このエンジンをカスタマイズするには、構成エディターのユーザー・インターフェース・コントロールを使用するか、またはカスタム・ロジックのスクリプトを記述します。ユーザー・インターフェース・コントロールを使用すると、データ・フローをより全体的なレベルで制御できます。スクリプトを記述する場合、あらゆるレベルでデータ・フローのほとんどの側面 (標準の IBM Security Directory Integrator 処理の指定変更を含む) を制御できます。システム・オブジェクトには、1 つの AssemblyLine 項目を反復処理し、コネクターをスキップし、新規 AssemblyLine を開始するための特殊機能があります。このカスタム・ロジックのインプリメントに使用されるスクリプト言語は JavaScript です。

IBM Security Directory Integrator には、統合ソリューションのフレームワークとともに即座に使用できるツールが提供されています。ただし、ほとんどの小さなマイグレーション・ジョブ以外では、JavaScript を作成して、製品組み込みの動作をカスタマイズまたは拡張する必要があります。

IBM Security Directory Integrator は Pure Java です。 IBM Security Directory Integrator でのコマンドの実行、コンポーネントやオブジェクトの使用、あるいはフローでのデータ操作を行うときには、常に Java オブジェクトを使用していることになります。 IBM Security Directory Integrator では、IBM Java バージョン 7.0.4 を使用しています。

それに対して、ユーザーのカスタマイズは JavaScript で行われます。表面的には類似していても、根本的に異なるこの 2 つのプログラム言語の組み合わせにより、詳細な確認が必要になります。

JavaScript を使用した経験は大いに役立ちます。このマニュアルに示されている例をお読みになることも、このような経験の一部となります。ただし、このマニュアルでは JavaScript 自体についてではなく、IBM Security Directory Integrator での JavaScript の適用について説明しています。JavaScript の参考資料は、独自に入手してください。

JavaScript については、市販のリファレンス・ガイドも多数あり、またネット上にも資料やチュートリアル、実例などが公開されています。ただし、Web 上の JavaScript に関する内容のほとんどが、HTML コンテンツの美化と自動化に関するものであるため注意してください。 理解しておく必要があるのはコア言語自体です。コア言語の説明については、http://devedge-temp.mozilla.org/library/manuals/2000/javascript/1.5/guide/index.html を参照してください。

このサイトには、HTML 形式の資料をダウンロードしてローカルにインストールすることができる便利なリンクも用意されています。「*The Definitive JavaScript Guide*」第 4 刷 (David Flanagan (O'Reilly) 著) は、JavaScript に関する素晴らしい手引書です。

すべての IBM Security Directory Integrator オブジェクトとソリューション内のデータ値は Java オブジェクトの形式であるため、Java に関する Javadoc も必要になります。これらの資料は、URL http://docs.oracle.com/javase/1.6.0/docs/api/index.html からオンラインで使用できます。

J2SE 資料自体は、http://docs.oracle.com/javase/1.6.0/docs/index.html にあります。

スクリプト記述が必要になるのは、AssemblyLine にカスタム処理を追加する必要がある場合です。スクリプト記述が役立つ状況としては、次のタスクがあります。

属性の操作または計算
1 つ以上の入力属性に基づいて出力属性の値を計算する必要がある。

データ・フィルタリング
特定の基準セットに一致する項目のみを処理する。

データの整合性または妥当性の検査
無効なデータ値を報告または訂正する必要がある。

フロー制御
使用しているコネクターの更新操作を指定変更する。

初期化 AssemblyLine の開始前に一部の初期化プロシージャーを実行する。

上記の状況 (および上記で説明されていないその他の多くの状況) では通常、スクリプト記述が必要です。

例

IBM Security Directory Integrator インストール済み環境の `examples/scripting` サブディレクトリーを参照してください。

内部データ・モデル: 項目、属性、値

IBM Security Directory Integrator コンポーネントは、接続システムの情報にアクセスし、システム固有のタイプのデータを Java オブジェクトを使用したシステム内部の表現に変換します。出力時には、コンポーネントは、内部データ・モデルからターゲット・システムのネイティブ・タイプへと逆に変換します。AssemblyLine 間でデータの受け渡しをする際にも、これと同じ内部表現が使用されます。したがって、IBM Security Directory Integrator の内部データ・モデルの仕組みを理解することが非常に重要です。

コンポーネントがデータ値を受け取るところを詳細に確認すると、対応する IBM Security Directory Integrator の *属性* オブジェクトが、読み込まれる属性の名前を使用して作成されています。そのデータ値自体 (多値属性の場合は複数値) が適切な Java オブジェクト (`Java.lang.String` または `java.sql.Timestamp` など) に変換され、属性に割り当てられます。IBM Security Directory Integrator API 資料を参照すると、属性オブジェクトには多数の有用なメソッド (`getValue()`、`addValue()`、および `size()` など) があることがわかります。これにより、属性値をスクリプトから直接作成、列挙、操作することが可能です。必要に応じて新規属性オブジェクトをインスタンス化することもできます。これを、次の属性マップの例 (ディレクトリーの objectClass 属性の拡張マッピング) に示します。

```
var oc = system.newAttribute( "objectClass" );

oc.addValue( "top" );
oc.addValue( "person" );
oc.addValue( "organizationalPerson" );
oc.addValue( "inetOrgPerson" );

ret.value = oc;
```

属性自体が*項目オブジェクト* というデータ・ストレージ・オブジェクトに収集されます。この*項目* は、システム内の基本データを搬送するオブジェクトであり、IBM Security Directory Integrator では重要な項目オブジェクトをスクリプト変数として登録することにより、これらの項目オブジェクトにアクセスできます。主な例は、AssemblyLine コンポーネント間 (および AssemblyLine 間) でのデータの受け渡しに使用される AL の作業項目オブジェクトです。この項目オブジェクトは、各 AssemblyLine のローカル・オブジェクトであり、スクリプト変数 *work* として使用可能です。

IBM Security Directory Integrator には、JavaScript での作業中に使用できるショートカットや便利な機能があります。このため、上記のような特定の拡張マッピングを次のようにコーディングできます。

```
ret.value = [ "top", "person", "organizationalPerson", "inetOrgPerson" ];
```

拡張マッピング機能は、JavaScript 配列と項目を使用して複数の属性値を渡します。

例えば入力属性マップ (マップされた属性を戻り時に *work* 項目に含める) で、「last」という属性に

```
ret.val = anentry;
```

を割り当てるとします。また、最初に *work* が空であり、*anentry* には属性「cn」、「sn」、および「mail」が含まれているとします。

属性マッピングの完了後、*work* には「cn」、「sn」、および「mail」属性が含まれますが、値が「anentry」の単純属性「last」は**含まれません**。つまり、属性マッピングでは、属性マップから項目オブジェクトが戻される時点で、このオブジェクトが受信項目 (マップの種類 (入力または出力) に応じて *work* または *conn* にマージされます。

注: IBM Security Directory Integrator では、階層オブジェクトを利用して、属性マッピングの実行前に最初に項目を属性にカプセル化し、この動作を迂回できます。例えば、

```
// this is the entry to return
e = system.newEntry();
e.setAttribute("some", "value");

// Create an Attribute object. We don't need to provide a name since the mapping will use current map's name.
attr = system.newAttribute(null);

// add the entry to the Attribute object and return that instead of the Entry object
attr.addValue(e);

return attr;
```

「last」属性の拡張属性マップとして入力された場合、属性マッピング後に、*work* 項目には「last」属性が含まれます。この属性は、2 つの属性 (「some」と「value」) からなる項目です。

Javadoc を参照すると、項目オブジェクトには、項目とその属性および値を処理するためのさまざまな関数 (`getAttributeNames()`、`getAttribute()`、および `setAttribute()` など) があることがわかります。属性を作成して AssemblyLine 作業項目に追加する場合、フックやスクリプト・コンポーネントなどで次のスクリプトを使用できます。

```
var oc = system.newAttribute( "objectClass" );

oc.addValue( "top" );
oc.addValue( "organizationalUnit" )

work.setAttribute( oc );
```

この場合、値を設定するのに、JavaScript 配列を使用することは**できません**。

```
oc.addValue( ["top", "organizationalUnit"] ); // Does not work like Advanced Mapping
```

このコードの実行結果として、単一値を取得する `oc` 属性が作成されます。この単一値はストリング配列です。

項目オブジェクトには、*プロパティー* も含まれます。単一値のみの場合を除き、プロパティーは属性と同様にデータを格納します。属性はデータの内容を保管しますが、プロパティーはパラメトリック情報を保持するため、この情報は別個に保持することができます。プロパティーは、属性マップ選択項目として表示されることや、作業項目リストに表示されることはありませんが、属性同様にスクリプトからアクセスできます。その際、`getProperty()` や `setProperty()` などの項目関数が使用され、プロパティー値を直接処理します。プロパティー値とは、属性値のような Java オブジェクトです。属性の処理時とは異なり、中間プロパティー・オブジェクトはありません。

多くの場合、ゼロ個以上の属性を含む項目 (各属性にゼロ以上の値が設定されている)、つまりフラット・スキーマにデータ・モデルを限定できます。

これは、データの表現とスキーマの単純化と調和という IBM Security Directory Integrator の特長の 1 つです。さらに複雑な構造の情報を処理する必要が生じたときの対応も表しています。 ただし、別の項目オブジェクト (独自の属性と値を持つ) を含め、あらゆるタイプの Java オブジェクトが属性値になるため、IBM Security Directory Integrator では、階層構造データを使用できます。

このより複雑かつ構造化された階層オブジェクトの処理方法については、『階層項目オブジェクトの操作』で説明します。

階層項目オブジェクトの操作

階層構造化データを操作するもう 1 つの方法として、IBM Security Directory Integrator 項目オブジェクトの階層オブジェクト・サポートを利用する方法があります。

以前のバージョンとは異なり、IBM Security Directory Integrator バージョン 7.1.1 以降 では階層項目オブジェクトの概念に対応しています。項目オブジェクトは階層のルートを表し、各属性は階層内のノードを表します。この論理に基づき、各属性の値は階層のリーフとなります。階層をトラバースするための API もあります。この API は、DOM 3 仕様を部分的にインプリメントしています。この仕様のクラスのうち、インプリメントされているクラスを次に示します。

- Org.w3c.dom.Document - Entry クラスによりインプリメント。
- Org.w3c.Element - Attribute クラスによりインプリメント。
- Org.w3c.Attr - Property クラスによりインプリメント。
- Org.w3c.Text および org.w3c.CDATASection - AttributeValue クラスによりインプリメント。

これらのクラスは、DOM 仕様で提供されているクラスのうち、階層データの表現に必要な最小限のクラスです。v7.0 以前の API は後方互換性の点から階層に対応していませんでした (例えば、子エレメントに対してアクセス/修正/除去することができない)。このため、DOM API のみが階層構造を操作できます。

項目構造の後方互換性を維持するため、デフォルトでは項目には常にフラットな属性が使用されます。提供される新規 DOM API のいずれかを呼び出した後のみ、オンデマンドで項目を階層構造にすることができます。これにより、項目の階層特性を認識するコンポーネントのみがその特性を利用できるようになり、階層特性を認識しないコンポーネントは引き続き稼働するために変更される必要がなくなります。IBM Security Directory Integrator v7.0 では、ユーザーが階層ツリーを容易に作成できるようにするための新しい名前表記が導入されました。ドットが含まれている名前はすべて、ドットで区切った複数の単純名からなる複合名であると解釈されます。このような複合名が階層項目に渡されると、複合名は単純名に分割され、その複合名で記述される階層が作成されます。

例えば次の JavaScript コードを実行するとします。

```
// create a new empty Entry object
var entry = new com.ibm.di.entry.Entry(true);
// create a new branch of 2 levels
entry.setAttribute("firstLevelChild.secondLevelChild", "level2Value");
// finds the already existing branch and creates a new node on level 3
entry.setAttribute("firstLevelChild.secondLevelChild.thirdLevelChild", "level3Value");
```

次の構造が作成されます。

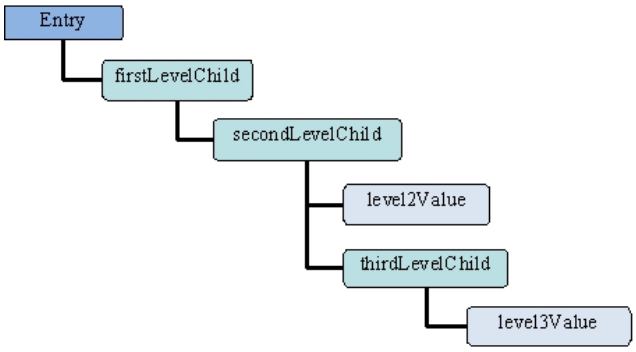


図 1. 単純階層項目

注: 項目構造が階層構造に変換されるまでは、名前は単純名に分割されないことを理解しておくことが重要です。例えば、項目がフラット項目であり、従来と同じフラット構造を引き続き作成する古いメソッドのみが使用される場合は次のようになります。

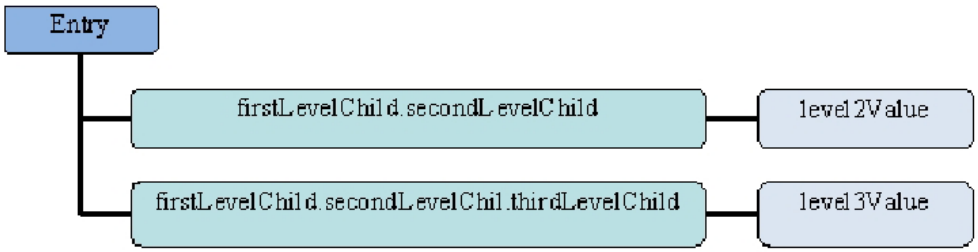


図 2. 従来のフラット構造の項目

場合によっては、項目オブジェクトがエスケープ文字を認識できるようにするために、項目の属性の名前にドットを含める必要があります (現時点では ¥¥ と ¥. のみがサポートされています)。

注: スクリプトから処理する場合は、円記号 (¥) を正確にエスケープするには、円記号 (¥) をもう 1 つ使用してください。
例えば次の階層が必要であるとします。

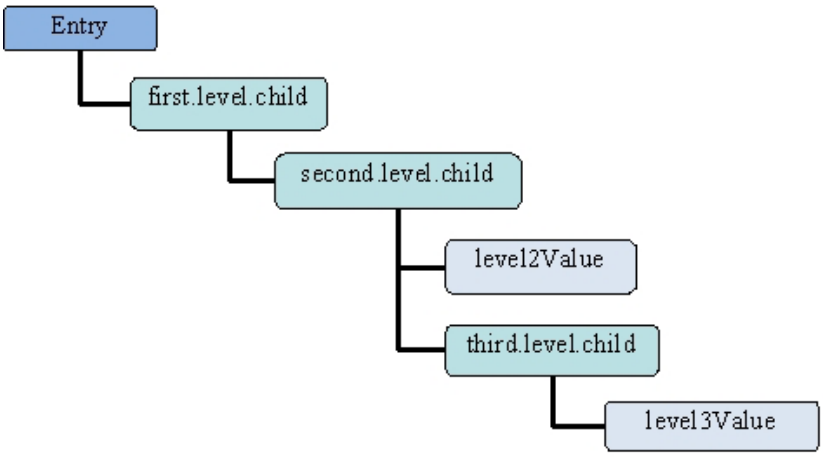


図 3. もう 1 つの単純階層項目

この階層項目を作成するスクリプトを以下に示します。

```
var entry = new com.ibm.di.entry.Entry(true);
entry.setAttribute("first¥¥.level¥¥.child.second¥¥.level¥¥.child", "level2Value");
entry.setAttribute("first¥¥.level¥¥.child.second¥¥.level¥¥.child.third¥¥.level¥¥.child", "level3Value");
```

階層項目を暗黙に処理する際に、以前のリリースの IBM Security Directory Integrator との後方互換性を維持するため、Attribute クラスと Entry クラスの両方によって公開される従来のすべてのメソッドが僅かに変更されています。例えば `Entry.getAttributeNames()` メソッドは、ツリーに含まれるリーフの絶対パスの配列を戻します。上記の構造の場合、getAttributeNames メソッドは次の配列を戻します。

```
["first¥.level¥.child.second¥.level¥.child", "first¥.level¥.child.second¥.level¥.child.third¥.level¥.child"]
```

Entry.size() メソッドは、getAttributeNames() メソッドによって戻される配列のエレメントの総数を戻します。この例では、項目オブジェクトの子が 1 つの属性のみであるために、size() メソッドから 1 が戻されると予想されることもありますが、実際には 2 (ツリー全体のリーフの数) が戻されます。

これは、メソッド getAttributeNames() と size() の両方がフラット構造のみを処理するためです。子の実際のサイズを取得するには、DOM API を次のように使用する必要があります。Entry.getChildNodes().getLength();

フラット構造の項目の場合、従来の API は、以前のリリースと同様に動作します。

属性オブジェクト

属性オブジェクトが拡張され、階層構造作成機能が追加されました。階層構造は DOM 仕様に準拠していることから、属性オブジェクトは XML ネーム・スペースの概念に対応しています。

階層機能に対応した新規メソッドを提供するため、Attribute クラスの拡張も必要でした。getValue/setValue/addValue メソッドもまた後方互換性を備えており、特定のエレメントの値のみを戻します。この場合、DOM API から各値にアクセスすると、値が AttributeValue クラスにラップされ、Text ノードとして扱われる点が異なります。属性の子エレメント (Attributes または AttributeValues など) を取得するには、DOM API を使用する必要があります。

項目の子属性は、その DOM メソッドへのアクセスがあった場合に、項目の構造を階層構造に切り替えることもできます。Entry クラスとは異なり、Attribute クラスはこの処理を暗黙的に実行します。明示的に切り替える機能は提供しません。

IBM Security Directory Integrator v7.0 では、複雑な構造へ容易にアクセスできるようにサーバーのスクリプト機能の拡張がサポートされています。詳しくは、56 ページの『スクリプト内のナビゲーション』を参照してください。

AttributeValue オブジェクト

このクラスは、階層ツリーの値を表します。DOM 仕様に従い、ツリーの値は常にストリングです。ただし、これまでのいくつかのリリースにおいて、AttributeValue クラスではあらゆる種類のオブジェクトが維持されていました。現行バージョンでもこれは有効です。AttributeValue オブジェクトの唯一の相違点は、DOM を介してこのオブジェクトにアクセスすると、被包含オブジェクトのストリング表現が戻される点です。AttributeValue クラスがノードを表現し、かつ DOM 仕様の定義に従って値を持つようにするには、インターフェース org.w3c.dom.Text または org.w3c.dom.CDATASection をインプリメントする必要があります。AttributeValue はこの両方をインプリメントするため、必要に応じていずれかのノードを表現できます。

プロパティー・オブジェクト

属性には、ゼロ個以上のプロパティー・オブジェクトを含めることができます。Property クラスは org.w3c.dom.Attr インターフェースをインプリメントするので、DOM の概念における属性を表現します。XML の概念では、プロパティーを使用して接頭部/ネーム・スペースを宣言します。

オブジェクトの転送

項目間で属性をマップすると、常にソース属性がコピーされます。

例を示します。

```
entry.appendChild(conn.getFirstChild());
// or
entry.setAttribute("name", conn.getFirstChild());
```

属性が項目の第一レベルの子ではない場合でも、この属性はコピーされます。この処理を実行するスクリプトを以下に示します。

```
entry.a.b.c.d.appendChild(conn.e.f.g);
```

属性オブジェクトを複製せずに項目間で移動するには、最初に元の親からこのオブジェクトを切り離し、次に新しい親に付加します。

例を示します。

```
var src = entry1.b.source;
entry1.b.removeChild(src);
entry2.a.target.appendChild(src);
```

属性オブジェクトを同じ項目内のある親から別の親へ移動すると、属性は自動的に移動します。複製は行われません。

例を示します。

```
entry.a.target.appendChild(entry.b.source);
```

この例では「ソース」属性がその親（「entry.b」）から切り離され、「entry.a.target」属性に付加されます。複製は行われません。

属性オブジェクトをソースから除去しない場合は、属性のコピーを次のように付加できます。

```
entry.a.target.appendChild(entry.b.source.clone());
```

スクリプト内のナビゲーション

IBM Security Directory Integrator ScriptEngine では、項目の属性をその名前で参照するだけで、属性に容易にアクセスできます。例えば `entry.attrName` は *attrName* という名前の属性を戻します。

1. JavaScript Engine は、名前を解決する際に、その名前が要求されたコンテキスト・オブジェクトに基づいて解決します。例えば呼び出し `entry.a` が実行される場合、名前 *entry* はコンテキスト・オブジェクトであり、*a* は解決する子オブジェクトの名前です。 JavaScript Engine は、最後のオブジェクトを解決するまで、左から右の方向で変換処理を実行し、各コンテキスト・オブジェクトを評価します。以下に示す図の場合、呼び出し `entry.a.b.c` は次に説明する順序で解決されます。まず、コンテキスト・オブジェクトとして使用する *entry* オブジェクトを検索します。
2. *a* という名前のコンテキスト・オブジェクトを検索します。*entry* オブジェクトには、*a* という名前の子が 1 つだけ含まれています。この子が、次のステップの次のコンテキスト・オブジェクトになります。

3. 名前 *b* のコンテキスト・オブジェクトを検索します。このコンテキスト・オブジェクトには、*b* という名前の子が 2 つあります。これらの子をリストに入れ、そのリストを戻します。
4. 最後に、直前の操作で戻されたリストから名前 *c* が検索されます。リストの各エレメントには、この名前の子が 1 つ以上含まれています。これらの子をすべて取得し、リストに入れます。このリストが、式全体の実際の解決結果となります。

この項目オブジェクトの例を次の図に示します。

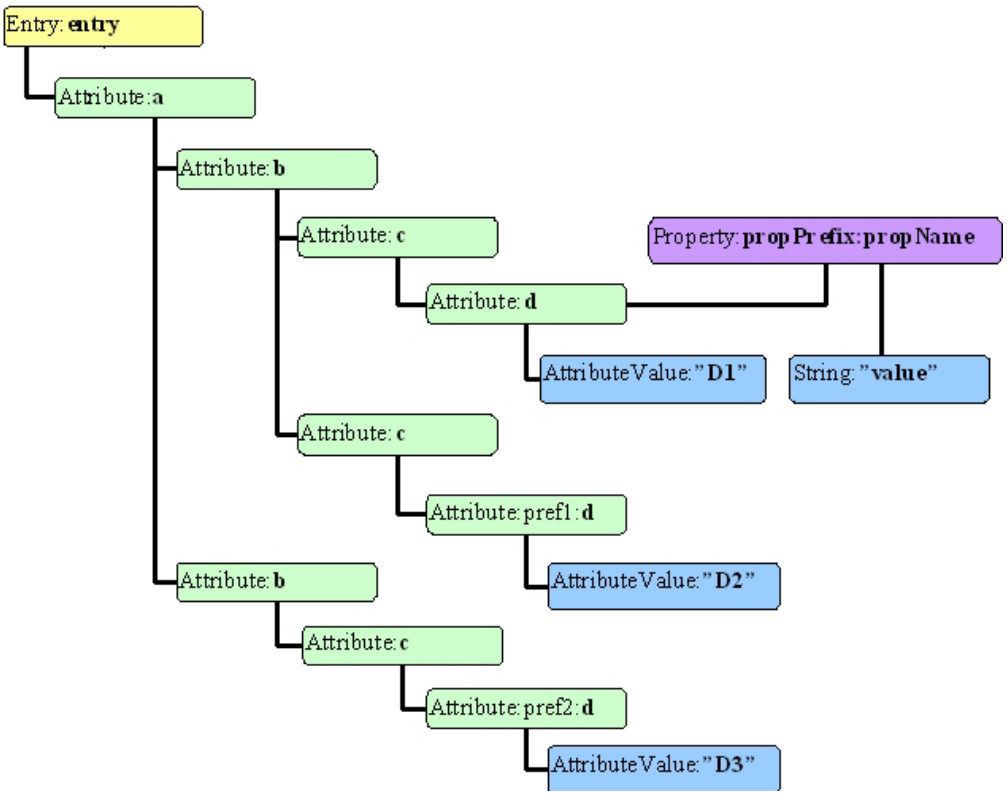


図 4. 階層項目オブジェクトの例

IBM Security Directory Integrator のスクリプト・エンジンでは、より恣意的な名前を子の解決処理で使用できます。例えば、名前にドットが含まれている子の場合、その名前を参照するには、次のような構文を使用します。

```
work["{namespace}name:Containing¥.invalid¥.charaters"]
```

ドットがエスケープされているため、ドットはローカル名の一部であり、スクリプト・エンジンがパス分離文字として処理してはいけないことを示しています。

操作の実行対象である現行コンテキスト・オブジェクトによっては、最終結果が異なることがあります。IBM Security Directory Integrator は、JavaScript Engine の標準オブジェクト操作に準拠するため、以下のオブジェクトに機能拡張をインプリメントしました。

項目　コンテキスト・オブジェクトがこのタイプのインスタンスである場合、名前解決メカニズムは次の構文を検索します。

- `@<name>` - 指定された名前のプロパティーを項目オブジェクト内で検索します。解決後のオブジェクトは、NULL またはそのプロパティーにマップされたオブジェクトです。
- `<prefix>:<localName>` - <prefix> に相当する接頭部を持ち、ローカル名が <localName> である子を項目オブジェクト内で検索します。解決後のオブジェクトは、NULL または既存の属性オブジェクトです。
- `<localName>` - ローカル名が <localName> である 1 番目の子を項目オブジェクト内で検索します。解決後のオブジェクトは、NULL または属性オブジェクトです。
- `{namespaceURI}<localName>` - 項目で、指定された namespaceURI に属し、指定された名前と同じローカル名を持つ 1 番目の子属性を検索します。

  注: 接頭部が指定されている場合、この接頭部は無視されます。名前解決メカニズムは、指定されている namespaceURI と localName のみを検索します。解決後のオブジェクトは、NULL または属性オブジェクトです。

属性 コンテキスト・オブジェクトがこのタイプのインスタンスである場合、名前解決メカニズムは次の構文を検索します。

- `@<prefix>:<localName>` および `@<localName>` - 指定された接頭部またはローカル名 (あるいはこの両方) を持つか、または指定されたローカル名のみを持つプロパティー・オブジェクトを、属性内で検索します。指定された名前に一致するプロパティーがない場合は NULL を戻し、それ以外の場合は 1 つのプロパティー・オブジェクトを戻します。
- `@{namespaceURI}<localName>` - 指定された namespaceURI に属し、指定されている名前と同じローカル名を持つプロパティーを属性内で検索します。

  注: 接頭部が指定されている場合、この接頭部は無視されます。名前解決メカニズムは、指定されている namespaceURI と localName のみを検索します。解決後のオブジェクトは、NULL またはプロパティー・オブジェクトです。
- `[<index>]` - 属性内での取得する値の位置を指定します。この表記を使用してこの属性の子にアクセスすることはできません。NULL または指定された位置のオブジェクトを戻します。
- `<prefix>:<localName>` および `<localName>` - 指定された接頭部またはローカル名 (あるいはこの両方) を持つ子を属性内で検索します。NULL、指定された名前の 1 つの子 (子が 1 つのみの場合)、または条件に一致するすべての子属性を含む NodeList を戻します。
- `{namespaceURI}<localName>` - 指定された namespaceURI に属し、指定された名前と同じローカル名を持つすべての子属性を属性内で検索します。

  注: 接頭部が指定されている場合、この接頭部は無視されます。名前解決メカニズムは、指定されている namespaceURI と localName のみを検索します。解決後のオブジェクトは、NULL、1 つの属性オブジェクト、または検索名に一致するすべての属性を含む NodeList です。

NodeList

コンテキスト・オブジェクトがこのタイプのインスタンスである場合、名前解決メカニズムは次の構文を検索します。

- `[<index>]` - NodeList 内で取得するエレメントの位置を指定します。NULL または指定された位置のオブジェクトを戻します。索引が範囲外にある場合は例外がスローされます。
- `@<prefix>:<localName>` および `@<localName>` - 指定された接頭部またはローカル名 (あるいはこの両方) を持つプロパティーを、NodeList の各エレメントで検索します。NULL、1 つのプロパティー・オブジェクト (1 つのみ検出された場合)、または NodeList で検出されたすべてのプロパティー・オブジェクトのリストを戻します。
- `@{namespaceURI}<localName>` - 指定された namespaceURI に属し、指定された名前と同じローカル名を持つプロパティーを各属性内で検索します。

  注: 接頭部が指定されている場合、この接頭部は無視されます。名前解決メカニズムは、指定されている namespaceURI と localName のみを検索します。解決後のオブジェクトは、NULL、1 つのプロパティー・オブジェクト (1 つのみ検出された場合)、または NodeList で検出されたすべてのプロパティー・オブジェクトを含む NodeList のいずれかです。
- `<prefix>:<localName>` および `<localName>` - 指定されている接頭部またはローカル名 (あるいはこの両方) を持つ子を、NodeList の各エレメント内で検索します。NULL、1 つの属性オブジェクト (1 つのみ検出された場合)、または NodeList で検出されたすべての属性オブジェクトを含む NodeList を戻します。
- `{namespaceURI}<localName>` - 指定された namespaceURI に属し、指定された名前と同じローカル名を持つすべての子属性を、各属性内で検索します。

  注: 接頭部が指定されている場合、この接頭部は無視されます。名前解決メカニズムは、指定されている namespaceURI と localName のみを検索します。解決後のオブジェクトは、NULL、1 つの属性オブジェクト、または検索名に一致するすべての属性を含む NodeList です。

次のオプションがあります。

1. 先頭から順にすべての *d* エレメントを参照して、これらのエレメントにアクセスできます。例えば `entry.a.b.c.d` の場合、そのパスに一致するすべての *d* 属性を含む NodeList タイプのオブジェクトが戻されます。この例の場合、3 つの *d* エレメントがすべて戻されます。
2. 接頭部とローカル名の両方を指定して属性にアクセスできます。例えば `entry["a.b.c.pref1:d"]` の場合、1 つの属性 (接頭部「pref1」を持つ属性) が戻されます。
3. [ ] 表記を使用して NodeList の各属性にアクセスできます。例えば `entry.a.b.c.d[0]` の場合、1 つの属性 (前述の構造の 1 番目の *d* エレメント) が戻されます。

4. [ ] 表記を使用してエレメントをナビゲートできます。例えば entry.a.b[0].c.d の場合、1 番目の *b* ブランチのすべての *d* 属性を含む属性リストが戻されます。
5. @ 表記を使用して属性のプロパティーを取得できます (DOM 命名規則を使用した要素の属性)。例えば entry.a.b.c.d[0]["@propPrefix:propName"] の場合、そのプロパティーの値 (DOM に準拠した属性の値) を含むストリング・オブジェクトが戻されます。propPrefix と propName がコロン記号 (「:」) で区切られていることと、プロパティーに接頭部がある場合はプロパティーを検出するために接頭部と名前の両方を指定する必要がある点に注意してください。あるいは、ネーム・スペースと propertyName を使用して接頭部を含むプロパティーを検索することもできます。
6. ユーザーが実行するメソッドの名前を持つ子を検索する前に、属性のメソッドを呼び出すことができます。例えば entry.a.b.[0].getChildNodes() の場合、1 番目の *b* 属性に含まれるすべての属性値を含む NodeList オブジェクトが戻されます。
7. 名前を指定して項目属性にアクセスできます。例えば entry.attrName の場合、*attrName* キーにマップされている属性が戻されます。
8. .@ 表記を使用して項目プロパティーにアクセスできます。例えば entry.@propName の場合、*propName* キーにプロパティーとしてマップされているオブジェクトが戻されます。
9. 子ノードが属するネーム・スペースを指定してそれらの子ノードにアクセスできます。例えば work.a.b[{someNamespace}c] の場合、「someNamespace」ネーム・スペースに属するすべての *c* オブジェクトが戻されます。

注:

1. 属性名が項目/属性のメソッドの名前と同じである場合、そのメソッドが呼び出されます。属性にアクセスする必要がある場合は、オブジェクトのメソッドを従来の方法で使用する必要があります (Entry.getAttribute("getAttribute");)。
2. フラット項目を使用する場合、検索する属性のネーム・スペースが指定されていないと、スクリプト・エンジンでは項目が階層構造に変換されません。例えば entry["{ns}element"] などです。コンテキスト・オブジェクトのタイプが属性の場合、スクリプト・エンジンは項目を自動的に階層項目に変換します (例: entry.attr.child)。
3. 検索する属性の名前にドットが含まれており、項目がフラット構造の場合、ドット表記ではなくブラケット表記を使用する必要があります。このようにしないと、子ノードの解決前に項目が強制的に階層構造に変換されます。例えば項目がフラット構造の場合、呼び出し entry.http.body を使用して属性 http.body にアクセスすることはできません。この属性にアクセスするには、entry["http.body"] を使用するか、または entry.http.body を呼び出す前に entry.enableDOM() を呼び出す必要があります。
4. 式から取得した参照 (例: a.b.c.d) を複数回使用する場合、この参照をローカル変数に割り当てておくことをお勧めします。これは、同じ式を繰り返し評価すると、同じ参照を取得する操作のオーバーヘッドがかかるためです。
5. 例えば式 entry.a.b.c[0].d は属性オブジェクトを参照しますが、entry.a.b[0].c.d は NodeList オブジェクトを参照します。参照オブジェクトを認識するには、オブジェクトの名前を確認します。次に例を示します。

```
var obj = a.b.c.d;
if (obj.getClass().getSimpleName().equals("Attribute") ) {
  // handle this as Attribute
} else {
  // handle this as a list of Attributes
}
```

例えば、式から戻される各エレメントを処理する必要がある場合は、戻されるエレメントが 1 つのみであっても、for/in ループ構造を次のように使用できます。

```
for (obj in entry.a.b.c.d) {
  // the obj will be an object of type Attribute
}
```

項目オブジェクトでも同様に使用できます。次に例を示します。

```
for (obj in work) {
  // the obj will be an Entry's attribute
}
```

6. ScriptEngineOptions では値を割り当てることができます。例えば、以下の式が有効です。

   entry.a.b.c.d[0]["@propPrefix:propName"] = "new value";
   プロパティー値が変更されます。プロパティーが存在しない場合は新規に作成されます。

   entry.a.b.c.d[0][0] = "new value";
   位置 0 の属性値が置換されます。

   entry.a.b.c.d[0][1] = "another value";
   属性に別の値が追加されます (索引が値配列のサイズと同じである場合)。

   entry.a.b.c.d[0][a.b.c.d[0].size()] = "appended value";
   オブジェクト・リストに新しい値を追加する必要があるが、最終エレメントの位置が分からない場合は、`Attribute#size()` を使用してこのエレメントの子の数を取得します。

   new com.ibm.di.entry.Entry().@propName = "someValue";
   項目オブジェクトに propName という名前のプロパティーがない場合はこのプロパティーが新規に作成され、その値としてストリング「someValue」が設定されます。

   entry.a.b.c[1] = "value";
   entry.a.b.c が NodeList として解決され、新規ストリング・オブジェクトが値として解決後の NodeList オブジェクトの 2 番目のエレメントに追加されます。例えば *entry.a.b.c* が属性に解決される場合、解決後の *c* 属性の 2 番目の値が新規ストリング値に置き換えられます。

   entry.a.b.c[2]["pref3:d"] = "value";
   entry.a.b.c リストの 3 番目の *c* 属性に新規の子 *pref3:d* が追加されます。*entry.a.b.c[2]* は属性またはリストに解決される点に注意してください。

   entry.a.b.c["{namespaceURI}:d"] = "myTextValue";
   d と同等のローカル名を持ち、ネーム・スペース `namespaceURI` に属する *c* の 1 番目の子属性に値が設定されます。このような属性が見つからない場合は属性が新規に作成され、値が割り当てられます。属性の作

成時に、ネーム・スペースに接頭部を関連付けることができます。この例では、`entry.a.b.c["{namespaceURI}prefix:d"] = "myTextValue";` によってこの関連付けが行われます。

entry["{http://ibm.com/xmlns/}first.second.third"] = null;

最初に、ネーム・スペース「http://ibm.com/xmlns/」に属する、ローカル名「first」を持つエレメントの検索が試行されます。見つからない場合、エレメント「first」が作成されて、その子エレメント「second」の解決が試行されます。見つからない場合、エレメント「second」が作成されて、そのネーム・スペースが「first」のネーム・スペースに設定されます。最後に、「third」エレメントの解決が試行されます。見つからない場合、値なしで「third」エレメントが作成されます。これは、式 `entry["{http://ibm.com/xmlns/}first.{http://ibm.com/xmlns/}second.{http://ibm.com/xmlns/}third"] = null;` と同等です。

スクリプトによる上記の構造の作成

```
// create a new entry that will hold the structure
var entry = new com.ibm.di.entry.Entry();
// create the first branch
var d = entry.newAttribute("a.b.c.d");
// create a new property and assign it a value
d["@propPrefix:PropName"] = "value";
// create a new value of the d Attribute
d[0] = "D1";
// append a new value for the entry.a.b attribute
entry.a.b.appendChild(entry.createElement("c"));
// create a new Attribute d with a prefix on the second c attribute,
// and assign the string "D2" as a first value
entry.a.b.c[1]["pref1:d"] = "D2";
// create a new child of the a Attribute
entry.a.appendChild(entry.createElement("b"));
// choose the second attribute from the entry.a.b NodeList and create a new child named c
entry.a.b[1].appendChild(entry.createElement("c"));
// create the d child of the c Attribute
entry.a.b[1].c["pref2:d"] = "D3";
```

XPath を使用したナビゲーション

Entry クラスには、XPath 式に基づいてデータをナビゲートおよび取得できる便利なメソッドがあります。XPath を使用して項目のデータを照会する方法は、スクリプト内で単純なナビゲーションを使用する方法よりもかなり高度な手法です。検索または値突き合わせ (あるいはこの両方) のロジックを 1 つの式にインプリメントする方法は、同じ結果を実現する複数行のスクリプトを作成する方法よりもは容易です。

Entry クラスのメソッドを以下に示します。

- `NodeList getNodeList (String xPath);`
- `Attribute getFirstAttribute(String xPath);`
- `String getStringValue(String xPath);`
- `Number getNumberValue(String xPath);`
- `Boolean getBooleanValue(String xPath);`

階層構造のフラット化

階層データを処理する際に、階層のノードをフラットにする必要があることがあります。このためには、ツリーをトラバースするスクリプトを作成するか、または以下のいずれかのメソッドを使用します。

- `getElementsByTagName(String namespace, String localName);`
- `getElementsByTagName(String tagName);`

これらのメソッドは、ツリーをトラバースし、指定された名前とネーム・スペースを持つエレメントを検索します。DOM API では、これらのメソッドで名前とネーム・スペースの両方に文字「*」を使用できます。この文字は、すべてのエレメントの名前/ネーム・スペースに一致するワイルドカードを表します。名前にこの文字が使用されている場合、NodeList のすべての属性ノードが戻され、ツリーがフラット化されます。階層の構造は変更されていない点に注意してください。戻された NodeList は、ツリーで検出されたエレメント・ノードのコンテナーに過ぎません。

56 ページの『スクリプト内のナビゲーション』の構造の項目に対して次のスクリプトを実行するとします。

```
var list = entry.getElementsByTagName("*");
```

list 変数に次の構造が格納されます。

```
list
   |
  + a
   |
  + b
   |
  + c
   |
  + d
   |
  + c
   |
  + pref:d
   |
  + b
   |
  + c
   |
  + pref2:d
```

例外

例外がスローされるケースを以下に示します。

- メソッド `entry.appendChild(newAttr)` が呼び出されたときに、メソッドに渡された newAttr オブジェクトの名前を持つ属性が項目に既に含まれている場合。
- 文書/ノード/エレメントにより定義され、Entry/Attribute クラスによりインプリメントされたメソッドに、予期しないパラメーターが渡された場合。
- 指定したスクリプトが、存在しない属性/NodeList の索引を参照している場合、ArrayIndexOutOfBoundsException がスローされます。

ソリューションへのスクリプト記述の統合

前述したように、統合ソリューション内でカスタム処理が必要なときには、スクリプトを使用します。IBM Security Directory Integrator のベスト・プラクティスは、このカスタム処理を属性変換とフロー制御という 2 つのカテゴリーに分類することです。

注: この分類は慣例であり、システムによって強制される制限またはルールではありません。カスタム・データ処理は、その性質上、データ・フロー内のいくつかの識別可能なポイント (なんらかの処理を開始する前、特定の項目を処理する前、失敗の後など) で必要になります。したがって、記述したコードをできる限りそのポイントの近くに配置すると、ソリューションが理解しやすくなり、保守も容易になります。

属性変換を実行する論理的な場所は、入力および出力の**属性マップ**です。AssemblyLine 内で下流にあるスクリプト・ロジックまたはその他のコネクターが必要とする新しい属性を計算する必要がある場合は、できる限り**入力マップ**内でこれを実行することをお勧めします。また、 1 つの出力ソースのために属性を変換する必要がある場合は、関連するコネクターの**出力マップ**に属性を配置すれば、**作業**項目オブジェクトで多くの出力固有変換を処理する必要はなくなります。

もう 1 つのカスタム・ロジック・カテゴリーであるフロー制御をインプリメントする最良の方法は、自動化ワークフロー内でロジックが必要とされるポイントで呼び出されるフック内にインプリメントすることです。これらの制御点には、構成エディターから簡単にアクセスできます。カスタム処理のインプリメントは、正しい制御点を識別し、適切な編集ウィンドウでスクリプトを追加するという単純な作業です。

スクリプト化されているコードの独立したブロックである AssemblyLine **スクリプト・コンポーネント**で独自のカスタム処理を作成し、 AssemblyLine の内部でコードの位置を変更することもできます。スクリプト・コンポーネントは、テストおよびデバッグ中に頻繁に使用されますが、実動時の構成でも重要な役割を果たすことがあります。コンポーネントにはわかりやすい名前を付けること、およびこのロジックを**属性マップ**や**フック**ではなくスクリプト・コンポーネントにインプリメントした理由を説明する記述[7]をスクリプト自体に含めることが重要です。

スクリプトを入力する適切な制御点を正しく識別することは重要です。それと同じくらい重要なのは、その制御点に関連付けられた 1 つのゴールのみを対象とするようにスクリプトのスコープを制限することです。複数の論理単位が相互に独立して維持されている場合は、それらのロジックを再利用できる可能性は高くなり、コンポーネントが他のコンテキストで再配列または再利用されたときにロジックが分断される可能性は低くなります。再使用可能なコードを作成する方法の 1 つは、同じコードを複数の場所へコピーおよび貼り付けするのではなく、**スクリプト・ライブラリー** (または「**プロローグ**」フック) 内に独自の関数を作成して、頻繁に使用されるロジックをインプリメントすることです。

ソリューションを作成するときに留意すべきいくつかのベスト・プラクティスを次に示します。

7. この記述は、他のユーザーにとって役立つ以外に、自分が作成したコードを後から見直す必要があるときにも役立ちます。

- 属性マップ内で属性操作を実行する。
- フロー制御 (フィルター処理、妥当性検査、ブランチなど) をフックに配置し、また必要に応じて AssemblyLine スクリプト・コンポーネントを配置する。
- 自動化動作 (AssemblyLine ワークフロー、コネクター・モードなど) をできる限り使用する。
- AssemblyLine を短くして集中化させることにより、ソリューションを単純化する。
- 頻繁に使用するロジックは、別個のブロックに入れる (例えば、スクリプトは「リソース」セクションに入れる)。
- 再使用を検討する。

上記の方法はベスト・プラクティスですが、状況によっては、確立された方法とは異なるスクリプト記述が必要になることもあります。そのような場合は、前述したように、作業内容を長期間にわたって保守および拡張できるようにソリューションを文書化しておくことが非常に重要です。

スクリプト記述による実行の制御

エンジンが公開するさまざまなクラスおよびオブジェクトは、AssemblyLine 内でユーザーが作成したスクリプトからアクセスでき、読み取りや変更を行うことが可能です。これらのオブジェクトは、任意の時点における AssemblyLine の状態、および IBM Security Directory Integrator 環境全体の状態を表します。これらのオブジェクトのいずれかを変更すると、IBM Security Directory Integrator 環境が変更され、統合プロセスの実行にも影響します。

注: コンポーネントまたは AssemblyLine のいずれかのインスタンスに変更を適用することが可能です。システム・パラメーターや Java パラメーターなどの稼働パラメーターを変更することもできます。構成ファイル (Config) に変更を行うこともできます。この場合は、構成オブジェクトの新しいインスタンスが構成の変更内容を反映します。

グローバル・オブジェクトについて詳しくは、IBM Security Directory Integrator 製品に含まれている Javadoc を参照してください (構成エディターで「**ヘルプ**」 > 「**Javadoc**」を選択します)。

インストール・パッケージには、使用可能なすべてのクラスとインスタンスについての説明が含まれています。

公開されているクラスとインターフェースを知ることにより、IBM Security Directory Integrator エンジンの各エレメント、およびエレメント間の関係についての理解を深めることができます。

変数の使用

処理中のデータのための標準コンテナー・オブジェクト (項目オブジェクト) と、その他の汎用変数やデータ・タイプ (JavaScriptによってユーザーに提供される汎用変数やデータ・タイプ、およびユーザー自身が作成する汎用変数やデータ・タイプ) を区別することが重要です。ユーザーの能力とスクリプト言語の機能による制限以外には、IBM Security Directory Integrator ソリューション内で記述できるスクリプ

トの内容には制限はありません。ただし、データ・フローのコンテキスト内でデータを操作する場合は、項目オブジェクトの構造に留意しながら項目オブジェクトを使用する必要があります。

項目オブジェクトは、それ自体がデータ値のコンテナーであり、属性を格納します。属性値は、それ自体がオブジェクト (`java.lang.String`、`java.util.Date`、およびより複雑な構造) です。属性値が、独自の属性と値のセットを持つ別の項目オブジェクトである場合もあります。IBM Security Directory Integrator の役割は、接続されたシステム内でデータがどのように格納されているか、およびこれらのネイティブ・タイプとシステムの (Java オブジェクト内の) データ表現を相互にどのように変換するかを認識することです。

属性値のクラスが分かっている場合は、この値に正常にアクセスし、値を解釈することができます。例えば、`java.lang.String` 属性に含まれる浮動小数点値を浮動小数点として使用する場合は、最初にスクリプト言語を使用して、手動でこの値を何らかの数値データ・タイプに変換する必要があります。

統合プロセス内のデータ・フローと直接関連しない変数またはプロセスを作成していて、グローバル・オブジェクトが使用可能なときには、原則として、スクリプト言語によって使用可能になったすべての変数 (オブジェクト) を宣言および使用することができます。これらの変数の目的は、ユーザーがスクリプトを記述する制御点に関連付けられた特定のゴールを達成しやすくすることです。これらの変数は、一時バッファーとしてのみ機能するようにし、IBM Security Directory Integrator 環境の状態には影響を及ぼさないようにする必要があります。

プロパティーの使用

IBM Security Directory Integrator サーバーは、AssemblyLine の存続期間中に、AssemblyLine の実行環境に関連したさまざまなコンポーネント・プロパティーを提供します。これらのプロパティーは、スクリプト内で (スクリプト・コンポーネントまたはコンポーネントのフックのいずれかから) 照会することができます。1 つのプロパティー (*lastCallStatus*) は、設定も可能です。

プロパティーにアクセスするには、AssemblyLine オブジェクト (コネクターなど) を使用してメソッド `get(property_name)` を呼び出し、property_name 値を抽出します。あるいは、`put(property_name, property_value)` メソッドを使用して、プロパティーを目的とする値に設定します。プロパティーを設定する際に、プロパティー名またはプロパティー値を NULL にすることはできません。NULL とした場合、例外とそれに対応するメッセージがスローされます。

使用可能なプロパティーの集合を以下に示します。

表 6. AssemblyLine の実行時に使用可能なコンポーネントのプロパティー

プロパティー	使用法
numErrors	発生したエラーの数。
numAdd	AssemblyLine が追加した項目の総数 (コネクターによって AddOnly モードで実行)。
numModify	AssemblyLine が変更した項目の総数 (コネクターによって更新モードで実行)。

表 6. AssemblyLine の実行時に使用可能なコンポーネントのプロパティー (続き)

プロパティー	使用法
numDelete	AssemblyLine が削除した項目の総数 (削除モードのコネクターによって実行)。
numGet	AssemblyLine が検索した項目の総数 (コネクターによってイテレーター・モードで実行)。
numGetTries	AssemblyLine が項目の検索を試行した回数の合計 (イテレーター・モードのコネクターで実行)。
numGetClient	受け入れられたクライアントの総数 (サーバー・モードのコネクターで使用可能なクライアント)。
numGetClientTries	AssemblyLine が次に接続されたクライアントの取得を試行した回数の総数 (コネクターによってサーバー・モードで実行)。
numCallreply	AssemblyLine が実行した呼び出し/応答操作の総数 (コネクターによって CallReply モードで実行)。
numLookup	AssemblyLine が実行したルックアップ操作の総数 (コネクターによって更新/削除/ルックアップ・モードで実行)。
numNoChange	AssemblyLine が処理したが変更されなかった項目の総数。
numSkipped	AssemblyLine がスキップした項目の総数。
numIgnored	AssemblyLine が無視した項目の総数 (コネクターによって更新/デルタ・モードで実行)。
lastCallStatus	AL の実行の状況が格納されます。これは、読み取り専用のプロパティーであるだけでなく、ユーザーが変更することが可能です。このプロパティーの値は、AL の実行の状況に応じて、「fail」または「success」になります。
lastConn	最後のコネクター操作からの Conn 項目。最初のコネクター操作の前は、lastConn の値は NULL になります。
lastError	Java オブジェクトとしての最後のエラー。
hooksInvoked	コンポーネントが最後に呼び出されたときに起動されたフックの名前の java.util.List。これらの名前は内部名です。
success	最後の操作が成功であった場合、このプロパティーは true に設定されます。そうでない場合は false に設定されます。
endOfData	イテレーター・コンポーネントがデータの終わりに達したとき、true になります。そうでない場合は、false になります。このプロパティーを変更しても効果はありません。

注: 読み取り専用のプロパティーを変更しようとすると、例外とそれに対応するメッセージがスローされます。

例

上記のようなコンポーネントのプロパティーの使用例を示すために、*FS* という名前のファイル・システム・コネクターといくつかのスクリプト・コンポーネントがあるものとします。以下の JavaScript コードは、FS の「GetNext 成功」フック内にあります。

```
if(work.getString("ID") == null)
throw new java.lang.Exception("Missing ID");
//AL サイクルを正しく実行するためには、ID が必要です。このため、例外をスローします
```

次の JavaScript コードは、FS の「エラー発生時のデフォルト」フック内にあります。

```
if(FS.get("lastError").getMessage().equals("Missing ID")) {
//AL の実行を促す ID を追加することによって、このエラーを修正できました
 work.setAttribute("ID", "SomeID"); //ID を追加します
 if(FS.get("fixErrors") == null) {
  var vector = new java.util.Vector();
  vector.add(FS.get("lastError"));
  FS.put("fixErrors", vector);//修正されたエラーすべてを自分のカスタム・プロパティーに保存します
 } else {//前に同様のエラーを修正しました
  var vector = FS.get("fixErrors");
  vector.add(FS.get("lastError"));
  FS.put("fixErrors", vector);//修正されたエラーすべてを自分のカスタム・プロパティーに保存します
 }
 FS.put("lastCallStatus", "success");
} else {//このエラーを修正できませんでした
 if(FS.get("notFixErrors") == null) {
  var vector = new java.util.Vector();
  vector.add(FS.get("lastError"));
  FS.put("notFixErrors", vector);
    //修正されていないエラーすべてを自分のカスタム・プロパティーに保存します
 } else {
  var vector = FS.get("notFixErrors");
  vector.add(FS.get("lastError"));
  FS.put("notFixErrors", vector);
    //修正されていないエラーすべてを自分のカスタム・プロパティーに保存します
 }
 FS.put("lastCallStatus", "fail");
}
```

最後に、スクリプト・コンポーネント内の以下のコードを考えてみます。

```
main.logmsg("AL Cycle status: " + FS.get("lastCallStatus"));
//この AL サイクルの AL の状況を出力します
//AL の実行中に発生したすべてのエラーを、自分のカスタム・プロパティー
// 「vector」を通して報告することもできます
```

スクリプト記述の制御点

AssemblyLine 内のスクリプト記述

AssemblyLine は、事前にプログラミングされた標準動作を提供します。この標準動作とは異なる動作を使用するには、スクリプトを記述して独自のビジネス・ロジックをインプリメントします。

スクリプト・コンポーネント

AssemblyLine に、コネクターの他にスクリプト・コンポーネントを追加するには、AssemblyLine エディターの「AssemblyLine のコンポーネント」ウィンドウの適切なコンポーネントまたはセクションで、「**コンポーネントの挿入... (Insert Components...)**」 > 「**スクリプト**」 > 「**スクリプト**」を右クリックします。スクリプト・コンポーネントは、AssemblyLine 内の任意の場所に配置でき、AssemblyLine が各項目を処理するたびに 1 回開始されます。

注: AssemblyLine 内でスクリプト・コンポーネントをイテレーターの前に配置しても、イテレーターが最初に処理されます。

詳しくは、 24 ページの『スクリプト・コンポーネント』を参照してください。

AssemblyLine フック

AssemblyLine フック (つまり、個々のコンポーネントではなく、AssemblyLine 全体に適用されるフック) は、AssemblyLine の「**フック**」タブ内に表示されます。これ

らのフックはすべて、AssemblyLine の実行ごとに 1 回だけ実行されます。「**シャットダウン要求**」の場合は、何らかの外部プロセスが AssemblyLine に対してシャットダウンするように指示するたびに、フックが実行されます。ただし、AssemblyLine を複数回 (例えば、AssemblyLine コネクターを使用して) 開始した場合は、フックも複数回開始されます。

コネクター内のフックは、そのフックが定義されているコネクターの実行時にのみ評価され、実行されます (定義済みで、空でない場合)。詳しくは、75 ページの『コネクター内のスクリプト記述』を参照してください。

サーバー・フック

サーバー・フックにより、JavaScript コードを書き込み、サーバー・レベルで発生するエラーやイベントに応答できます。 AssemblyLine およびコンポーネント・フックとは異なり、サーバー・フックは別個のスクリプト・ファイルに保管されます。これらのファイルは、現行のソリューション・ディレクトリーの *serverhooks* フォルダーで保持され、明確に指定されたスクリプト関数を含む必要があります。 IBM Security Directory Integrator のサーバーおよび構成インスタンスは、サーバー・レベルのカスタム・フックを呼び出すために、IBM Security Directory Integrator コンポーネント用のメソッドを提供します。サーバーのフックは、スクリプト・ファイルに定義されている関数名です。関数は、ソリューション・ディレクトリーの「serverhooks」ディレクトリーにスクリプト・ファイルをドロップするだけでインプリメントできます。

特定のイベント発生時にサーバーにより呼び出されるフックに加えて、ユーザーのスクリプトから呼び出すこともできます。これらのフックへの呼び出しは同期化され、マルチスレッド化する可能性の問題を回避します。

開始時、IBM Security Directory Integrator は *serverhooks* サブディレクトリーにあるすべてのユーザー・スクリプトをロードおよび実行します。スクリプトは、関数宣言を含む場合と含まない場合があります。関数宣言がないスクリプトは、その開始時に 1 回実行されます (構成インスタンスが開始される前)。標準的な IBM Security Directory Integrator サーバー・フック関数を定義するコードは接頭部が「SDI_」となっており、操作中のさまざまなポイントで実行されます。

すべての IBM Security Directory Integrator サーバー・フック関数には、以下の JavaScript シグニチャーがあります。

```
/**
   * @param NameOfFunction The configuration instance invoking the function
  * @param source The component invoking the function
  * @param user Arbitrary parameter information from the source
  */
 function SDI_functionName(Name_of_function, source, user) {
 }
```

「NameOfFunction」パラメーターおよび「source」パラメーターは、それぞれ常に構成インスタンスと呼び出し側コンポーネントへのアクセスを提供します。「user」パラメーターは、各種フック関数においてさまざまな目的で使用されます。

次の標準機能名が、多くの IBM Security Directory Integrator コンポーネントにより呼び出されます。

関数名	呼び出し元 (ソース)	「user」パラメーターと期待される値
SDI_ALStarted	構成インスタンス	AssemblyLine の開始時に呼び出されます。 user = 開始された AssemblyLine *戻り値は無視されます*
SDI_ALStopped	構成インスタンス	AssemblyLine の停止時に呼び出されます。 user = 停止した AssemblyLine *戻り値は無視されます*
SDI_ConfigStarted	サーバー	構成インスタンスの開始時に呼び出されます。 user = 構成インスタンス *戻り値は無視されます*
SDI_ConfigStopped	サーバー	構成インスタンスの停止後に呼び出されます。 user = 構成インスタンス *戻り値は無視されます*
SDI_Shutdown	サーバー・インスタンス/構成インスタンス	IBM Security Directory Integrator サーバーが JVM (System.exit() など) を終了する直前に呼び出されます。 user = 終了状況 (整数) *戻り値は無視されます*

IBM Security Directory Integrator サーバー・フック関数へのアクセスは、`main.invokeServerHook()` メソッドにより実行できます。この関数は同期化され、1 回に複数のスレッドが 1 つのフックを実行することを回避します。 すべての呼び出しは同期的に実行されるため、呼び出し元は関数の戻りを待機します。結果として、サーバー・フックであまり多くの時間を消費しないよう注意してください。

前述のように、ソリューション・ディレクトリーの「serverhooks」サブディレクトリーにファイルを作成することで、スクリプトを定義して使用可能にできます。機密情報を含むスクリプトは、ディレクトリーに追加する前にサーバー API を使用して暗号化する必要があります。スクリプト・ファイルの暗号化には、`serverapi/cryptoutils` ツールを使用できます。IBM Security Directory Integrator は、拡張子が .jse のファイルを自動的に暗号化解除するので、暗号化したファイルにはこの拡張子を付加することをお勧めします。

また、serverhooks ディレクトリーにあるファイルは、プラットフォームの標準照合シーケンスにより大文字小文字を区別してファイル名を最初にソートした後にロードおよび実行されます。最上位ディレクトリーにあるすべてのファイルは、サブディレクトリー内のファイルが処理される前にロードされます。

サーバー・フックの一部の使用例を以下に示します。

- 自身のスクリプトで使用するために IBM Security Directory Integrator で常にロードするカスタム・オブジェクトを、IBM Security Directory Integrator 開始時にサーバー・フックにフックされた JavaScript スニペットからインスタンス化できる。これにより、構成ブラウザーの Java ライブラリーのフォルダー内で単にクラスを参照する場合より、多くの制御を獲得できます。
- そのイベントの監査ログを作成する、または何らかのトランスポート (SNMP、HTTP、JMS など) を使用してそのイベントを他のシステムに伝搬する、1 つ以上のカスタム AL を開始する。
- 構成のロード時または AL の開始時に常に呼び出す、企業のセキュリティー・ポリシーをインプリメントする。

スクリプトからのサーバー・フックの呼び出し: com.ibm.di.server.RS クラス (スクリプト変数「main」) には、サーバー・フックを呼び出すためのメソッドがあります。

```
/**
 * Invokes a server hook.
 *
 * @param name The name of the hook (also the filename)
 * @param caller The object invoking the hook
 * @param userInfo Arbitrary information to the hook from the caller
 */

public Object invokeServerHook(
 String name,
 Object caller,
 Object userInfo) throws Exception;
```

この呼び出しは、Java オブジェクト (任意のタイプ) を戻すことができます。そのため、IBM Security Directory Integrator がサーバー・フックの実行中にこれを無視しても、ユーザーのスクリプト呼び出しにおいて戻り値を使用できます。

AssemblyLine のスクリプトでは、次のように記述できます。

```
main.invokeServerHook("MyCustomHook", this, "custom information");
```

AssemblyLine 内部での AL コンポーネントへのアクセス

各 AL コンポーネントは、そのコンポーネントに対して指定した名前を使用した事前登録済みのスクリプト変数として使用できます。

system.getConnector() のような、関数へのスクリプト記述呼び出しによりコンポーネントを動的にロードできます。ただし、この方法は、経験の浅いユーザーの方にはお勧めしません。[8]

8. この呼び出しによって取得できるコネクター・オブジェクトは、コネクター・インターフェース・オブジェクトです。これは、AssemblyLine コネクターのデータ・ソースに固有の部分です。コネクターのタイプ を変更すると、そのコネクターのデータ・ソース・インテリジェンス(特定のシステム、サービス、またはデータ・ストアのデータにアクセスするための機能を提供している部分で、

AssemblyLine でのパラメーターの引き渡し

AssemblyLine にデータを取り込む方法としては、次の 3 つの方法があります。

- 独自の初期項目を AssemblyLine 内で生成する (プロローグ・スクリプトなど)。
- 1 つ以上のイテレーターからデータの提供を受ける。
- AL コネクター、AL 関数コンポーネント、または API 呼び出しを使用して、別の AssemblyLine のパラメーターを指定して AssemblyLine を開始する。

AssemblyLine の開始時に別の AssemblyLine のパラメーターを指定する場合は、いくつかのオプションがあります。

- **タスク呼び出しブロック** (TCB) を使用する (推奨方法)。TCB の詳細については以下のとおりです。
- 初期作業項目を直接提供する。

注: これらの 2 つのオプションは、以前のバージョンとの互換性を維持する目的で用意されています。

タスク呼び出しブロック (TCB)

タスク呼び出しブロック (TCB) は、呼び出し元で AssemblyLine の多数のパラメーターの設定のために使用される特殊な項目オブジェクトです。

基本的な使用法: タスク呼び出しブロック (TCB) は、呼び出し元で AssemblyLine の多数のパラメーターの設定のために使用される特殊な項目オブジェクトです。TCB は、AssemblyLine により指定された入力パラメーターと出力パラメーターのリスト (「**操作**」タブに定義されている命令コードを含む) を提供できるほか、AssemblyLine コネクター用のパラメーターを呼び出し元で設定することを可能にします。TCB は、以下の論理セクションで構成されています。

- AssemblyLine に渡される初期作業項目: `tcb.setInitalWorkEntry()`
- コネクター・パラメーター: `tcb.setConnectorParameter()`
- AssemblyLine の入力または出力マッピング・ルール (構成エディターの「**操作**」タブに設定されている内容)
- AssemblyLine からすべての作業項目を受け取る、ユーザー指定のアキュムレーター・オブジェクト (オプション): `tcb.setAccumulator()`

例えば、初期作業項目を指定して AssemblyLine を開始し、MyInput という名前のコネクターの *filePath* パラメーターに d:¥myinput.txt を設定するには、次のコードを使用します。

```
var tcb = system.newTCB();                          // Create a new TCB
var myIWE = system.newEntry();                      // Create a new entry object
myIWE.setAttribute("name","John Doe");              // Add an attribute to myIWE
tcb.setInitialWorkEntry ( myIWE );                  // Set the IWE and parameters
// Note that since this is a JavaScript string, we must "escape" the forward slash
//      or use a backslash (Windows syntax)
tcb.setConnectorParameter ( "MyInput", "filePath", "d:¥¥myinput.txt" );
```

コネクター・インターフェースとも呼ばれます) がスワップアウト(交換) されることになります。 AssemblyLine コネクターの機能の大部分 (属性マップ、リンク基準、フックなど) は IBM Security Directory Integrator カーネルによって提供されており、コネクターのタイプを変更してもそのまま保持されます。

```
var al = main.startAL ( "MyAssemblyLine", tcb ); // Start the AL with the tcb
al.join();                                       // Wait for AL to finish
```

操作を設定した AssemblyLine の開始: AssemblyLine は*操作* を使用して定義できます。これは、AssemblyLine に対して複数の入力マップが定義されるという概念です。AssemblyLine の呼び出し方法に応じて、アクティブになる入力マップは異なります。 AssemblyLine 内部では、op-entry を調べてどの操作がアクティブであるかどうかを確認し、ブランチ・コンポーネントを使用して AssemblyLine 内部から該当する操作へのフローを調整する必要があります。

操作を設定して AssemblyLine を開始する方法の 1 つに、TCB とスクリプト・コードを使用する方法があります。

「al1」という名前の AssemblyLine に、「Default」、「Op1」、および「Op2」という操作があり、このスクリプトでは操作を「Op1」に設定して AL を開始するとします。

```
var tcb = system.newTCB("al1");
tcb.setALOperation("Op1");
main.startAL(tcb);
```

操作を指定しないと、操作を「Default」に設定して AL を開始します。

```
var tcb = system.newTCB("al1");
main.startAL(tcb);
```

AL に Default 操作がない場合 (「Op1」と「Op2」操作のみの場合)、2 番目のスクリプトでは例外がスローされます。

AssemblyLine 操作の概念について詳しくは、「*リファレンス*」の『アダプターを使用した新規コンポーネントの作成』を参照してください。

アキュムレーターの使用: 既に説明したように、TCB を使用して、AssemblyLine にアキュムレーター・オブジェクトを渡すこともできます。アキュムレーターは、以下のクラスまたはインターフェースのうちのいずれかです。

java.util.Collection
すべての作業項目が複製され、コレクション (ArrayList、Vector など) に追加されます。

com.ibm.di.connector.ConnectorInterface (コネクター・インターフェース)
このコネクター・インターフェースの putEntry() メソッドは、各 AssemblyLine サイクルの最後に作業項目とともに呼び出されます。

com.ibm.di.parser.ParserInterface (パーサー)
このパーサーの writeEntry() メソッドは、各 AssemblyLine サイクルの最後に作業項目とともに呼び出されます。

com.ibm.di.server.AssemblyLine Component (AssemblyLine コネクター)
この AssemblyLine コネクターの add() メソッドは、各 AssemblyLine サイクルの最後に作業項目とともに呼び出されます。

アキュムレーターがこれらのクラスまたはインターフェースのいずれかでない場合は、例外が戻されます。

例えば、1 つの AssemblyLine のすべての作業項目を XML ファイルに累積するには、次のスクリプトを使用できます。

```
var parser = system.getParser ( "example_name.XML" ); // Get a Parser
// Set it up to write to file
parser.setOutputStream ( new java.io.FileOutputStream ( "d:/accum.xml" ));
parser.initParser();                                  // Initialize it.
tcb.setAccumulator ( parser );                        // Set Parser to tcb

var al = main.startAL ( "MyAssemblyLine", tcb );      // Start AL with tcb
al.join();                                            // Wait for AL to finish

parser.closeParser(); // Close the parser - this flushes and
             // closes the output file
```

同様に、パーサーを手動でプログラミングする代わりに、次のスクリプトのようにコネクターを構成することもできます。

```
var connector = system.getConnector("myFileSysConnWithXMLParser");
tcb.setAccumulator ( connector );

var al = main.startAL( "MyAssemblyLine", tcb);
al.join();

connector.terminate();
```

通常、TCB を初期化した後、TCB は AssemblyLine により使用されます。AssemblyLine に操作の指定がある場合、TCB は、AssemblyLine が予期している形式になるように入力属性を初期作業項目に再マップします。結果オブジェクトの設定も同様に処理されます。 この処理を実行するのは、AssemblyLine 内の内部作業項目名が変更されても、AssemblyLine への外部呼び出しインターフェースが変更されないようにするためです。 TCB が AssemblyLine に渡された後には、TCB から出力を受け取ることはできません。結果オブジェクトと統計を検索するには、AssemblyLine の `getResult()` と `getStats()` を使用します。

TCB の結果マッピングはエピローグより前に実行されるため、AssemblyLine の呼び出し側が最終結果に到達する前であれば、最終結果にアクセスできます。

AssemblyLine コンポーネントの使用不可化: IBM Security Directory Integrator では、特定の AssemblyLine コンポーネントを AssemblyLine の初期化時に作成も初期化もしないように指定できます。このためには、TCB を使用して該当するコンポーネントを使用不可にします。

AssemblyLine コンポーネントは、デフォルトで使用可能になっています。

AssemblyLine 内でコンポーネントを使用可能/使用不可にするには、AssemblyLine の TaskCallBlock (TCB) オブジェクトで `com.ibm.di.server.TaskCallBlock.setComponentEnabled(String name, boolean enabled)` メソッドを呼び出す必要があります。メソッドの name 引数は使用可能または使用不可にするコンポーネントの名前を指定します。メソッドの enabled 引数はコンポーネントを使用可能にするのか、使用不可にするのかを指定します。

AssemblyLine コンポーネントの実際の使用可能化または使用不可化は、`com.ibm.di.server.TaskCallBlock.applyALSettings(AssemblyLineConfig alc)` メ

ソッドで行われます。このメソッドは、AssemblyLine の初期化時に呼び出されます。使用不可を指定したコンポーネントは、AssemblyLine の初期化の進行中に作成/初期化されません。

LOOP コンポーネントが使用不可にされた場合、その LOOP に含まれるすべてのコンポーネントも使用不可になります。

コンポーネントが構成エディターにより使用不可にされた場合でも、以下のスクリプト呼び出しを使用して使用可能にできます。

```
com.ibm.di.server.TaskCallBlock.setComponentEnabled(String name, boolean enabled)
```

初期作業項目 (IWE) の提供

TCB を使用してパラメーターを渡す代わりに、初期作業項目 (IWE) を指定できます。この方法は、以前のバージョンとの後方互換性を維持するためにサポートされています。

スクリプトで **system.startAL()** 呼び出しを実行して AssemblyLine を開始する場合、AssemblyLine にパラメーターを渡すには、初期作業項目に属性値またはプロパティー値を設定します。初期作業項目にアクセスするには、*work* 変数を使用します。その後、これらの値を適用して、コネクター・パラメーターを設定します。例えば、AssemblyLine の「**プロローグ - 開始**」フック内で *connectorName*.setParam() 関数を使用します。

注: task.setWork(null) 呼び出しを使用して作業項目をクリアする必要があります。クリアしないと、AssemblyLine 内のイテレーターが最初のサイクルでパススルーします。

getResult() 関数を使用して、AssemblyLine の結果 (AssemblyLine が停止した時点の作業項目) を確認できます。「リファレンス」の『ランタイム提供コネクター (Runtime provided Connector)』も参照してください。

IWE を使用してコネクターのパラメーター値を渡す例を次に示します。

```
var entry = system.newEntry();
entry.setAttribute ("userNameForLookup", "John Doe");

// Here we start the AssemblyLine
var al = main.startAL ( "EmailLookupAL", entry );

// wait for al to finish
al.join();

var result = al.getResult();

// assume al sets the mail attribute in its working entry
task.logmsg ("Returned email = " + result.getString("mail"));
```

コネクター内のスクリプト記述

入力マップと出力マップ

これらのタブで、カスタム属性マッピングを実行します。属性を選択するときには、「**拡張マッピング**」チェック・ボックスを選択し、編集ウィンドウ

にスクリプトを入力する必要があります。 必要な処理をすべて完了した後で、取得された結果の値を ret.value に割り当てる必要があります。例を示します。

```
   ...
ret.value = myResultValue;
```

あるいは IBM Security Directory Integrator で、キーワード *return* とその後に結果として渡す単純値を使用します。

```
return "mystring";
```

コネクター・フック

フックを使用すると、発生した特定のイベントに応答したり、コネクターの基本的な機能を指定変更したりできます。フックのスクリプトを記述するときには、グローバル・オブジェクトにアクセスできます。ただしフックの種類によっては、一部の標準オブジェクトにアクセスできないこともあります。一時オブジェクトが使用可能かどうかについて詳しくは、「リファレンス」の『AssemblyLine およびコネクター・モードのフローチャート』を参照してください。また、環境、AssemblyLine、コネクター、項目、および属性を完全に制御できます。フックは、プロセス・フローをカスタマイズするためにさまざまな制御点を提供します。 34 ページの『AssemblyLine フローおよびフック』を参照してください。

スクリプト記述による内部パラメーターの設定

以下のスクリプトを使用して、コネクターの接続パラメーターを設定できます。

```
myConnector.setParam ( "filePath", "examples/scripting/sample.csv" );
```

一般にこのような処理はプロローグで実行しますが、AssemblyLine の実行中にも非常に役立つ場合があります。ただし、コネクターの停止と再初期化をユーザーが行うことが条件です。

```
myConnector.terminate();
myConnector.setParam ( "filePath", "examples/scripting/sample.csv" );
myConnector.initialize(null);
```

パーサー内のスクリプト記述

パーサー内のスクリプト記述とは、具体的には、スクリプト記述によってユーザー独自のパーサーをインプリメントすることです。このプロセスについて詳しくは、「リファレンス」の『スクリプト・パーサー』を参照してください。

Java + Script ≠ JavaScript

JavaScript は、Java では**ありません**。 Java に類似しており、混同する場合があります。Netscape が初めて作成された頃は、JavaScript は *Live!Script* と呼ばれていました。JavaScript には幅広いサポートがありますが、Microsoft のバージョンである *JScript* など、ダイアレクト が存在することが分かります。 *ECMAScript* として知られる標準的な定義がありますが、その仕様については、次の URL を参照してください。 http://www.ecma-international.org/

構文は類似していますが、Java と JavaScript では、データとデータ・タイプの取り扱いは異なります。これが混乱の主な原因の 1 つとなっており、これにより、JavaScript を使用する際にエラーが発生します。

データの表現

Java は、符号付き整数、10 進数、単一のバイトなどの単一値であるプリミティブをサポートします。プリミティブでは、機能は提供されず、複素数以外のデータ内容のみです。 プリミティブを計算や式で使用し、変数を割り当てて関数呼び出しで渡すことができます。Java では、データ内容 (高度な複素数データも含む) を実行するだけではなく、メソッドとしても知られるオブジェクト関数の形でインテリジェンスも提供する豊富なオブジェクトもまた提供します。

`task.logmsg( "Hello, World" )` へのスクリプトの呼び出しを行うと、タスク・オブジェクトの `logmsg()` メソッドが呼び出されます。

Java の多くのプリミティブは、対応するオブジェクトを持ちます。その 1 つの例が整数です。整数は、プリミティブ形式 *(int)* で使用することも、`java.lang.Integer` オブジェクトの操作で使用することもできます。

JavaScript には、プリミティブの概念は適用されません。代わりに、すべてのデータを JavaScript オブジェクトとして表します。また、Java の豊富なデータ語と比較すると、JavaScript には、ごく少数のネイティブ・オブジェクトしかありません。したがって、Java が分数以外の数値オブジェクトとその小数部を区別している (符号ありと符号なしの間も区別し、精度レベルの異なる類似オブジェクトを提供している) のに対し、 JavaScript では、すべての数値を *Number* というオブジェクト・タイプで一括りにしています。

結果として、JavaScript で数値を比較すると、結果はエラーになります。

```
if (myVal == 3) {
 // do something here if myVal equals 3
}
```

`myVal` は、算術演算、または Java 10 進数オブジェクトの参照で設定されると、オブジェクトの値は、3.00001 または 2.99999 です。この値は、3 に非常に近い値ですが、上記の等価テストは通過できません。こうした問題を避けるために、オペランドを Java の整数オブジェクトに変換して符号付きの分数以外の値にします。これにより、ブール式が期待どおりに動作します。

```
if (java.lang.Integer( myVal ) == 3) { ...
```

または、変数が適切な Java オブジェクトを参照して開始することを確実にできます。一般に、使用しているオブジェクトのタイプには注意が必要です。

あいまいな関数呼び出し

Java では、*char* というプリミティブ型が提供されます。この char には、単一文字値が含まれます。文字のコレクションは、文字プリミティブの配列として、Java で表されるか、`java.lang.String` オブジェクトとして処理されます。前述のように JavaScript は、プリミティブを使用できません。文字データは、JavaScript のストリング・オブジェクトを使用して処理する必要があります。 JavaScript の単一文字 ("a") を指定していても、*string* と考慮されます。

Java 関数をスクリプトから呼び出す場合を考えてみます。関数の名前、および呼び出しで使用するパラメーターの数およびタイプを用いて実際のメソッドに対して一致させます。このマッチングは、LiveConnect から JavaScript への拡張により行われます。LiveConnect は、参照している関数シグニチャーを検索します。これは、Java と JavaScript がそれぞれ別の方法でパラメーター・タイプを表すことによる、小さくないタスクです。しかし、JavaScript と LiveConnect は、内部変換を行い、Java と JavaScript のデータ・タイプの一致を試行します。

1 つのメソッドに複数のバージョンがある、つまりそれぞれが異なるパラメーター・セットを取ると、問題が発生します。特に、2 つの関数において、パラメーターの数が同じであるがタイプが異なる場合、**および**、タイプが JavaScript で区別されない場合です。ストリングの MD5 暗号化を実行するスクリプトの例を見てみます。

```
// Create MessageDigest object for MD5 hash
var md = new java.security.MessageDigest.getInstance( "MD5" );

// Get the EID attribute value as byte array.
var ab = java.lang.String( "message to encrypt" ).getBytes();

md.update( ab );

var retHash = md.digest();
```

上記の `update()` 呼び出しは、関数呼び出しがあいまいであるという評価例外により失敗します。これは、`MessageDigest` オブジェクトが、この関数について複数のバージョン、つまり一方は単一のバイトを受け入れ、もう一方はバイト配列 `(byte[])` を期待するなど、類似したシグニチャーを持つことが原因です。異なる数のパラメーターを取る別のタイプがこのメソッドにあり、一意的に識別可能なシグニチャーを指定できる場合、この問題を回避できます。幸いにして、`MessageDigest` では、バイト配列と一組の数値 (オフセットと長さパラメーター) を取る `update()` のバージョンが提供されます。したがって、コードを変更して、この呼び出しの代わりに使用できます。

```
md.update( ab, 0, ab.length );
```

オブジェクトの後の大括弧内を引用符で囲むことにより、使用する Java メソッドの正確なシグニチャーを常時指定できるようになりました。

```
md["update(byte[])"](ab);
```

これは、単一のバイト配列パラメーターで宣言した `update()` 関数のバージョンを呼び出しています。

Java と JavaScript ストリングにおける文字/ストリングのデータ

Java と JavaScript は、両方ともストリング・オブジェクトを提供します。この 2 つのタイプのストリング・オブジェクトの動作は類似しており、一部類似した関数も提供しますが、実際は大きく異なります。例えば、オブジェクト・タイプはそれぞれ、ストリング長を戻すために、別々のメカニズムを使用します。Java ストリングでは、`length()` メソッドを使用します。JavaScript ストリングには、`length` 変数があります。

```
var jStr_1 = new java.lang.String( "Hello, World" ); // Java String
task.logmsg( "the length of jStr_1 is " + jStr_1.length() );

var jsStr_A = "Hello, World"; // JavaScript String
task.logmsg( "the length of jsStr_A is " + jsStr_A.length );
```

この微妙な違いが、原因不明の構文エラーを引き起こす場合があります。例えば、`jsStr_A.length()` を呼び出そうとすると、ランタイム・エラーになります。これは、このオブジェクトに `length()` メソッドがないことが原因です。

ストリングの比較では、さらにわかりにくい誤りが発生する場合があります。

```
var jsStr_A = "Hello, World"; // JavaScript String
var jsStr_B = "Hello, World"; // JavaScript String

if ( jsStr_A == jsStr_B )
 task.logmsg( "TRUE" );
else
 task.logmsg( "FALSE" );
```

予想どおり、上記の結果は、「TRUE」です。ただし、Java ストリングの場合は、以下のように多少異なります。

```
var jStr_1 = java.lang.String( "Hello, World" ); // Java String
var jStr_2 = java.lang.String( "Hello, World" ); // Java String

if ( jStr_1 == jStr_2 )
 task.logmsg( "TRUE" );
else
 task.logmsg( "FALSE" );
```

この結果は「FALSE」です。理由は、上記の等価演算子が、値のマッチングではなく、両方の変数がメモリー内で*同じオブジェクト* を参照しているかどうかを確認して比較するためです。Java ストリング値を比較するには、適切なストリング・メソッドを使用する必要があります。

```
if ( jStr_1.equals( jStr_2 ) ) ...
```

さらに詳細に説明します。次のコードの例では、結果は「TRUE」になります。

```
var jsStr_A = "Hello, World"; // JavaScript String
var jStr_1 = java.lang.String( "Hello, World" ); // Java String

if ( jsStr_A == jStr_1 )
 task.logmsg( "TRUE" );
else
 task.logmsg( "FALSE" );
```

JavaScript では、Java のストリング・オブジェクトなどの不明なタイプを計算できないため、まず `jStr_1` を等価の JavaScript ストリングに変換し、計算を実行します。

つまり、作業するオブジェクト・タイプについてよく理解しておく必要があります。そして、IBM Security Directory Integrator 関数は、常に Java オブジェクトを戻します。このことを覚えておくと、スクリプト・コードのエラーを最小限にできます。

変数のスコープと命名

JavaScript は、相対的には略式の言語であり、値を割り当てる前に変数を定義する必要はありません。厳しいタイプ・チェックを強制されることもなく、変数を再定義する際に通知されることもありません。これにより、JavaScript を速く簡単に使用できる反面、判読不能なコードやわかりにくいエラーが起こりやすくなります (特に、組み込みの変数を上書きする変数が作成できるためです)。

バグがデバッグを抜けてしまうのは、組み込み変数、例えば `work`、`conn`、および `current` と同じ名前で変数を宣言した場合です。したがって、IBM Security Directory Integrator が使用している予約名についてもよく理解しておく必要があります。

それ以外の共通する問題は、組み込みの構成やスクリプト・ライブラリーで使用される、既存の変数を再定義する変数をユーザーが新たに作成した場合に発生します。こうした誤りは、変数の命名とスコープ に注意することで回避できます。スコープは、変数の影響範囲を定義します。 IBM Security Directory Integrator では、すべてのフック、スクリプト・コンポーネント、および属性マップで使用可能なグローバル変数と、関数に対してローカルな変数について説明します。

スコープを詳細に理解するためには、まず、すべての AL が独自のスクリプト・エンジンを持つため、独自のスクリプト・コンテキストで実行することを理解する必要があります。関数宣言で特にローカルに定義されている変数以外は、そのスクリプト・エンジンに対してグローバルです。したがって、グローバル変数を作成するには、次のコードを使用します。

```
myVar = "Know thyself";
```

この変数は、この時点の AL から使用可能です。この変数をローカルにするには、2 つのステップがあります。まず、変数を宣言するときに、`var` キーワードを使用します。次にその宣言を関数内に含めます。

```
function myFunc() {
 var myVar = "Know thyself";
}
```

ここで、上記で定義した `myVar` は、閉じ中括弧で終了します。関数内に変数を配置するだけでは、十分ではありません。`var` を使用し、新規のローカル変数を宣言していることを示す必要があります。

```
var glbVar = "This variable has global scope";
glbVar2 = "Another global variable";

function myFunc() {
    var lclVar = "Locally scoped within this block";
    glbVar3 = "This is global, since "var" was not used";
};
```

関数内でローカル変数を宣言すると、いつでも呼び出すことができます。変数がスコープから外れると、それまでのタイプと値が復元されます。

`var` キーワードはグローバル変数を定義するためには不要ですが、このキーワードを定義しておくことがベスト・プラクティスです。また、変数をスクリプトの一番上で定義することをお勧めします。その際は、その変数が何を目的に作成されたか

を利用者が理解できるようなコメントを含めておくことも大切です。 この方法により、コードがより読みやすくなり、また、変数の命名とスコープに関する意識的な判断が必要になります。[9]

Java クラスのインスタンス化

スクリプト・コードを使用して、外部 Java クラス (IBM Security Directory Integrator ランタイム環境に追加したクラス) または CLASSPATH を呼び出すことができます。

例えば、標準 `java.io.FileReader` を使用する場合は、次のスクリプトを使用します。

```
var javafile = new java.io.FileReader( "myfile" );
```

これで、*javafile* というオブジェクトが作成されました。このオブジェクトを使用すると、オブジェクトのすべてのメソッドを呼び出すことができます。

同じ方法を使用してユーザー独自のオブジェクトをインスタンス化できます。

```
var myfile = new my.FileReader("myfile");
```

スクリプト記述内でのバイナリー値の使用

バイナリー値を属性から取得するには、項目の `getObject()` 関数を使用します。バイナリー属性値は、バイト配列として戻されます。 JavaScript の例を示します。

```
var x = conn.getObject("objectGUID");
for ( i = 0; i < x.length; i++ )
{
   task.logmsg ("GUID[" + i + "]: " + x[i]);
}
```

この例では、-128 から 127 までのいくつかの数値がログ・ファイルに書き込まれます。 データに対して他の処理を実行することもできます。コネクターが ByteArray として格納していたパスワードを読み取った場合は、次のコードを使用して、このパスワードをストリングに変換できます。

```
password = system.arrayToString(conn.getObject("userpassword"));
```

スクリプト記述内での日付値の使用

IBM Security Directory Integrator で日付を操作する場合、`java.util.Date` のインスタンスが使用されます。使用可能なスクリプト言語のいずれかを使用して、日付を処理するためのユーザー独自の機構をインプリメントできます。ただし、これは一般的な方法ではありません。

IBM Security Directory Integrator スクリプト記述エンジンは、日付を構文解析するための機構を提供します。 *system* オブジェクトには、常にアクセス可能な `parseDate(date, format)` メソッドがあります。

---

9. 関数の命名は、多少異なります。 Java などのプログラム言語は、その名前とパラメーターの数とタイプを組み合わせて関数を識別します。JavaScript は名前のみを使用します。したがって、同じ関数を複数定義すると、その定義に異なる数のパラメーターがあるかどうかに関係なく、 JavaScript では、最後の関数だけが「記憶」されます。

注: java.util.Date のインスタンスを取得すると、標準 Java ライブラリーおよびクラスを使用して処理を拡張できます。

日付を処理する単純な JavaScript の例を示します。このコードは、任意のスクリプト記述制御点に配置して開始できます。

```
var string_date1 = "07.09.1978";
var date1 = system.parseDate(string_date1, "dd.MM.yyyy");

var string_date2 = "1977.02.01";
var date2 = system.parseDate(string_date2, "yyyy.dd.MM");

task.logmsg(date1 + " is before " + date2 + ": " +
  date1.before(date2));
```

このスクリプト・コードは、最初に (異なる形式の) 2 つの日付値を構文解析して java.util.Date に変換します。次に標準 java.util.Date.**before()** メソッドを使用して、第 1 の日付インスタンスが第 2 の日付インスタンスより前かどうかを判別します。次に、このスクリプトの出力をログ・ファイルに出力します。

スクリプト記述内での浮動小数点値の使用

以下の例は、ユーザーが作成するスクリプト・コードの内部で浮動小数点値をどのように使用できるかを示します。以下の例は、すべて JavaScript 内でインプリメントされます。他のいくつかのスクリプト言語を使用して同じ例が繰り返される場合もありますが、構文は異なっていることがあります。次の単純なスクリプトは、2 つの変数に浮動小数点値を割り当てて、その平均を求めます。このコードは、任意のスクリプト記述制御点から開始できます。ログ・ファイルへの出力は " r= 3.85 " です。

```
var a = 5.5;
var b = 2.2;
var r = (a + b) / 2;
task.logmsg("r = " + r);
```

次の例は、この単純なスクリプトを拡張したものです。入力コネクター内に「Marks」という名前の多値属性があり、この属性に浮動小数点値を表す複数のストリング値 (java.lang.String) が格納されているとします。これは一般的な状況です。この属性は、出力コネクター内にある「AverageMark」という名前の属性にマップされます。この属性に、「Marks」属性の値の平均値が格納されます。次のコードは、「AverageMark」属性の拡張マッピングで使用されます。

```
// First return the values of the "Marks" attribute
var values = work.getAttribute("Marks").getValues();

// Zero out counter and sum variables
var sum = 0;
var count = 0;

// Loop through the values, counting and summing them
for (i=0; i<values.length; i++)
{
   // use the Double() function to convert value to number
   sum = sum + new Number(java.lang.Double(values[i]));
   count++;
}

// If count > 0, compute the average
```

```
var average = (count > 0) ? (sum / count) : 0;

// Return the computed average
ret.value = average;
```

この例で中心的な役割を果たす呼び出しは `java.lang.Double(values[i])` です。この呼び出しにより、「Marks」の現在の索引付き値が、平均の計算で使用できる値に変換されます。

第 3 章 構成エディター

IBM Security Directory Integrator Eclipse 構成エディター (CE) は、IBM Security Directory Integrator ソリューションを開発するための主要ツールです。構成ファイルの作成、保守、テスト、デバッグを実行できます。構成エディターは Eclipse プラットフォームをベースに構築されており、包括的かつ拡張可能な開発環境を提供します。

本書に示されている概念をより容易に理解するには、Eclipse の一般的なコンセプト (エディター、ビュー、およびその他の Eclipse 拡張ポイント) について理解しておく必要があります。

プロジェクト・モデル

Eclipse ベースの構成エディター (CD) の導入により、 IBM Security Directory Integrator (IBM Security Directory Integrator) でのソリューション開発は、 V7.0 以前のバージョンのようなプレーン構成ファイルの開発ではなく、プロジェクト・モデルとワークスペース に基づいて行われます。プロジェクト・モデルとワークスペースを使用して開発作業を実施できます。ソリューションをデプロイできる状態になったら、ワークスペースから構成ファイルを抽出して適切な IBM Security Directory Integrator サーバー (ランタイム環境) に送信します。これには、次に示すようなメリットがあります。

- 不要なエレメントが含まれていない構成ファイル: 不要な構成エレメントや未使用の構成エレメントはワークスペースからエクスポートされません。
- IBM Security Directory Integrator 構成の編集効率の向上
- 共通コンポーネントの再利用性の向上
- ソース・コード管理システム (CVS など) が使用可能

ワークスペース

CE を開始すると、ワークスペース・ディレクトリーを選択するように指示されます。ワークスペース・ディレクトリーには、すべての IBM Security Directory Integrator プロジェクトとファイルが CE により保管されます。これはソリューション・ディレクトリーとは異なりますが、ソリューション・ディレクトリーの中にワークスペース・ディレクトリーを作成することができます。CE ではワークスペースからファイルが収集され、IBM Security Directory Integrator サーバーで実行可能なランタイム構成ファイル (`rs.xml`) が作成されます。ワークスペースのファイルとプロジェクトは、各種ランタイム構成ファイルのソースとして見なすことができます。

インストール・ディレクトリー、ソリューション・ディレクトリー、および作業ディレクトリー

この 3 つのディレクトリーの役割を混同する場合があります。前のセクションで説明したように、ワークスペース・ディレクトリーは CE に属し、プロジェクト・ファイルが保管されるディレクトリーです。インストール・ディレクトリーとソリュ

ーション・ディレクトリーは、IBM Security Directory Integrator サーバーの実行時に TDI に属します。ただし、CE でコネクターを構成する場合などは、属性ディスカバー機能を使用することがよくあります。これにより、CE でコネクターが開かれ、コネクターに対して操作が実行されます。一部のコネクターではさまざまな目的でファイル名が使用されるため、相対パスを使用する際にわかりにくいことがあります。このため、CE のデフォルトの作業ディレクトリーは常に、主に使用するソリューション・ディレクトリーと同じである必要があります[10]。これは主に、CE 内部で複数のサーバーとソリューション・ディレクトリーが作成される可能性があるためです。これらのサーバーとディレクトリーを処理し、これらのサーバーで AssemblyLine を実行できますが、CE でコネクターを使用する場合の構成の相対パスは、AssemblyLine の実行時にコネクターを使用する場合とは異なります。

例えば、`abc.txt` を入力ファイルとして使用するファイル・コネクターの構成があるとします。 CE で属性をディスカバーする場合、CE の作業ディレクトリー (インストール時に指定された優先ソリューション・ディレクトリー) 内でファイルが検索されます。このコネクターを含む AssemblyLine を実行する場合、特に問題はありません。次に、現行作業ディレクトリー以外のディレクトリーを指し示すソリューション・ディレクトリーを使用して新規サーバーを作成します。コネクターを構成し、このコネクターによりファイル名が正しい場所 (現行作業ディレクトリー内の abc ファイル) に解決されます。ただし AssemblyLine を実行すると、ファイルを検出できないために失敗します。これは、新規サーバーが稼働する作業ディレクトリー (ソリューション・ディレクトリー) が、CE とは異なるためです。

デフォルトでは、CE の作業ディレクトリーは、AssemblyLine の実行に使用されるデフォルト・サーバーのソリューション・ディレクトリーに設定されます。このため、何も変更しなければ、新規ソリューション・ディレクトリーを作成するか、またはデフォルト・サーバーのソリューション・ディレクトリーを変更する場合を除いては、これらの問題は発生しません。

ワークベンチ

ワークベンチとは、IBM Security Directory Integrator ソリューションの構成と開発を行うための主要ユーザー・インターフェース・アプリケーションです。ワークベンチに含まれるさまざまなビューとエディターを使用して、これらの作業を実施できます。

IBM Security Directory Integrator サーバー・ビュー

以前のバージョン (7.0 より前) とは異なり、AssemblyLine を実行するたびに IBM Security Directory Integrator サーバー・インスタンスが始動されるわけではありません。代わりに、CE を開始すると IBM Security Directory Integrator サーバーが自動的に始動されます。CE では、開発した AssemblyLine をテストおよび実行できます。以前のバージョンとは対照的に、テスト実行のオブジェクトが完了してもこのサーバーは終了せずに、次のテスト実行を待機します。

---

10. 作業ディレクトリーとワークスペースを混同しないでください。ワークスペースとは、プロジェクト・ファイルが保管される領域であり、作業ディレクトリーとは、ファイル・システム上ですべての非完全修飾ファイル名が定義される位置です。移植性の理由から、作業ディレクトリーをソリューション・ディレクトリーとして使用してください。

「サーバー」ビューでは、サーバー定義を管理できます。このビューで定義されるサーバーはローカル・サーバーまたはリモート・サーバーであり、作成するさまざまな IBM Security Directory Integrator プロジェクトで使用されます。インストール・パスが定義されているサーバーは、CE が起動できる「ローカル」サーバーと見なされます。

1 つのサーバーが自動的に定義され、*Default* という名前が付きます。このサーバーは、新規 IBM Security Directory Integrator プロジェクトで使用されるデフォルト・サーバーです。デフォルト・サーバーの作成時には、グローバル・プロパティー・ファイルの値と、ユーザーが定義したソリューション・ディレクトリー (インストーラーによりセットアップされる `SDI_SOLDIR` 変数などから取得) が使用されます。

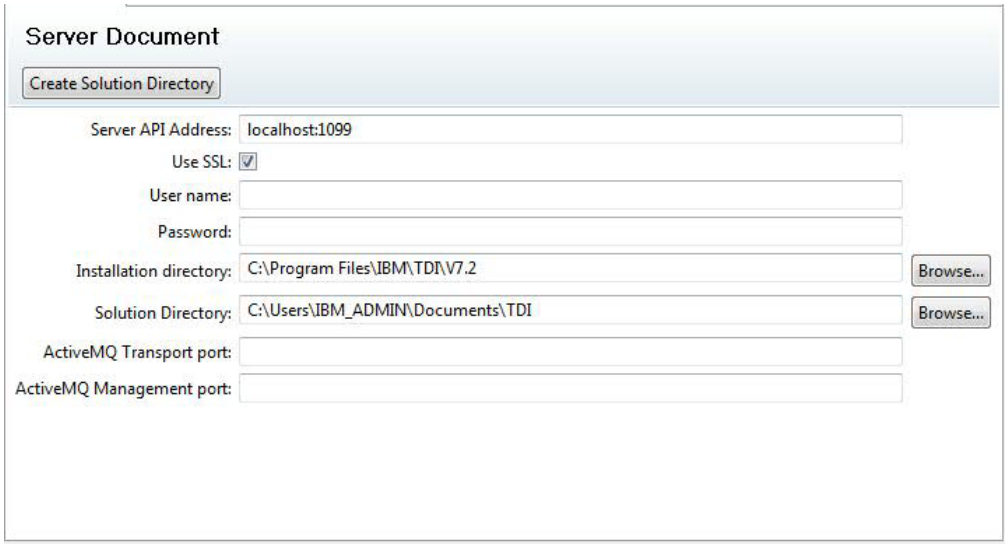


図 5. デフォルトの IBM Security Directory Integrator サーバー定義

IBM Security Directory Integrator プロジェクト

IBM Security Directory Integrator ソリューションを開発するには、最初に IBM Security Directory Integrator プロジェクトを作成する必要があります。これは Eclipse のプロジェクトであり、関連するファイルとリソースをまとめたものです。

プロジェクトの作成時に、共通構成オブジェクトが含まれているいくつかのフォルダーがプロジェクトに取り込まれます。新規 IBM Security Directory Integrator プロジェクトのレイアウトを次に示します。

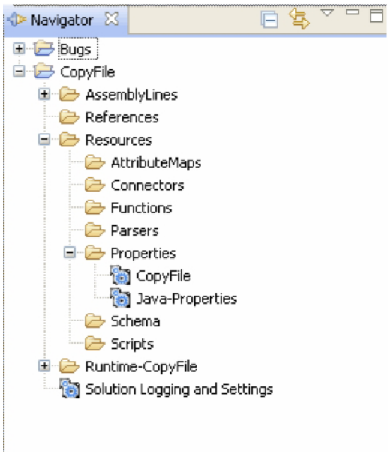


図 6. IBM Security Directory Integrator プロジェクト・ツリー

「AssemblyLine」フォルダーには、プロジェクトの AssemblyLine が含まれています。「リソース」フォルダーには、AssemblyLine で共用または使用されるか、または一般にソリューションに適用されるすべてのコンポーネント (プロパティー、ロギング・パラメーターなど) が含まれています。新規リソースを作成するには、メインメニューから「**ファイル**」/「**新規...**」ウィザードを使用するか、または各フォルダーのコンテキスト・メニューを使用します (新規 AssemblyLine を作成するには「AssemblyLine」フォルダーを右クリックします)。

「ランタイム」ディレクトリー (ランタイム-*ProjectName*) は、ランタイム構成ファイルとカスタム・プロパティー・ファイルが含まれている特殊フォルダーです。プロジェクトでコンポーネント (コネクター、AssemblyLine、プロパティー・ファイルなど) が変更されるたびに、このディレクトリーの関連するファイルが更新されます。生成されるすべてのファイルには、派生ファイルとしてタグが付けられます。つまり、ファイルの内容を変更しようとすると警告が出されます。これらのファイルを変更すると、ファイルが上書きされる点に注意してください。ランタイム・ディレクトリーは、ソース制御を使用する場合にコピーまたはチェックアウト可能であり、IBM Security Directory Integrator サーバーにより使用されるプロジェクトのランタイム・ファイルを格納するためのディレクトリーを作成するためのものです。

構成ファイル

サーバーによって実行される構成ファイル (Config) は、AssemblyLine、コネクターなどがすべて含まれている 1 つの複合 XML 文書です。IBM Security Directory Integrator Eclipse CE では、各コンポーネントに対して固有の物理ファイルが割り振られます。これらの各ファイルには 1 つの構成オブジェクトが含まれます。

ソリューション開発中に、構成ファイルを複数のファイルに分割する理由の 1 つに、コンポーネントの共用を容易にすることがあります。また、各コンポーネントをそれぞれ個別のファイルに含めると、CVS などのソース制御システムや、複数の担当者がソリューションの異なる部分を同時に開発するマルチユーザー開発環境で役立ちます。

このバージョンよりも古いバージョンの構成をインポートするには、インポート・ウィザードを使用します。このプロセスの結果として、インポートされた構成は、

個々の構成ファイルに分割されます。もう 1 つの方法として、「**ファイル**」 > 「**Security Directory Integrator 構成ファイルを開く...**」オプションを使用する方法があります。このオプションでは、7.0 以前の構成が新規プロジェクトにインポートされます。ただし、このオプションでは、自動更新がソース・ファイルに設定される点に注意してください。リンク・ファイル機能について詳しくは、次のセクションを参照してください。

ランタイム構成ファイル

ランタイム構成ファイルは、標準ビューには表示されません。これは、IBM Security Directory Integrator サーバーがソリューションの実行に使用するファイルであり、古いバージョン (7.0 より前) では直接処理されていたファイルです。CE で構成ファイルを保管するたびに、ランタイム構成ファイルも更新されます。このファイルは、実行のために IBM Security Directory Integrator サーバーに転送されます。このファイルはプロジェクト・ビルダーで保守されます。エンド・ユーザーはこのファイルを変更しないでください。

ランタイム構成ファイルが変更された場合に、このファイルを自動的にエクスポートするようにプロジェクトを構成できます。プロジェクト・プロパティー・パネルを使用して、プロジェクトにリンクされているファイルを構成します。

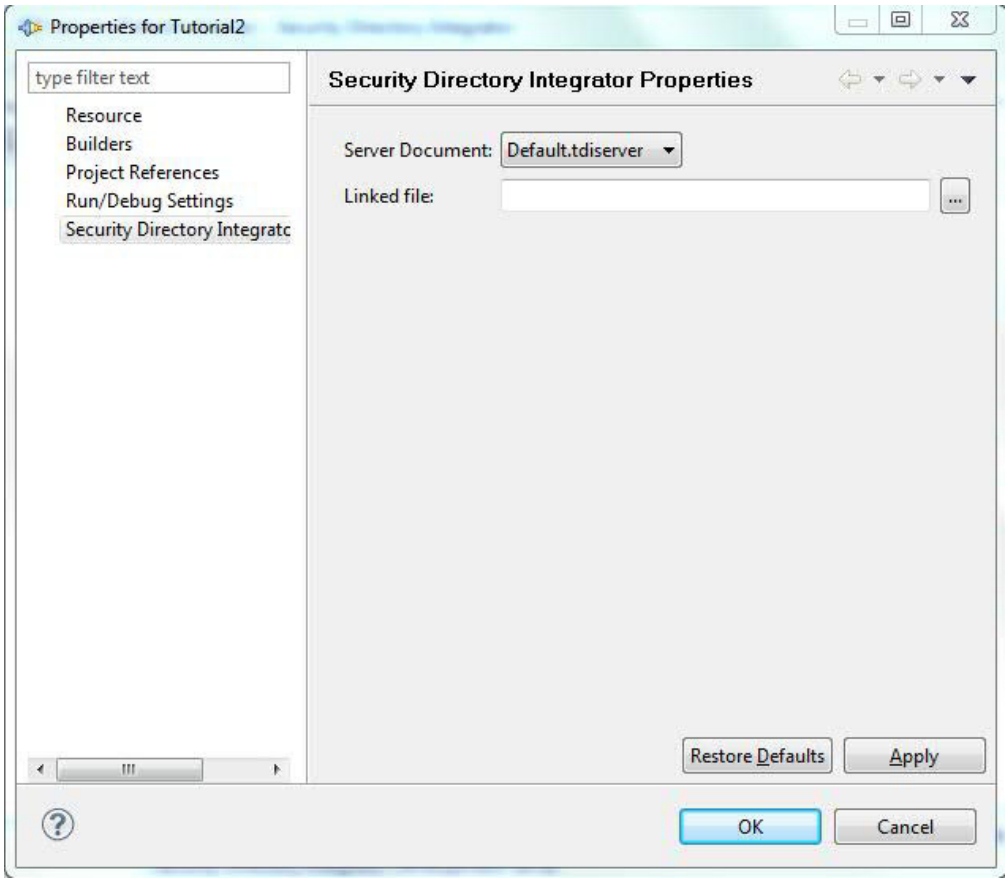


図 7. IBM Security Directory Integrator の「プロジェクトのプロパティー (Project Properties)」ウィンドウ

ランタイム構成が更新されるたびに、プロジェクトの IBM Security Directory Integrator プロパティー・セクションに指定されているファイルにも、更新された構

成がコピーされます。「**ファイル**」 > 「**Tivoli Directory Integrator 構成ファイルを開く...**」コマンドを使用すると、リンクされているファイルが自動的に設定されます。

プロジェクト・ビルダー

プロジェクトには、カスタム・プロジェクト・ビルダーが関連付けられています。プロジェクト・ビルダーの目的は、すべての成果物を組み合わせて*実行可能な構成ファイル* を作成することです。

このビルダーは、リソースが変更されるたびに自動的に実行するか、または標準の「**プロジェクト**」 > 「**ビルド**」メニュー項目を選択して手動で実行することができます。いずれの場合でも、IBM Security Directory Integrator プロジェクト・ビルダーでは、起動時に更新された構成ファイルが維持されます。リソースが更新 (変更、追加、または削除) されている場合、ビルダーではそのリソース更新内容を使用して実行可能構成ファイルが更新されます。認識される構成ファイルに対してのみ操作が実行されます。つまり、.gif ファイルには操作が実行されませんが、AssemblyLine には操作が実行されます。実行可能構成ファイルの名前はドット (.) で始まるため、通常は非表示になっています。`.rs.xml` という名前のファイルがプロジェクト・フォルダーにあります。これは、実行のために IBM Security Directory Integrator サーバーに送信されるコンパイル済み構成ファイルです。

ビルダーは、ターゲット構成のコンポーネントを再配置して名前を変更します。「リソース」フォルダーにプロパティーとコネクターの両方が含まれている場合、これらのコンポーネントはターゲット構成の標準フォルダー (「プロパティー」フォルダーと「コネクター」フォルダー) に再配置されます。

プロジェクト・ビルダーでは、変更されたすべてのコンポーネントに、明らかなエラーや潜在的な問題があるかどうかも検査されます。このような問題は Eclipse の標準「**問題**」ビューに表示されます。各問題項目には、問題の説明と問題の発生場所が示されます。このため、ダブルクリックしてエディターを起動すると、問題の検出場所がエディターに示されます。

プロパティーと置換

すべての IBM Security Directory Integrator プロジェクトは IBM Security Directory Integrator サーバーに関連付けられています。関連付けられているサーバーは、プロジェクトの AssemblyLine の実行に使用されるサーバーです。サーバーが異なるマシンにインストールされていることがあるため、プロジェクト・モデルでは、選択されたプロパティー・ストアのローカル・コピーを使用してプロパティーへのアクセスを提供する必要があります。

プロパティー・ストアは、必要に応じてサーバーからダウンロードできます。ローカル・コピーには、各プロパティーの値が 2 つずつ含まれています。これらは、サーバーからダウンロードした値 (リモート値) と、ユーザーによって設定された値 (ローカル値) です。これは、競合している可能性があるプロパティーを確認し、ソリューションにより使用されているプロパティーのセットを抽出できるようにすることを目的としています。プロパティーをプロパティー・ストアに追加する前に、プロパティー・ストアをダウンロードする必要はありません。ただし、ソリューシ

ョンの実行時に、既存のプロパティーが誤って上書きされることを防ぐため、ローカル値を持つプロパティーはすべて、サーバーの値と突き合わせてチェックされます。

プロパティー・ファイルを編集するには、「リソース」フォルダーのプロパティー・ファイルを開きます (またはこのフォルダーにプロパティー・ファイルを作成します)。プロパティー・ファイルを作成したら、その内容を変更し、アップロードとダウンロードを実行することができます。アップロードとダウンロードは、ローカルのプロパティーとサーバー上のプロパティーを同期するための操作です。

注: IBM Security Directory Integrator は現在、キー/値 (または値/キー) ペアのプロパティー・ファイルで等号「=」またはコロン「:」を分離文字として使用しています。このため、プロパティー名およびプロパティー値には、等号またはコロンを使用できません。IBM Security Directory Integrator V6.0 以前では、プロパティー・ファイルのキー/値の分離文字は「:」文字のみだったため、V6.0 以前からマイグレーションしたプロパティー・ファイルは編集が必要な場合があります。

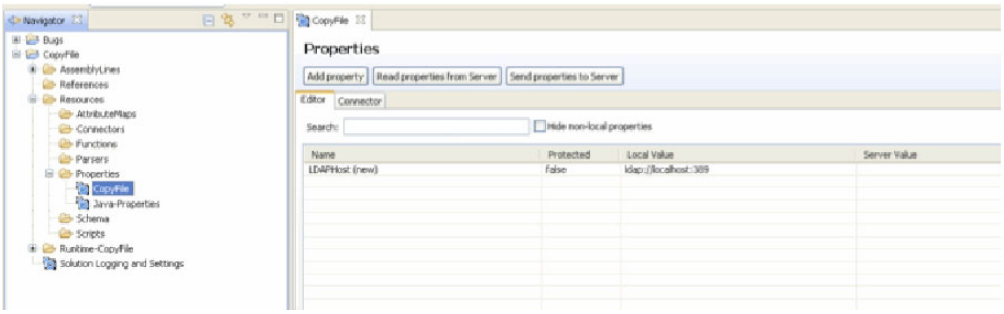


図 8. 「プロパティー」ビュー

相対パスが設定されているカスタム・プロパティー・ストア (例えばコネクター構成では相対パスが使用される) の場合、対応するファイルがランタイムの *Project* ディレクトリーに生成される点に注意してください。新しいカスタム・プロパティーを作成すると、デフォルトのパスは「{config.$directory}/*Filename*.properties」に設定されます。この *Filename* は、新規プロパティー・ストアに指定した名前です。「{config.$directory}」は、ランタイム・ファイルの位置に解決されます。

デフォルトでは、共用プロパティー・ストア (グローバル・ストア、ソリューション・ストア、およびシステム・ストアなど) はプロジェクトには追加されません。これらのファイルの変更を維持したい場合は、プロジェクトにこれらを追加できます。共用プロパティー・ストアを表示するには、「サーバー」ビューまたはメイン・ツールバーの「**サーバー・ストアのブラウズ**」を使用します。

ユーザー・インターフェース・モデル

ユーザー・インターフェース (UI) は、一連のビュー、エディター、およびその他の UI 関連リソースによって構成されます。これらのリソースは、Eclipse プラットフォームで定義されている標準拡張ポイント・メカニズムを介して提供されます。

CE には多数の拡張ポイント・コントリビューションがありますが、ここでは最も重要なコントリビューションについてのみ説明します。

表 7. *Eclipse CE 拡張ポイント・コントリビューション*

コントリビューション	説明
エディター	すべての主要構成ファイルを編集するためのエディターが提供されます。 • AssemblyLine (.assemblyline ファイル) • コネクター (.connector ファイル) • 関数 (.function ファイル) • スクリプト (.script ファイル) • プロパティー (.tdiproperties ファイル) • 属性マップ (.attributemap ファイル)
ビュー	構成ファイルのさまざまな側面を視覚化するための各種ビューが提供されます。
メニュー、ツールバー	構成オブジェクトに対して実行される CE のすべてのアクションは、標準アクションとして定義されます。
ウィザード	プロジェクトおよび構成ファイルの新規作成を支援する各種ウィザードが提供されます。

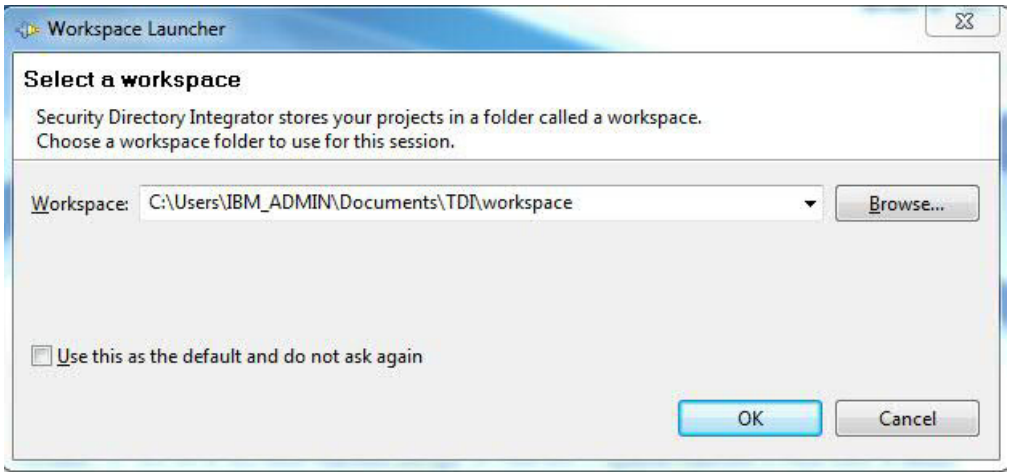
CE は MVC (モデル、ビュー、コントローラー) パラダイムに準拠しています。構成ファイルのエディターでは、構成ファイルの内容を表示するウィジェットが作成されます。このウィジェットでは、使用されるツールバーとメニューが Eclipse フレームワークに登録されるため、これらのツールバーやメニューのコントリビューションを作成できます。構成ファイルに提供するすべてのアクションのインプリメントとコントリビュートには、標準 Eclipse メカニズムが使用されます。

ユーザー・インターフェース

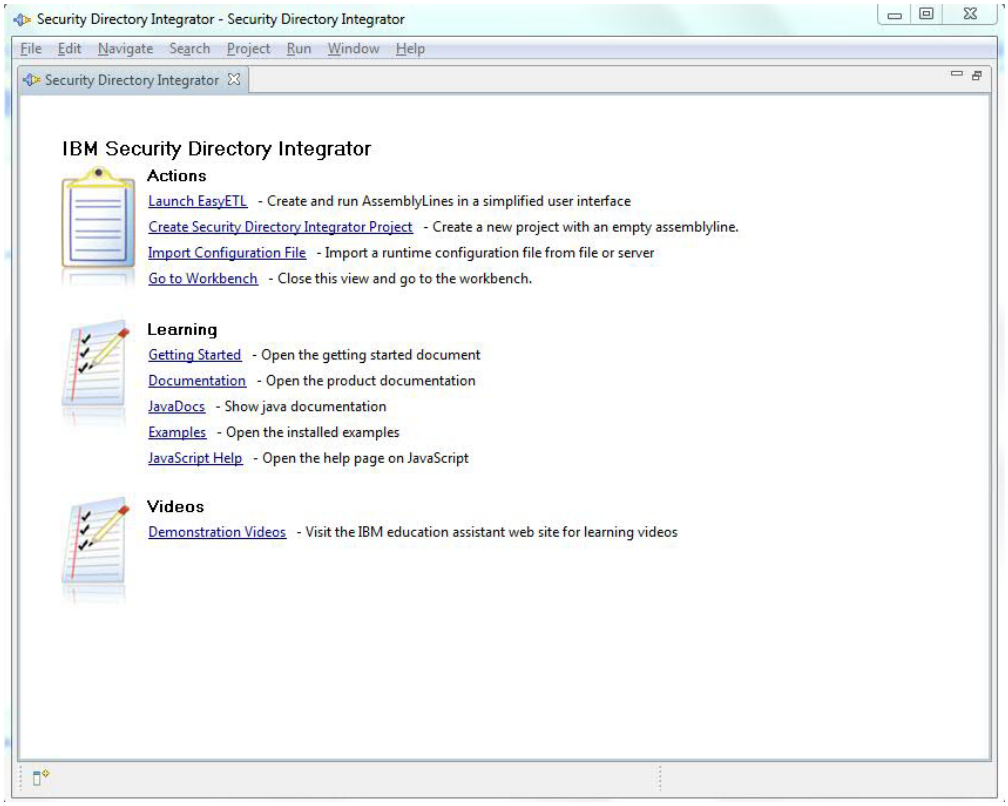
アプリケーション・ウィンドウ

構成エディター (CE) を初めて開始したときには、ワークスペース・ディレクトリーの入力を求めるプロンプトが出されます。ワークスペース・ディレクトリーとは、プロジェクトを保管するファイル・システムの場所です。このディレクトリーは後で、「**ファイル**」メニューから変更することができます。

このダイアログは、指定したワークスペース・ディレクトリーをデフォルトの場所にするチェック・ボックスを選択しない限り、CE を開始するたびに表示されます。

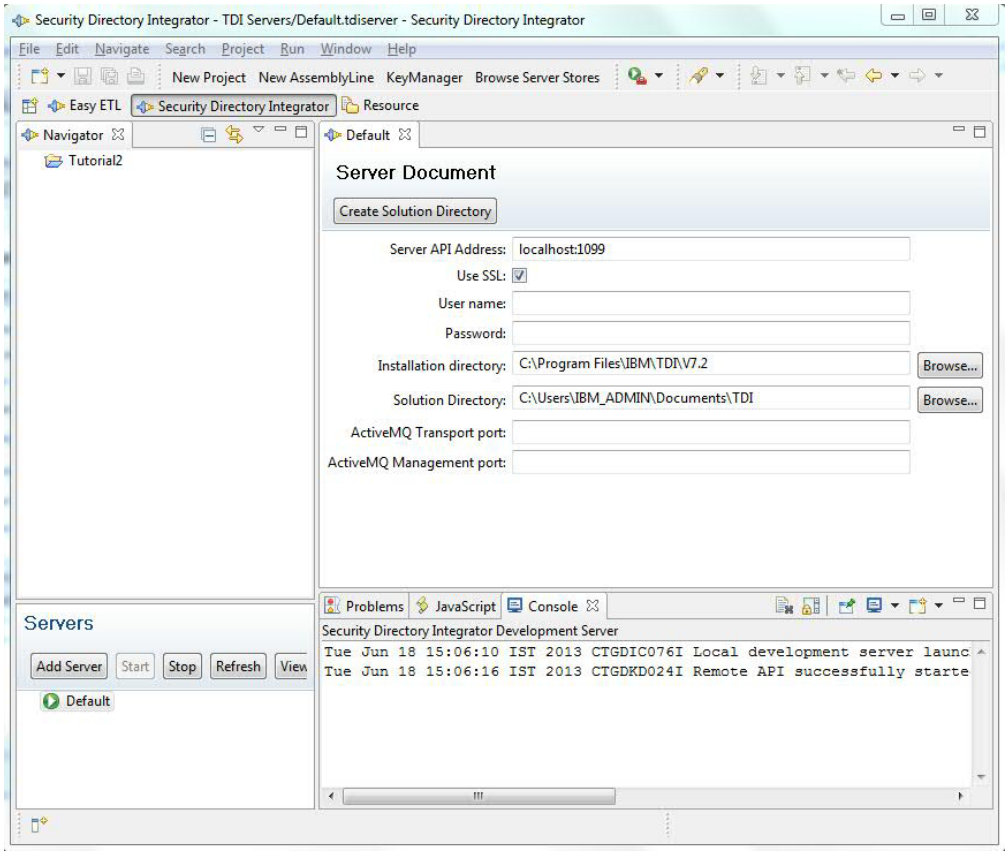


このダイアログの後、メイン・ワークスペース・ウィンドウが表示されます。初めて CE を開始する場合は、次のようなウェルカム画面が表示されます。



このウェルカム画面には、一般的なタスクや情報サイトへのいくつかのクイック・リンクがあります。「資料」リンクをクリックすると、構成済みの製品資料 (システム・プロパティー *com.ibm.tdi.helpLoc*) を開くことができます。

ウェルカム画面を後でもう一度開くには、「**ヘルプ**」 > 「**ようこそ**」を選択します。ウェルカム画面を閉じると、次のようなワークスペース・ウィンドウが表示されます。



これは、プロジェクトや構成を管理するメイン・ウィンドウです。

この図には、開いている 1 つのエディターがあり (Default.server という文書のエディター)、いくつかのビューがあります。この図の中にあるビューが最も重要なビューです。

「ナビゲーター」(左上) には、サーバー構成および IBM Security Directory Integrator ソリューション用のすべてのプロジェクトとソース・ファイルが表示されます。「ナビゲーター」には、テキスト・ファイルなどのその他のファイルやプロジェクトが表示されることもあります。CE は、IBM Security Directory Integrator プロジェクトを特異的に取り扱うため、他のファイルやプロジェクトは CE の影響を受けません。その仕組みについては、プロジェクト・ビルダーについてのセクションに説明があります。

「サーバー」ビュー (左下) には、「IBM Security Directory Integrator サーバー」プロジェクトに定義されている各サーバーの状況が表示されます。ユーザー定義した任意の数のサーバーが表示されます。このビューには、サーバー上で操作するさまざまな機能とその構成が表示されます。「リフレッシュ」ボタンを選択すると、ビュー内のすべてのサーバーの状況が更新されます。

エディター・エリア (右上) は、すべてのエディターが表示される場所です。AssemblyLine 構成などの文書を開くと、このエリアに表示されます。このエリアは上下に分かれており、右下のエリアにはその他の関連情報を提供する各種のビューがあります。その中で最も重要なのが、「問題」ビュー (IBM Security Directory Integrator コンポーネントでの潜在的な問題を示します)、「エラー・ログ」(ソリュ

ーションの開発中に発生したエラーを示します)、そして「コンソール」ビュー (IBM Security Directory Integrator サーバー (CE によって始動されたものなど) の実行のためのコンソール・ログを示します) の各ビューです。

「サーバー」ビュー

「サーバー」ビューには、サーバー・インスタンスの管理に役立つ多数の機能が組み込まれています。サーバー・インスタンスの追加と除去以外にも、サーバー、サーバーのアクティブ構成インスタンス、および AssemblyLine に関連する多数のコマンドがあります。

図 9. 構成エディターの「サーバー」ビュー

メイン・ツールバーには次のボタンがあります。

サーバーの追加
別のサーバーを追加するときに使用します。

開始 「ローカル」サーバー (ファイル・システムでアクセス可能なサーバーなど) を始動するときに使用します。

選択した構成インスタンスがある場合は、その AssemblyLine のうちの 1 つ以上を開始できます。

停止 サーバー、構成インスタンス、または AssemblyLine に停止要求を送信するときに使用します。

リフレッシュ
「サーバー」ビューの内容を最新表示するときに使用します。

ログの表示
「ローカル」サーバーの標準ログ・ファイル (「ibmdi.log」ファイル) を表示するときに使用します。

ビューの各項目のポップアップ・メニューには、選択されている項目に基づいて実行可能なその他のコマンドが表示されます。

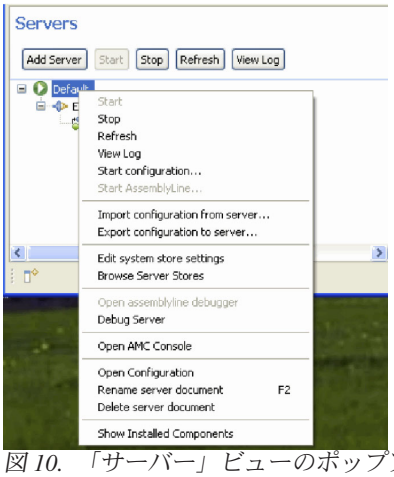


図 10. 「サーバー」ビューのポップアップ・メニュー

表示されるコマンドには次のものがあります。

開始　サーバーを始動します。

停止　サーバー、構成インスタンス、または AssemblyLine を停止します。

リフレッシュ
「サーバー」ビューを最新表示します。

ログの表示
ibmdi.log ファイルをテキスト・エディターで開きます。

構成の開始...
サーバーが選択されている場合、そのサーバーにある構成インスタンスを開始できます。

AssemblyLine の開始....
構成インスタンスが選択されている場合、その構成インスタンスから AssemblyLine を開始できます。

サーバーからの構成のインポート...
サーバーから CE プロジェクトに構成をインポートします。

サーバーへの構成のエクスポート...
選択したサーバーのランタイム構成に CE プロジェクトをエクスポートします。

システム・ストア設定の編集
選択されているサーバーのシステム・ストアの設定を表示します。

サーバー・ストアのブラウズ
サーバー・ストア・テーブルの表示/編集のためにサーバー・ストア・データ・ブラウザーを開きます。

AssemblyLine デバッガーのオープン
選択されている AssemblyLine にデバッガーを接続します。これにより、既に実行中の AssemblyLine をデバッグできます。

サーバーのデバッグ
選択されているサーバーのサーバー・デバッグ・セッションを開きます。

インストール済みコンポーネントの表示
選択されているサーバーにインストールされているコンポーネントとそのバージョンのリストを表示します。

AMC コンソールのオープン
選択されているサーバーの AMC コンソールを開きます。インストールされているサーバーに AMC がない場合、このオプションはぼかし表示されます。

注: AMC 機能は推奨されません。IBM Security Directory Integrator の将来のバージョンでは除去されます。

サーバー文書の削除
「サーバー」ビューからサーバーを削除します。

サーバー文書の名前変更
選択されているサーバーの名前を変更します。この操作はサーバー自体には影響しません。サーバーのローカル表現のみが変更されます。

メニュー・オプションは、選択されているコンポーネントに基づいて使用可能または使用不可になります。

式エディター

式エディターは、さまざまなコンテキストで使用できます。一般には、接続パラメーター、リンク基準などのパラメーター値を指定する際に、単純な値をパラメーターに入力する代わりに式エディターを使用します。

プロパティーの使用

このオプションを指定すると、プロパティー・ストアから既存のプロパティーを選択するか、プロパティーと値のペアを新しく作成することができます。

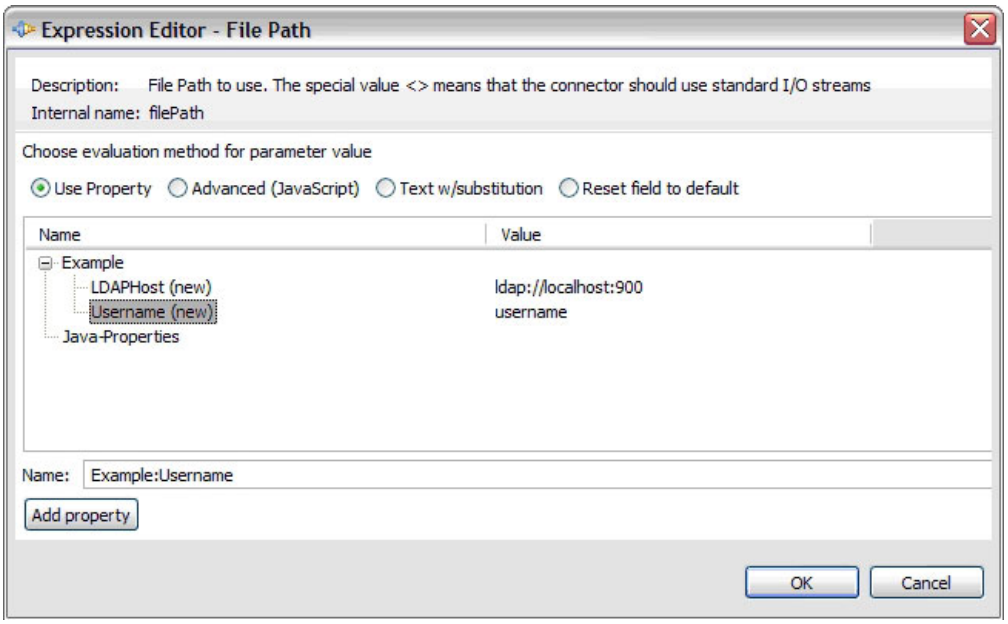


図 11. 式エディター: シンプル・プロパティー

プロパティーを選択する場合、ツリーの下の「*名前*」テキスト・フィールドが、そのプロパティーに対する式で更新されます。デフォルトでは、式は、ストア名 (ここでは「Example」) の後にコロンが続き、その後にプロパティー名 (ここでは「Username」) が続くという形になります。「**OK**」をクリックすると、テキスト・フィールド内の式がパラメーターに使用されます。つまり、プロパティーのツリーを介さずに、このフィールドに直接式を入力することもできます。例えば、このソリューションが実行されるサーバー上に「FilePath」というプロパティーがあるとわかっている場合は、ストア名がわからなくてもそれを除去できます (つまり、接頭部「store:」を付けずに「FilePath」を入力します)。

拡張 (JavaScript)

このオプションを指定すると、値は、エディターで入力する JavaScript コードの結果として計算されます。

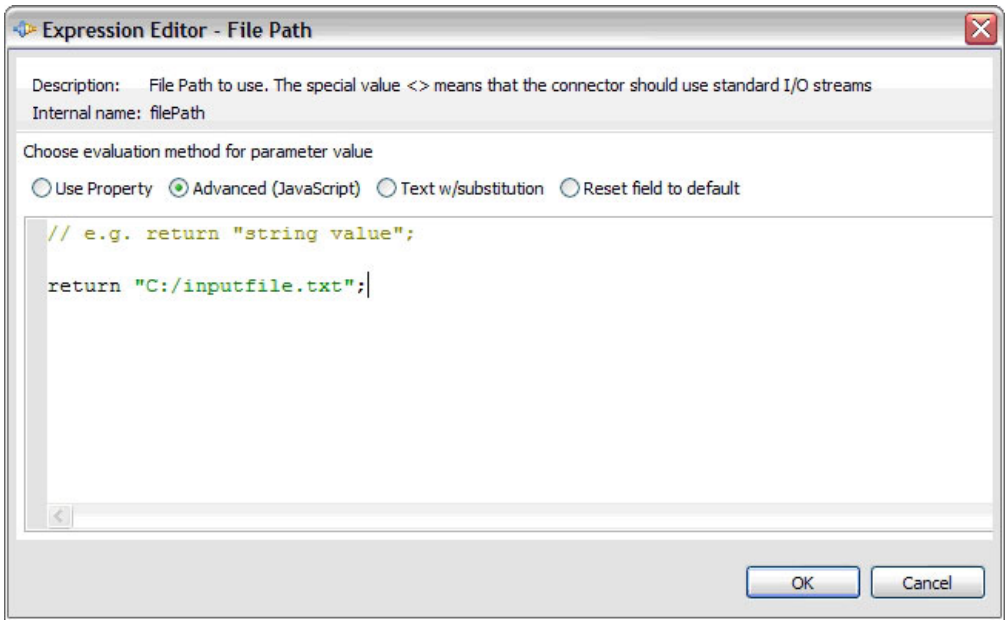


図 12. 式エディター: 拡張 (JavaScript)

テキストの置換

これは、バージョン 6.x からの「IBM Security Directory Integrator 式」です。バージョン 7 以降は、変数とプロパティーの複合評価には JavaScript を使用することをお勧めします。ただし、大量のテキストを入力する場合には、このオプションが役に立ちます。置換オプションは、バージョン 6 の式と互換性があります。

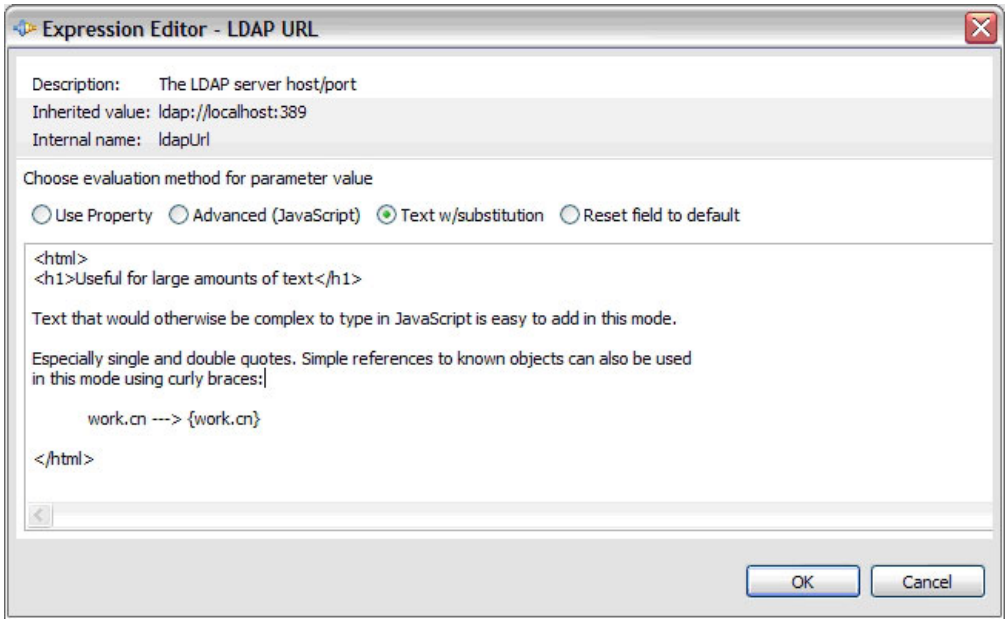


図 13. 式エディター: v.6 スタイルのテキストの置換

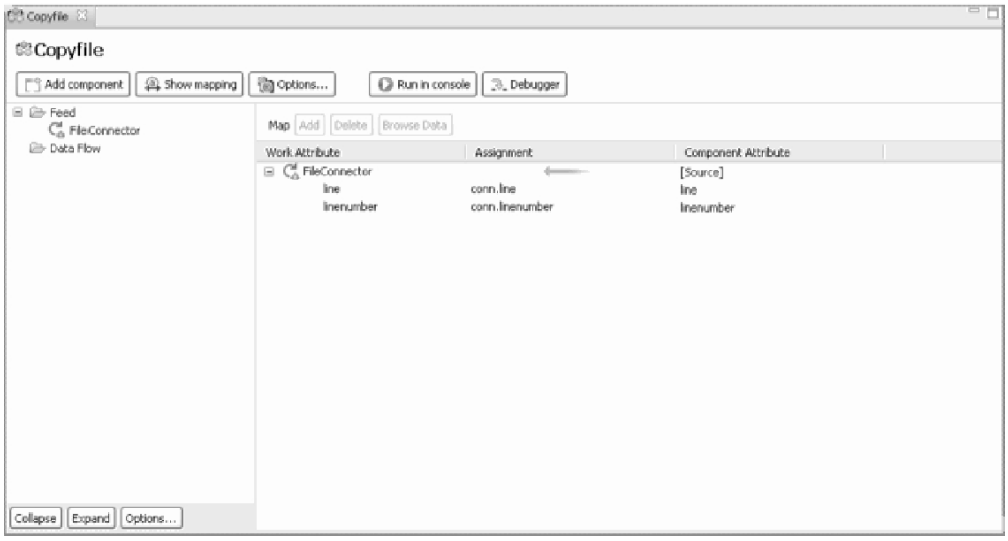
フィールドをデフォルトにリセット

この最後のオプションは、パラメーター値をデフォルト値 (継承値ともいう) にリセットする場合に使用します。このオプションを選択して「**OK**」を押すと、パラメーター値がリセットされます。

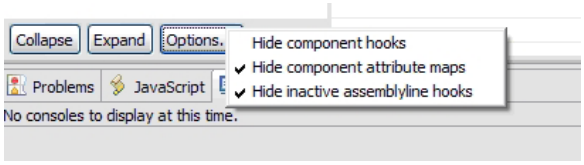
AssemblyLine エディター

AssemblyLine エディターは、IBM Security Directory Integrator ソリューションの開発時に使用される基本エディターです。

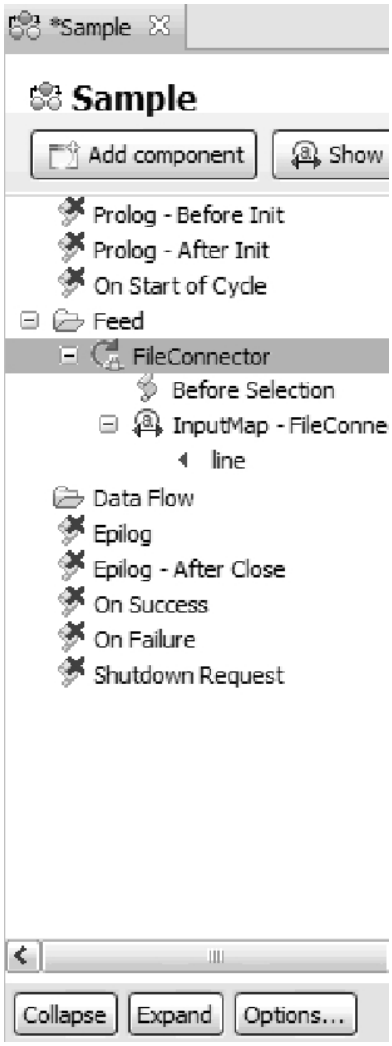
このエディターでは、コンポーネントを追加および構成することにより、AssemblyLine を構築します。作業を進めながら、AssemblyLine を実行して、追加したコンポーネントの効果を確認することができます。



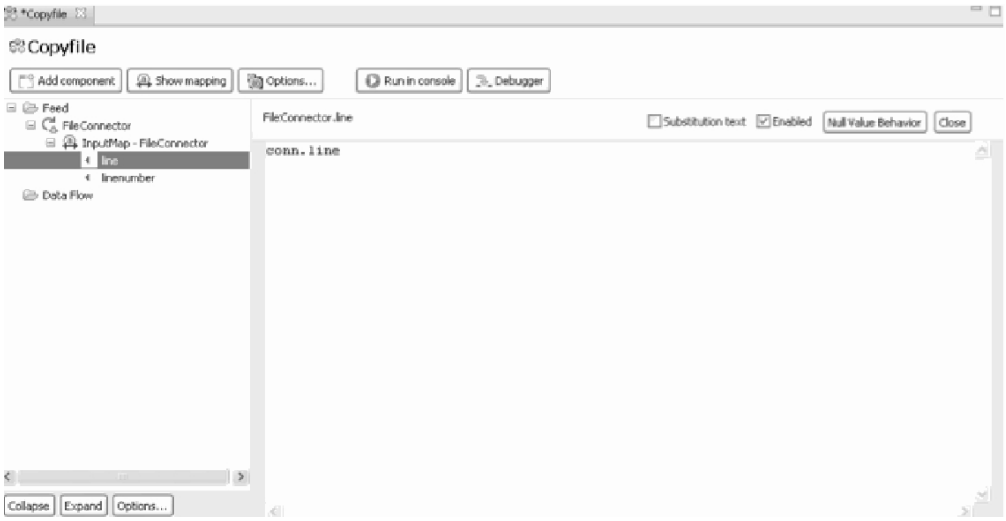
AssemblyLine エディターには、2 つのメイン・セクションがあります。左側のセクションには、AssemblyLine のコンポーネントとフックが表示されます。ツールバーで、ツリーに表示させる詳細のレベルを選択できます。



「**オプション...**」ボタンでは、コンポーネントのツリー・ビューに AssemblyLine のどの部分を表示させるかを選択できます。

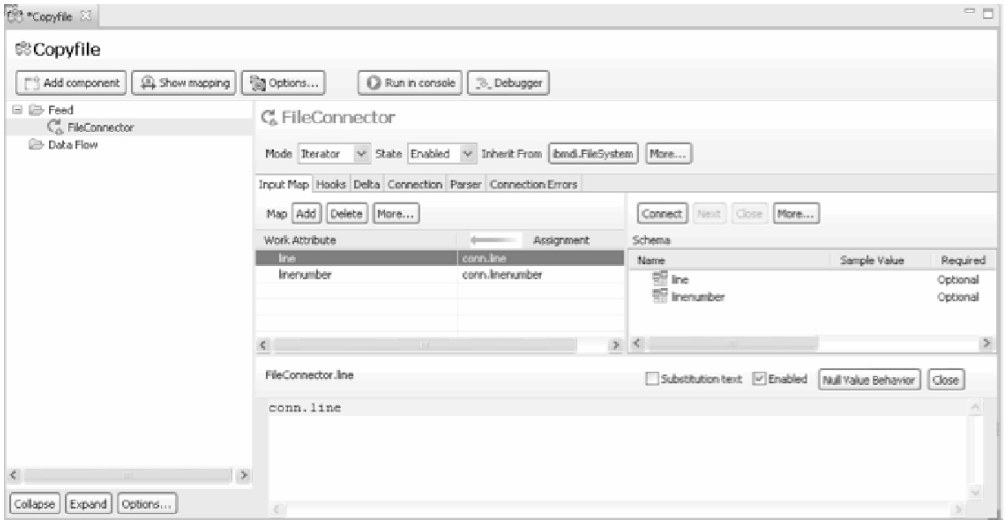


この図では、すべての AssemblyLine フックおよびコンポーネントの属性マップをツリー内に示しています。このツリー内でフックまたは属性マップ項目を選択すると、右側に、その項目の次のようなビュー (コンポーネント・エディター内より大きいサイズのビュー) が表示されます。



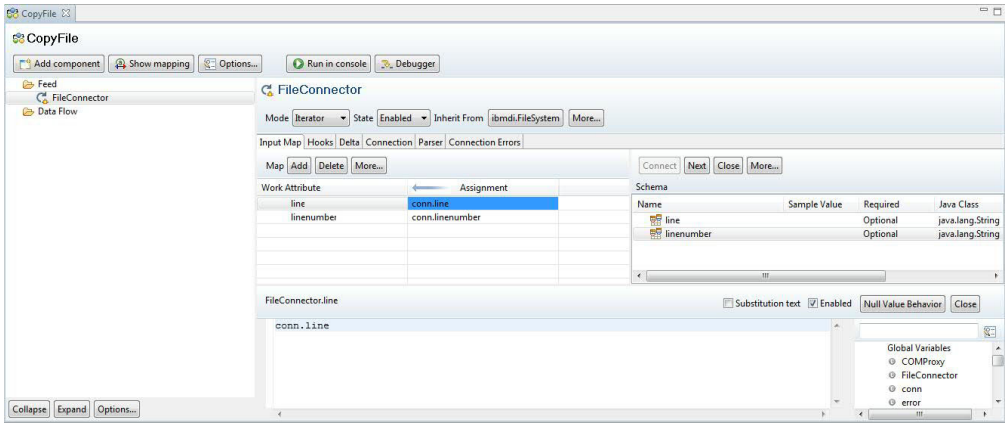
右側のマッピング・セクションには、属性マップを持つすべてのコンポーネントのマップが表示されます。このツリーの第 1 レベルにはコンポーネント名があり、個々の属性マップ項目がその下に表示されます。このような項目を選択すると、その項目の詳細エディターが表示されます。AssemblyLine のコンポーネント・ビューにフックを表示した場合は、同様にフックをダブルクリックして、スクリプト・エディターを表示させることができます。

次の図は、クイック・エディターが属性マップのスクリプトとともに表示される様子を示しています。

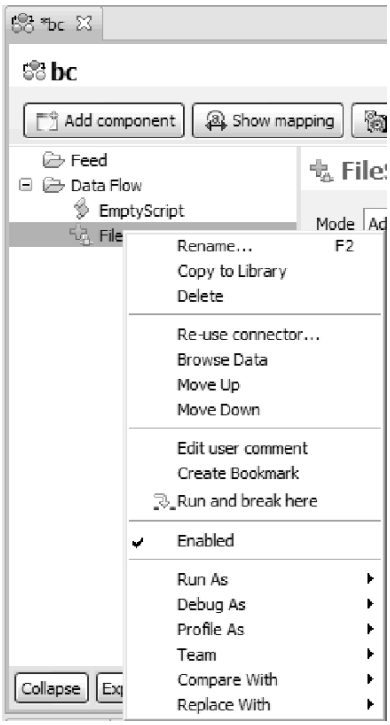


クイック・エディターには、そのほかのいくつかのオプションも表示されます。「置換テキスト」は、テキストや単純な拡張マクロを入力できる バージョン 6 の「IBM Security Directory Integrator 式」フォーマットです。バージョン 7 からは、JavaScript が提供する式構文がより強力になったため、このフォーマットは主に大きなテキスト値または複合テキスト値用となっています。

コンポーネント・フロー・セクションには、AssemblyLine のすべてのコンポーネントが表示されます。 1 つのコンポーネントを選択すると、エディターの右側がそのコンポーネントの構成画面に切り替わります。CE の便利なショートカットの 1 つに Ctrl+M のショートカットがあります。これを使用すると、現在のエディターまたはビューがアプリケーション画面いっぱいまで最大化されます。Ctrl+M では、最大化と通常サイズの切り替えを行うことができます。



コンポーネントをクリックすると、全面に表示した属性マッピング・ビューが、そのコンポーネントの構成画面に切り替わります。コンポーネントを右クリックして、そのコンポーネント用のポップアップ・メニューにアクセスすることもできます。



AssemblyLine のオプション

「**設定**」ドロップダウン・ボタンから、いくつかのオプションを選択することができます。

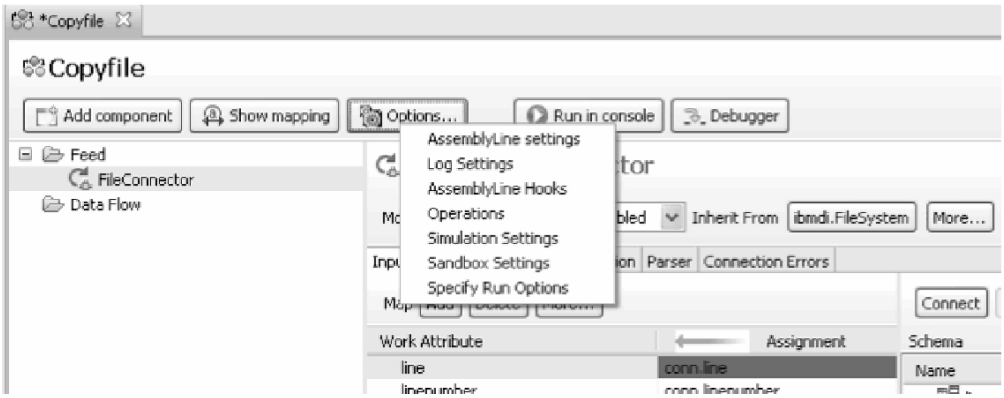


図 14. AssemblyLine のオプション・メニュー

このドロップダウン・メニューから、設計とランタイム・オプションの両方に関する、AssemblyLine のさまざまな側面を管理するいくつかのオプション画面が提供されます。次の画面があります。

- 104 ページの『AssemblyLine 設定』
- 105 ページの『ログ設定』

- 106 ページの『AssemblyLine フック』
- 107 ページの『AssemblyLine 操作』
- 108 ページの『シミュレーション設定』
- 109 ページの『Sandbox 設定』

AssemblyLine 設定

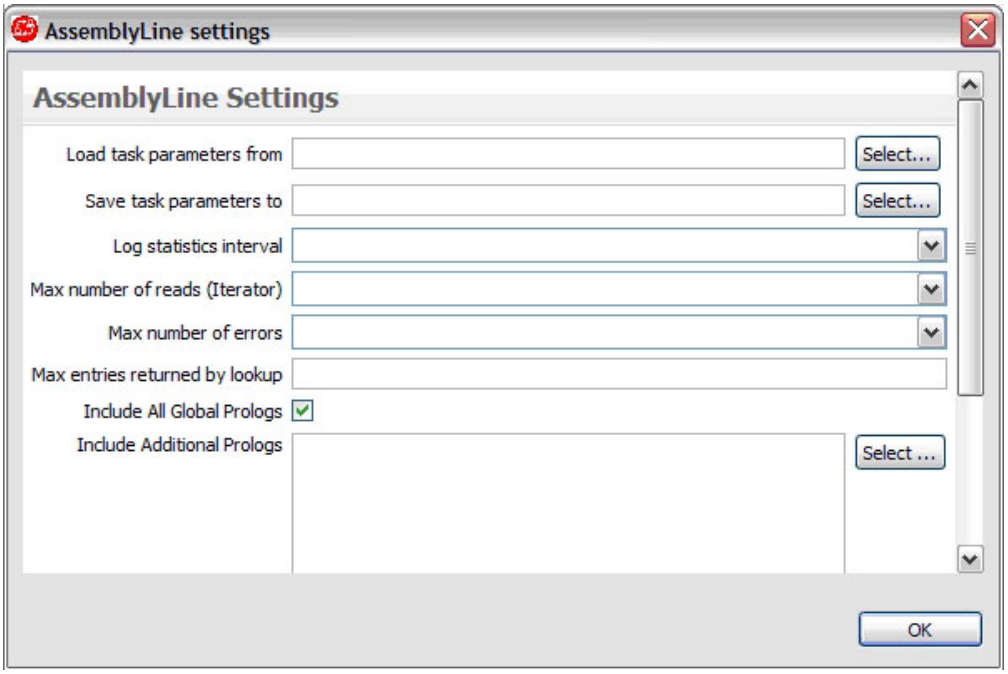


図 15. AssemblyLine 設定

このウィンドウでは、AssemblyLine の実行方法に影響を与えるオプションを指定することができます。

ログ設定

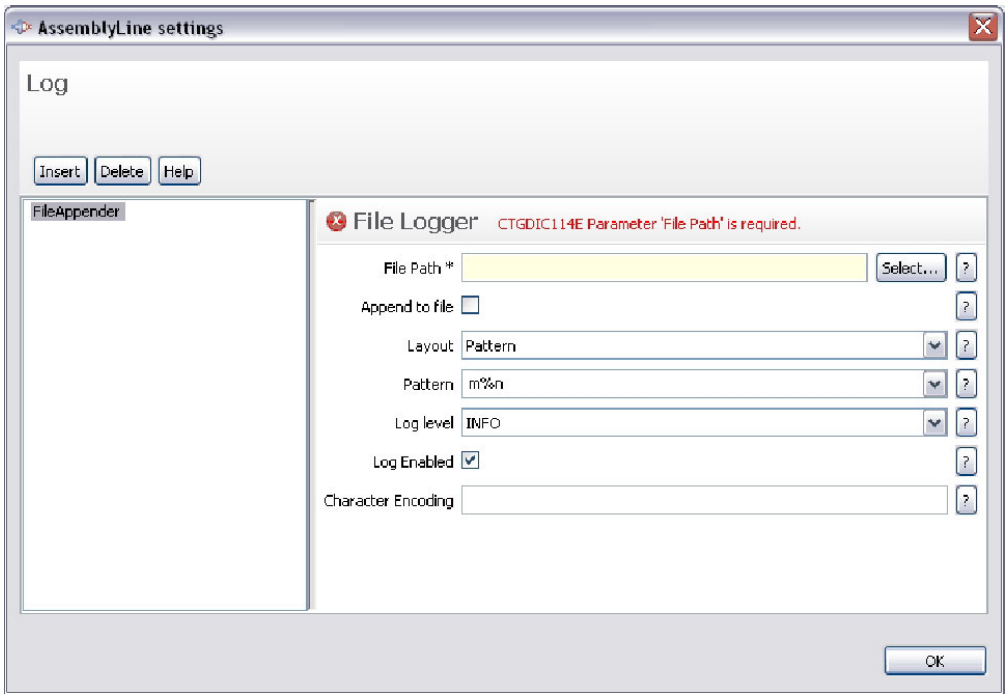


図 16. AssemblyLine のログ設定

このウィンドウでは、この AssemblyLine のロガーを追加できます。追加するロガーは、グローバルではなく、AssemblyLine のみを対象としてアクティブになります。

AssemblyLine フック

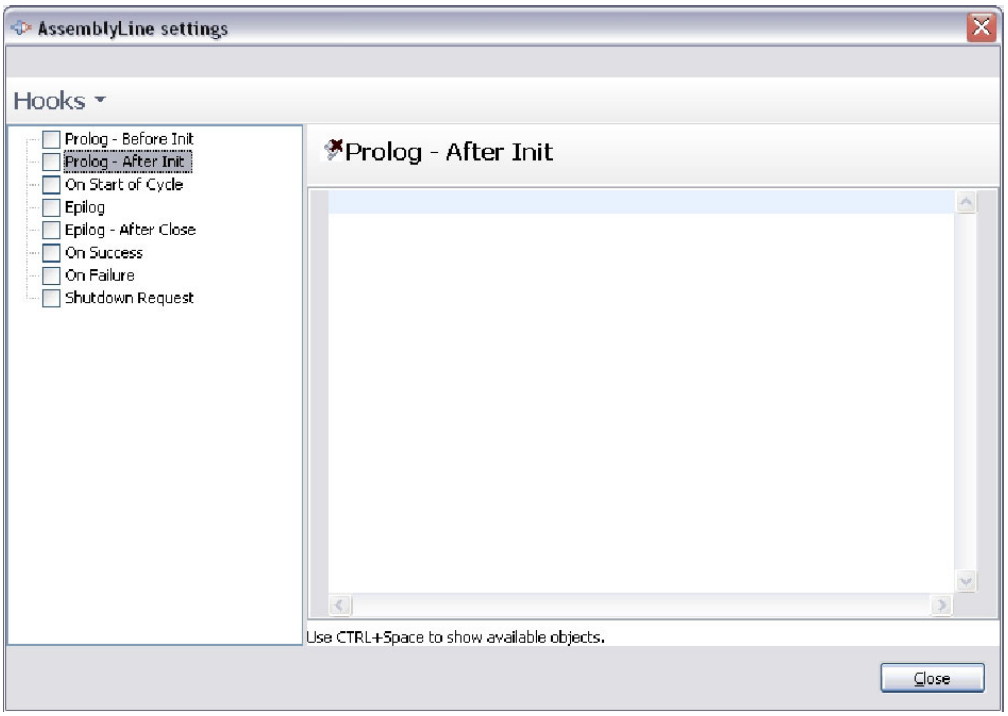


図 17. AssemblyLine フック

このウィンドウでは、 AssemblyLine レベルのフックを使用可能または使用不可にすることができます。使用可能にしたフックは、AssemblyLine エディターのコンポーネント・パネルにも表示されます。

AssemblyLine 操作

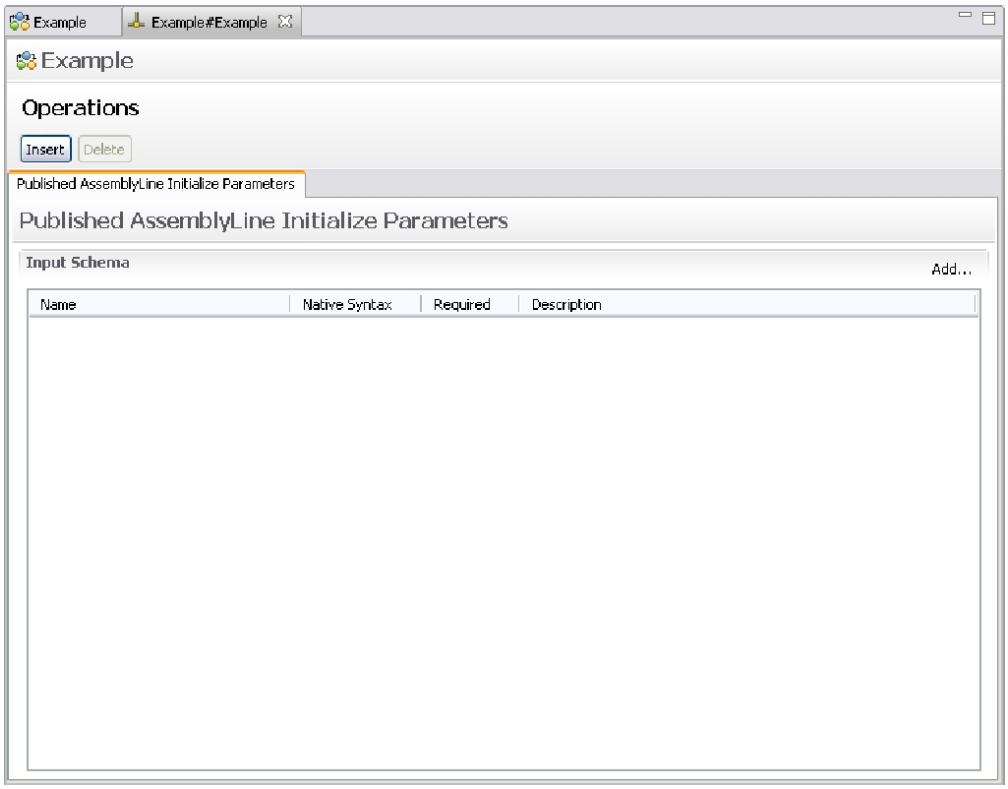


図 18. AssemblyLine 操作

このウィンドウでは、AssemblyLine の操作を定義できます。操作の詳細な説明については、「リファレンス」の『アダプターを使用した新規コンポーネントの作成』の『AL の操作』のセクションを参照してください。「挿入」ボタンを使用すると、新しい操作を追加することができます。デフォルトで使用可能になる「**パブリッシュ済み AssemblyLine の初期化パラメーター**」は、コンポーネントが初期化される前に、AssemblyLine に値を提供するために使用される特殊な操作です。

シミュレーション設定

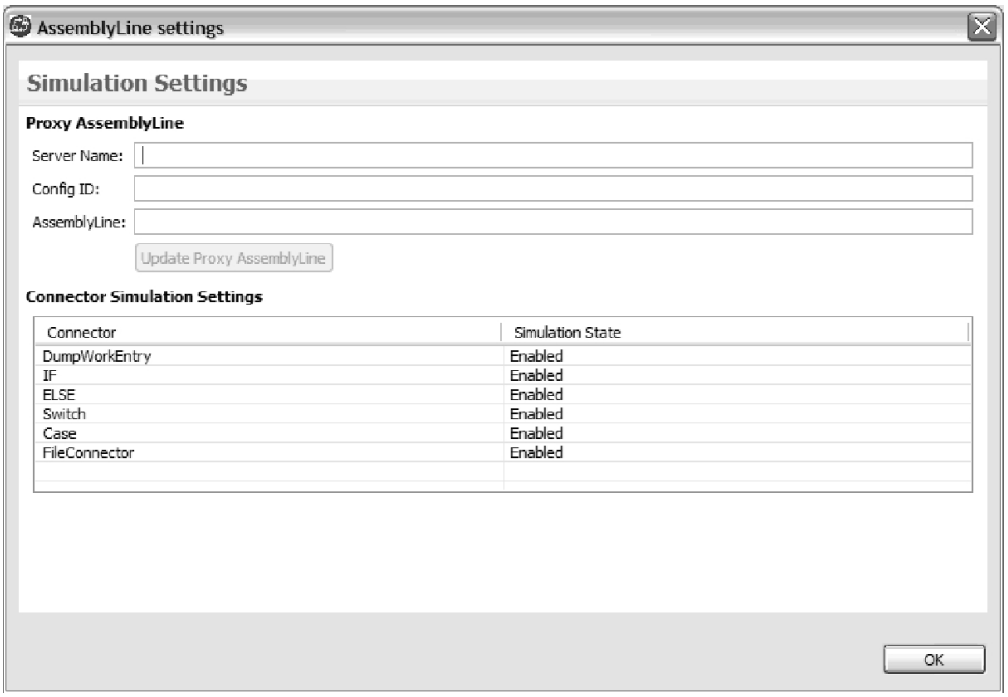
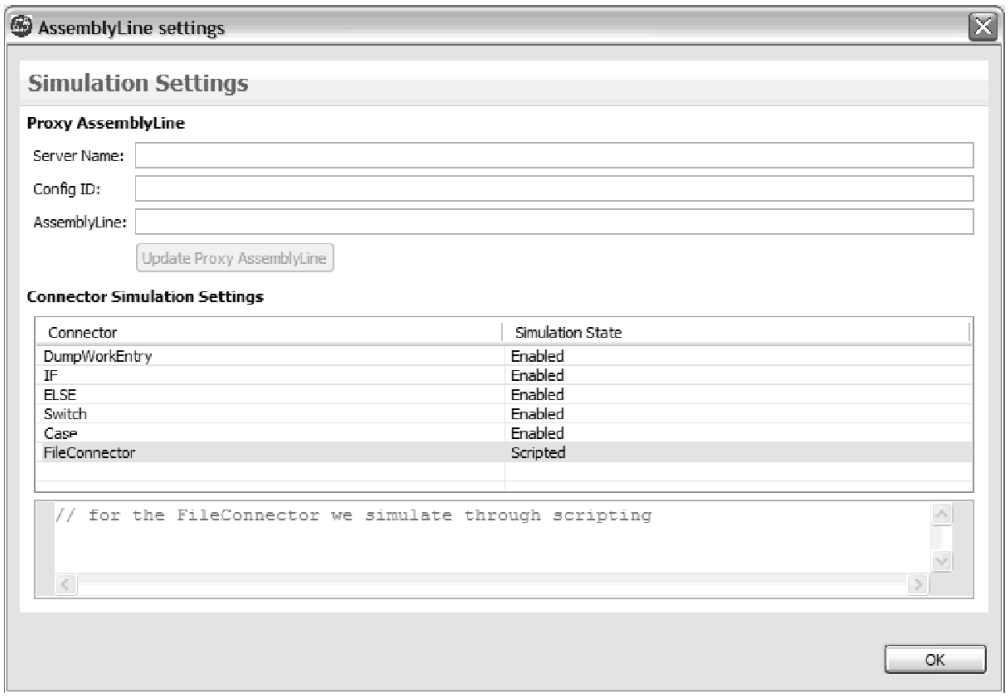


図 19. AssemblyLine のシミュレーション設定

このウィンドウでは、コンポーネントごとのシミュレーション設定を構成できます。詳しくは、 199 ページの『AssemblyLine シミュレーション・モード』を参照してください。シミュレーションのスクリプト記述を選択した場合、このパネルの下部にスクリプト・エディターが表示されます。



「**プロキシー AssemblyLine の更新**」ボタンを使用して、シミュレーション・コードで使用されるプロキシー AssemblyLine を作成または更新することもできます。

図 20. スクリプト・エディターを表示した AssemblyLine の「シミュレーション設定」ウィンドウ

Sandbox 設定

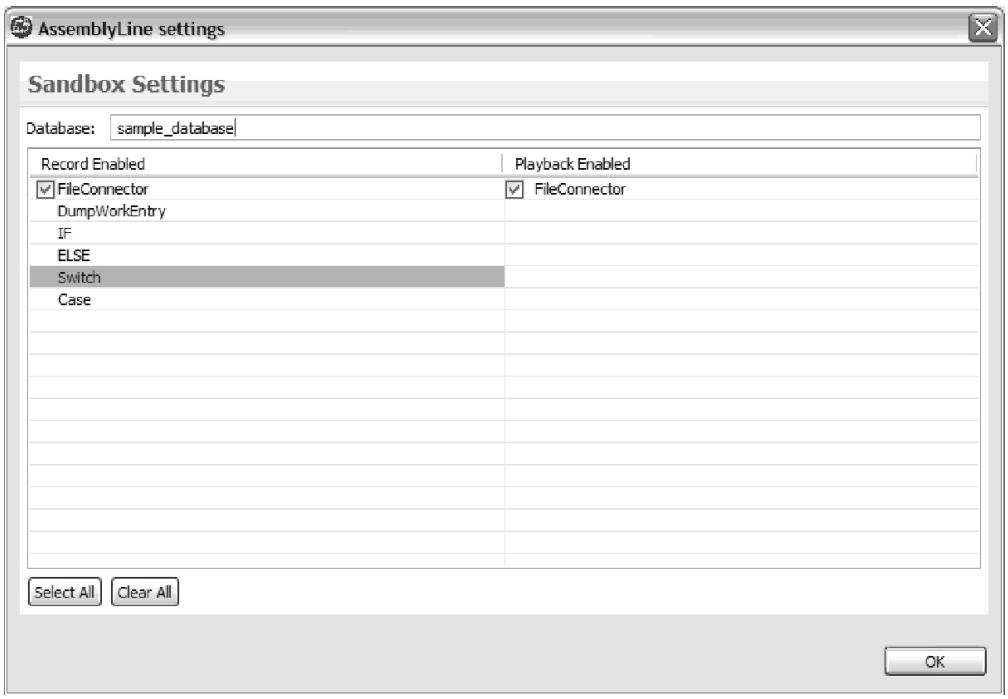


図 21. AssemblyLine の Sandbox 設定

「Sandbox 設定」では、AssemblyLine を記録モードまたはプレイバック・モードで実行するときに、どのコンポーネントを記録使用可能またはプレイバック使用可能にするかを構成することができます。

IBM Security Directory Integrator の Sandbox 機能について詳しくは、197 ページの『Sandbox』を参照してください。

実行オプションの指定

実行オプション・ダイアログで、AssemblyLine の実行方法を構成できます。

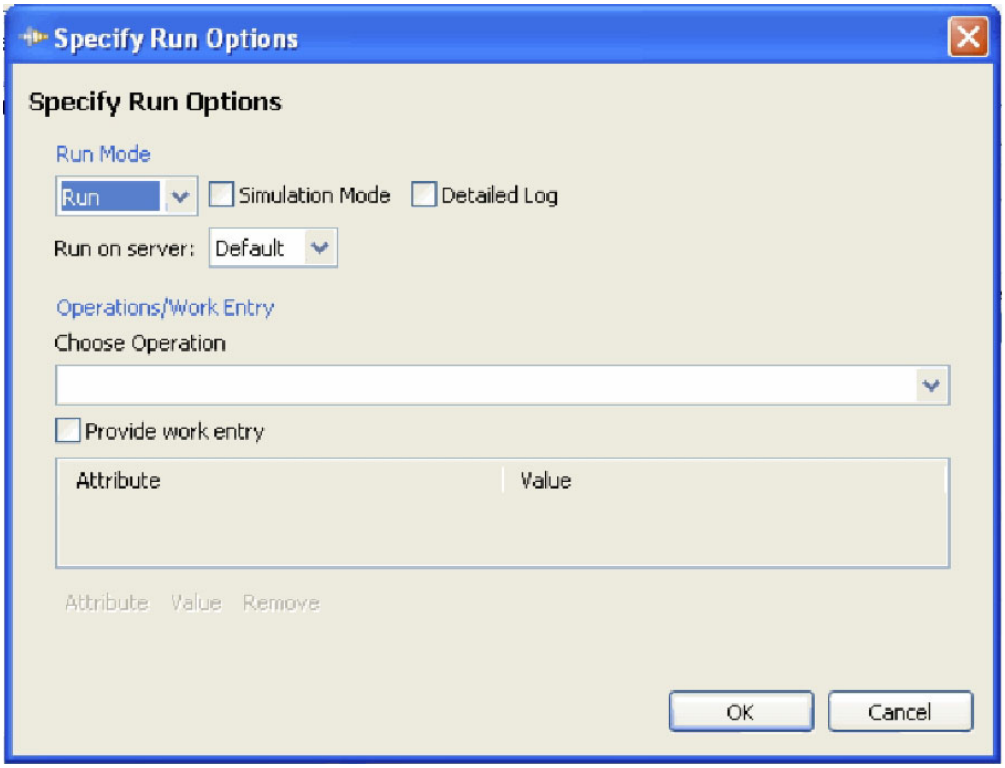


図 22. 「実行オプションの指定」ダイアログ

「**作業項目を提供します**」を選択すると、静的な項目を作成して AssemblyLine の開始時に送ることができます。

コンポーネント・パネル

AssemblyLine でコンポーネントを開くと、「AssemblyLine」ウィンドウの下部にクイック・エディター・パネルが表示されます。

該当するコンポーネントを以下に示します。

- 111 ページの『IF/ELSE/ELSE-IF ブランチ』
- 111 ページの『Switch/Case ブランチ』
- 113 ページの『FOR-EACH 属性値』
- 113 ページの『条件付きループ』
- 114 ページの『コネクター・ループ』
- 115 ページの『属性マップ』

IF/ELSE/ELSE-IF ブランチ

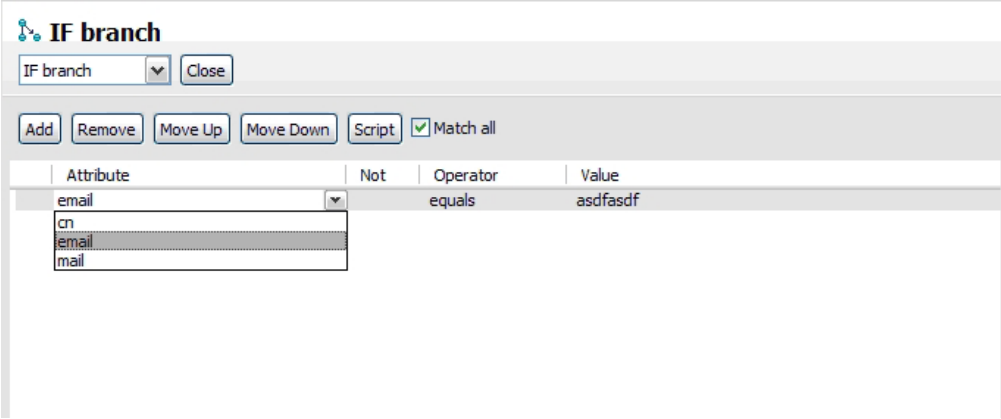


図 23. IF/ELSE/ELSE-IF ブランチのクイック・エディター

IF および ELSE-IF ブランチでは、単純式と、*true*または *false* を戻すカスタム・スクリプトを指定できます。すべての条件の評価結果が *true* になる場合 (*AND* モード) にのみ true を戻すようにするには、「**すべてに一致**」チェック・ボックスを使用します。このチェック・ボックスのチェック・マークが外れている場合、一致する必要がある条件は 1 つのみになります (*OR* モード)。

ELSE ブランチにはパラメーターはありません。

属性/演算子/値コントロールの新規行を追加するには、「**追加**」ボタンを使用します。これらの行を除去するには、「**除去**」ボタンを使用します。値には定数または式を指定できます。値テキスト・フィールドの後にあるボタンをクリックし、式エディターを使用して式を構成します。式の順序を変更するには、移動ボタンを使用します。式は上から下の順に評価されます。タイトルの下のドロップダウンを使用して、ブランチ・タイプ (IF、ELSE など) を変更することもできます。

Switch/Case ブランチ

Switch 構成には、値を選択するためのオプションが多数あります。この値は、含まれている Case ブランチの値と突き合わせられます。一致する場合に Case ブランチが実行されます。Switch ブランチは常に実行されます。

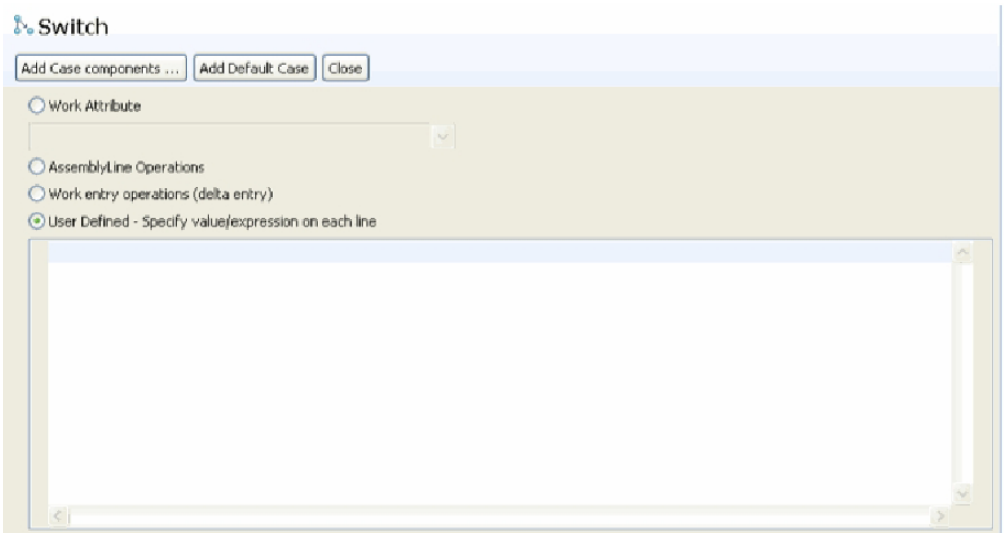


図 24. Switch/Case ブランチ

Switch/Case ブランチで選択できるオプションを次に示します。

作業属性
既知の作業属性のリストから選択します。

AssemblyLine 操作
既知の AssemblyLine 操作名のリストから選択します。

作業項目操作
作業項目の操作値 (`work.getOperation()` メソッドなど) を使用します。

ユーザー定義
ユーザー定義値を指定できます。

この Switch ブランチの中に Case ブランチを生成するには、「**Case コンポーネントの追加...**」ダイアログを使用します。選択内容に基づいて、適切な値が自動的に提示されます。作業項目操作を選択すると、既知の操作値 (追加、変更など) がすべて提示されます。

デフォルトの Case を追加するには、「**デフォルトの Case の追加**」ボタンを使用します。デフォルトの Case が実行されるのは、Switch コンポーネントの値に一致する Case ブランチがない場合です。

FOR-EACH 属性値

図 25. 属性値ループ

このコンポーネントは、属性の値をループします。ループする作業項目属性名 (作業属性名) と、ループ属性名を指定します。ループ属性名には、作業項目属性の現行のループ値が設定されます (例: `foreach f in work.attr.values; set loopattr = f`)。

条件付きループ

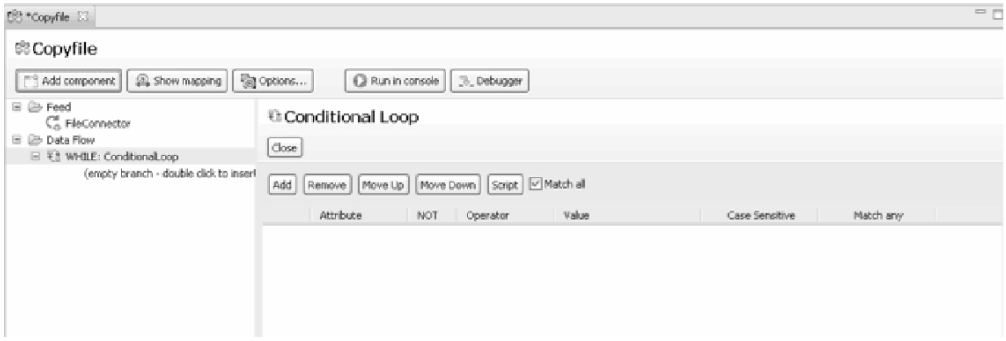


図 26. 条件付きループ

条件付きループでは、単純式と、true/false を戻すカスタム・スクリプトを指定できます。

属性/演算子/値コントロールの新規行を追加するには、「**追加**」ボタンを使用します。これらの行を除去するには、「除去」ボタンを使用します。値には定数または式を指定できます。値テキスト・フィールドの後にあるボタンをクリックし、式エディターを使用して式を構成します。

「**すべてに一致**」チェック・ボックスの設定により、ブランチを実行する条件として、すべての行が一致する必要があるか (「**すべてに一致**」にチェック・マークが付いている場合)、または 1 つの行のみが true である必要があるかどうかが決定します。

コネクター・ループ

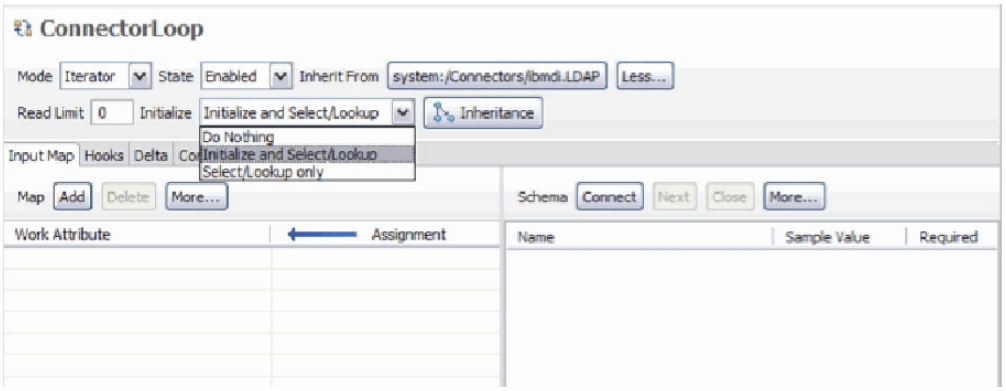


図 27. コネクター・ループ

コネクター・ループはコネクター・エディターを使用してコネクターを構成します。上記の図と異なる点がいくつかあります。初期化オプションには、通常の初期化オプションの代わりにコネクター・ループに関連するオプションが表示されます。また、「モード」ドロップダウンから選択できるのは「イテレーター」と「ルックアップ」のみです。

コネクターの場合に表示される標準のタブ以外に、出力属性マップも表示されます。このマップの「**コネクター・パラメーター**」タブで、コネクター・パラメーターの動的割り当てを構成できます。標準の出力マップとは多少異なり、この出力マップには、固定スキーマ (選択されているコネクターのパラメーターのリスト) が表示されます。スキーマ部分には、スキーマを対象とした「接続」ボタンと「次へ」ボタンは表示されません。

図 28. 「コネクター・ループ」のコネクター・パラメーター

属性マップ

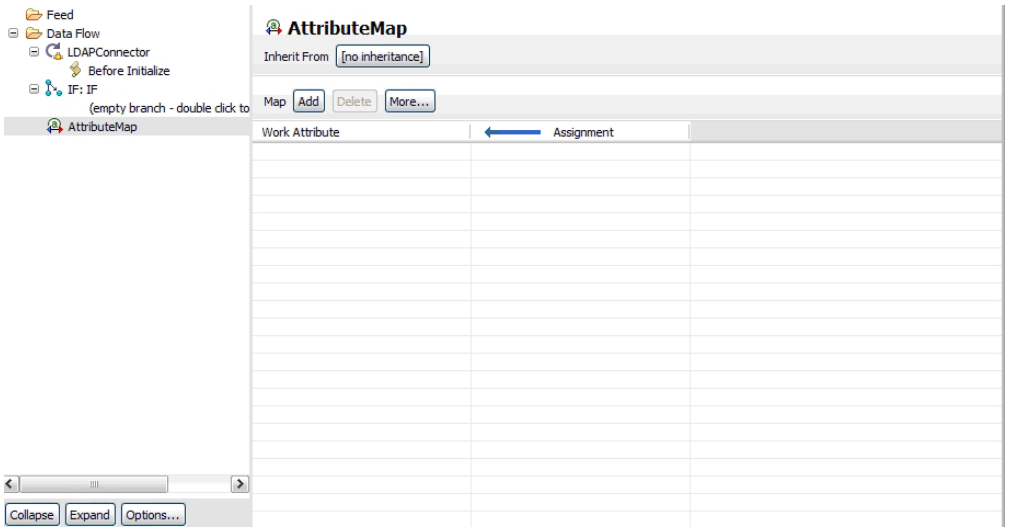


図 29. 独立属性マップ・コンポーネント

属性マップ・コンポーネントには 1 つのパネルが表示されます。このパネルでは、その他のコンポーネント (コネクターなど) 内部の属性マッピングから独立して、作業項目に属性をマップできます。また、このコンポーネントでは、その他のコンポーネント (コネクター、関数、およびライブラリー内のその他の属性マップ・コンポーネントなど) の属性マップを再利用できます。詳しくは、118 ページの『属性マッピングおよびスキーマ』を参照してください。

ユーザー・ドキュメンテーション・ビュー

場合によっては、AssemblyLine が非常に複雑になることがあります。他のユーザーが構成を簡単に読み取れるように、ドキュメンテーション・ビューを使用して AssemblyLine の一部を文書化することができます。

AssemblyLine の概要で、コンポーネントを右クリックして「**ユーザー・コメントの編集**」を選択すると、次のドキュメンテーション・ビューが表示されます。

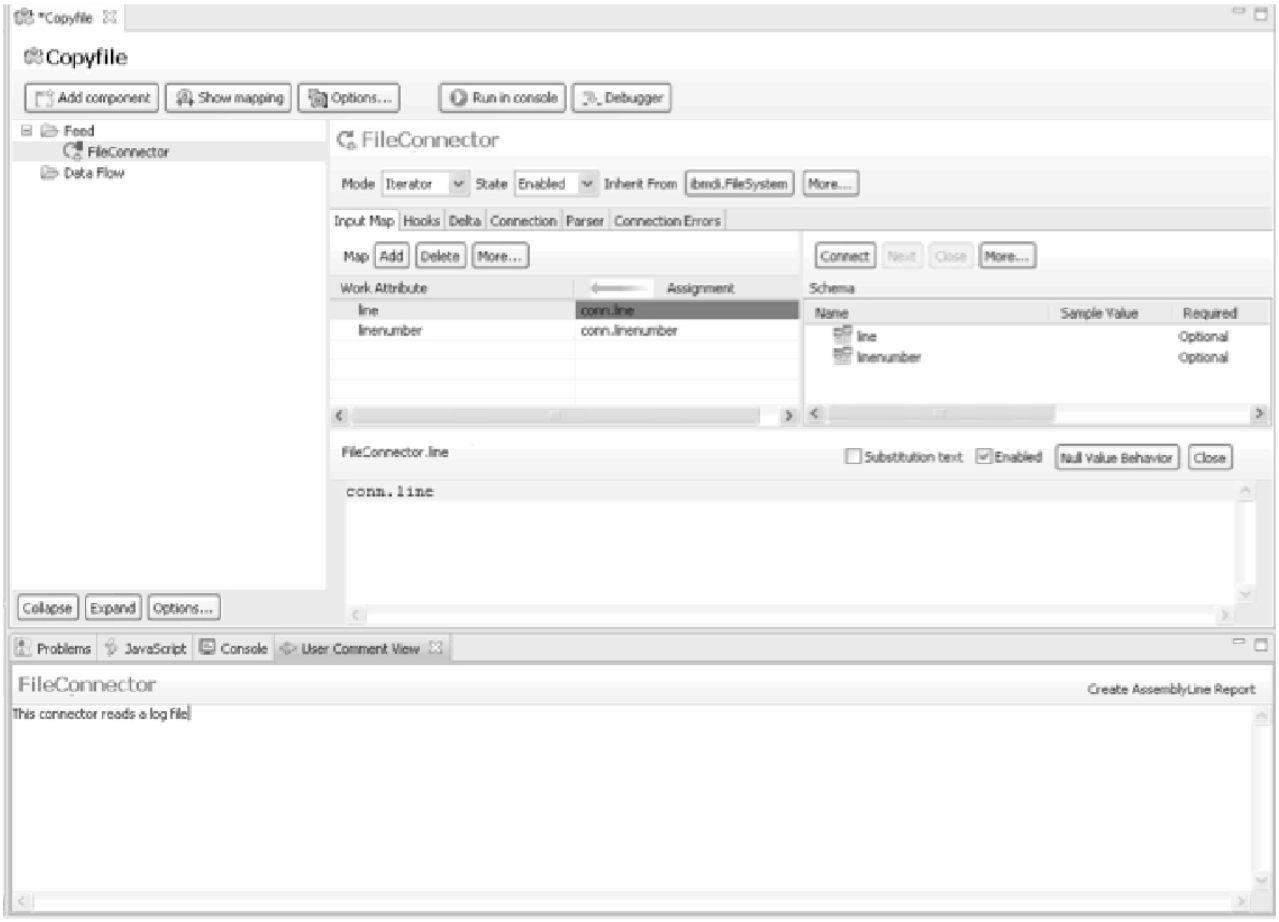


図 30. ユーザー・ドキュメンテーション・ビュー

AssemblyLine エディターの選択が変更されるにつれて、ドキュメンテーション・ビューに現在の選択が反映されます。ビューに書き込んだテキストは、AssemblyLine を保存すると保存されます。 コメントがあるコンポーネントは、コンポーネント・アイコンの左上隅に印が付きます。

「**AssemblyLine レポートの作成**」ボタンを押すと、AssemblyLine に入力されたすべてのユーザー・コメントが含まれたレポートが作成されます。使用されるレポート・テンプレートは、*TDI_Install_dir*/XSLT/ConfigReports ディレクトリーにある UserCommentsReport.xslです。

図 31. AssemblyLine レポートの例

「AssemblyLine の実行」ウィンドウ

AssemblyLine を実行すると、ログの出力を示したウィンドウが表示されます。

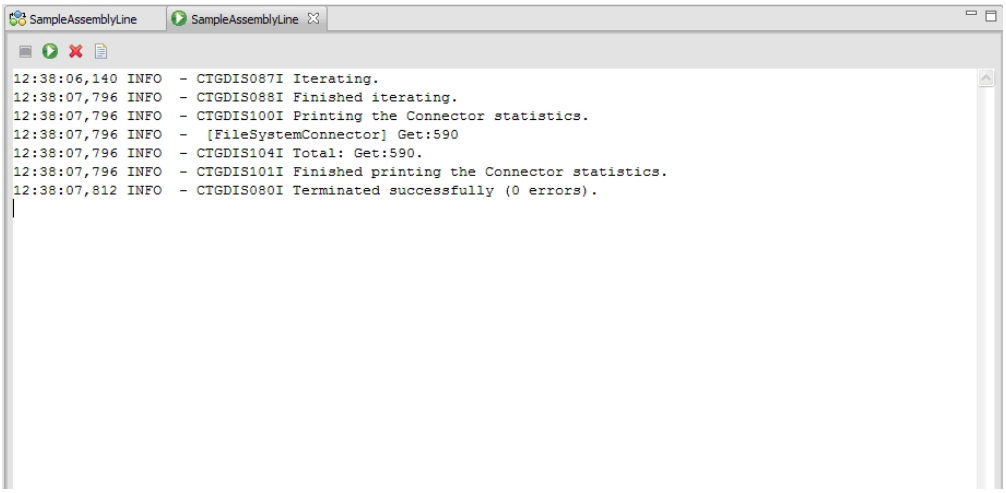


図 32. コンソール・ログ

この画面では、実行中の AssemblyLine を停止したり、AssemblyLine が終了した後に再始動したりすることができます。

注: AssemblyLine の停止とは、構成エディターが AL を実行中の構成インスタンス (通常はデフォルトのローカル・サーバー) に停止通知を送信することを意味します。これにより、即時にスレッドが強制終了されるわけではなく、サーバーが制御を取り戻すとすぐに実行が停止されます。これは、「**停止**」ボタンを押すと、AssemblyLine を実行していたサーバー・プロセス全体が強制終了される以前のバージョンの処理とは異なります。
このほかに 2 つのボタンがあります。1 つは、ログ・ウィンドウを消去するためのボタンで、もう 1 つはログ・ファイルを別のエディター・ウィンドウで開くためのボタンです。メモリー不足の問題を避けるために、ログ・ウィンドウには最新の数百行のみが表示されます。

ログは、接頭部が「tdi_ce_al_log」で、「.log」という拡張子を持つ一時ファイルに書き込まれます。このファイルは、プラットフォーム固有の一時ディレクトリー (TEMP/TMP 環境変数で定義されることが多い) に格納されます。「AssemblyLine の実行」ウィンドウを閉じると、ログ・ファイルは自動的に削除されますが、アプリケーションやマシンの破損があった場合は、これらのログ・ファイルを手動で削除しなければならないことがあります。このファイルに使用するエディターは、デフォルトでは単純なテキスト・エディターですが、別のエディター (外部エディターを含む) に「.log」という拡張子をマップさせることにより、デフォルトのエディターを変更することができます。「**ウィンドウ**」 > 「**設定**」メニュー・オプションを使用して、次のダイアログを開きます。

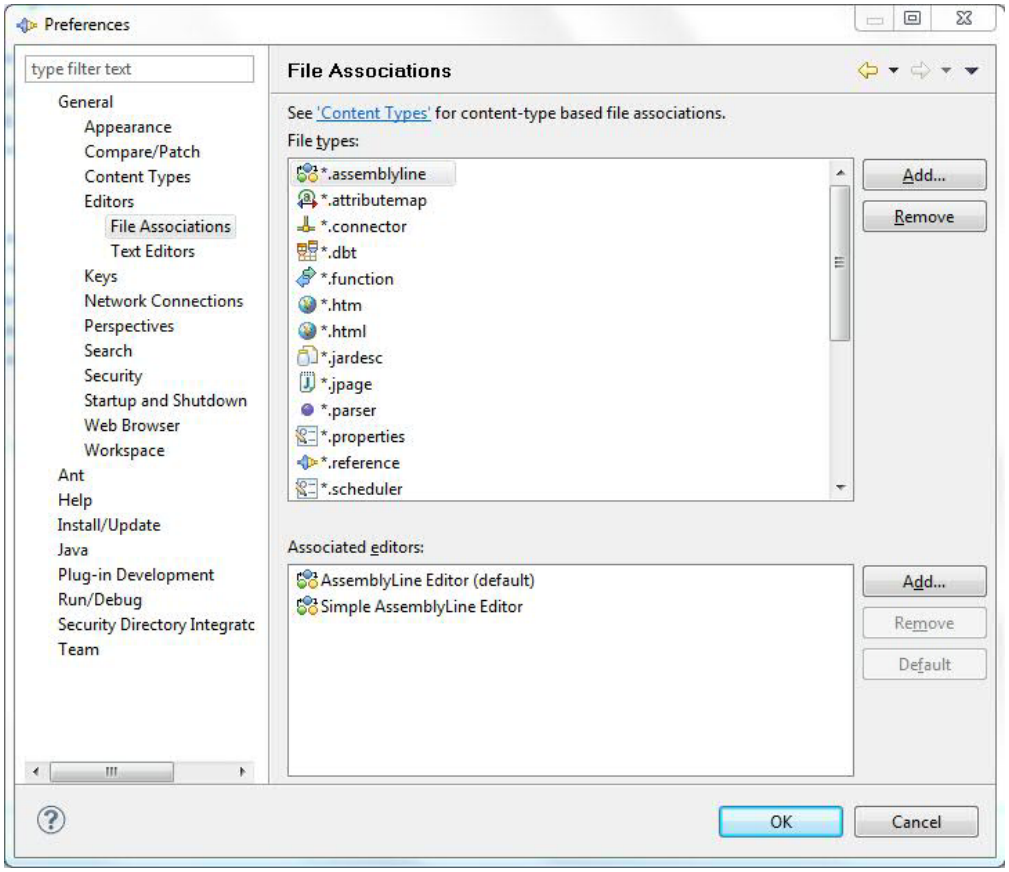


図 33. 構成エディターの「ファイルの関連付け」の設定

ここで、「.log」という拡張子を追加して、それをエディターに関連付けることができます。

属性マッピングおよびスキーマ

属性のマッピングは、AssemblyLine 内の「属性マップ」パネルを使用して行うか、コンポーネント・エディターで行います。

AssemblyLine エディター内では、以下に示すように、「属性マップ」セクション内で右クリックして「属性の追加」を選択するか、またはツールバーの「**追加**」ボタンを使用して、属性を追加することができます。

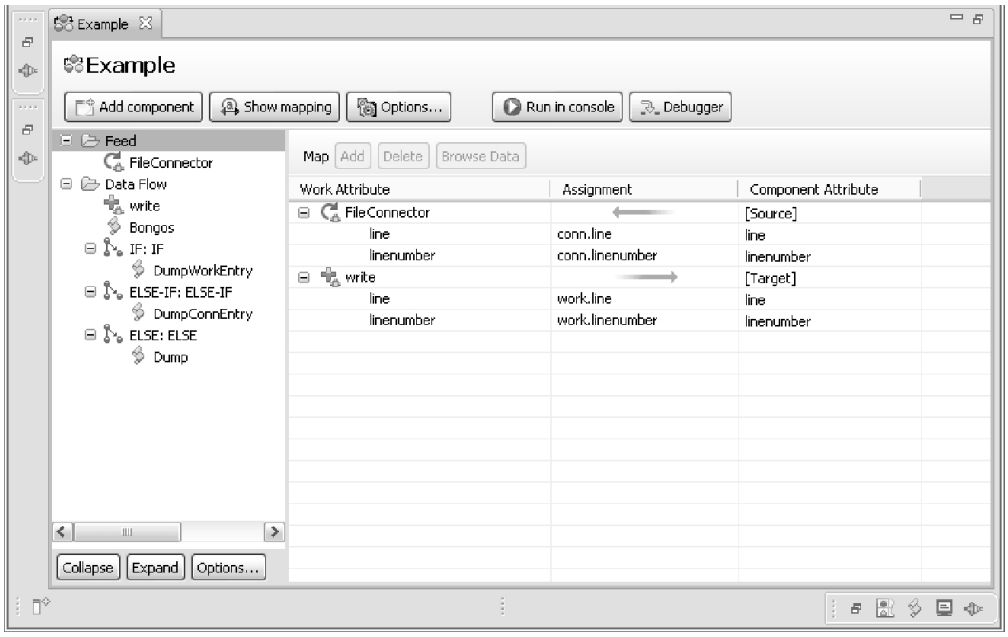


図 34. 属性マッピング

このウィンドウには、AssemblyLine 内のコンポーネントのスキーマは表示されません。スキーマを表示する場合は、左側のツリー内でコンポーネントを選択して、そのコンポーネントのエディターを開きます。

属性マッピングの標準的なシナリオでは、最初にコンポーネントのスキーマをディスカバーします。スキーマがディスカバーされると、CE はそのコンポーネントのスキーマの照会メソッドを実行するバックグラウンド・ジョブを実行します。スキーマが戻されない場合、CE は項目を読み取って、その項目からスキーマを取得するかどうかを確認するメッセージを表示します。その後、その結果が編集中のコンポーネントのスキーマとして取り込まれます。

次の図は、属性をディスカバーした後のコンポーネントの入力スキーマの内容を示しています。何らかの理由でコンポーネントのスキーマが表示されない場合は、ツールバーの「**追加...**」ボタンを使用してスキーマ項目を手動で追加するか、「**継承の変更**」オプションを使用して別のコンポーネントの構成からスキーマを再利用することができます。

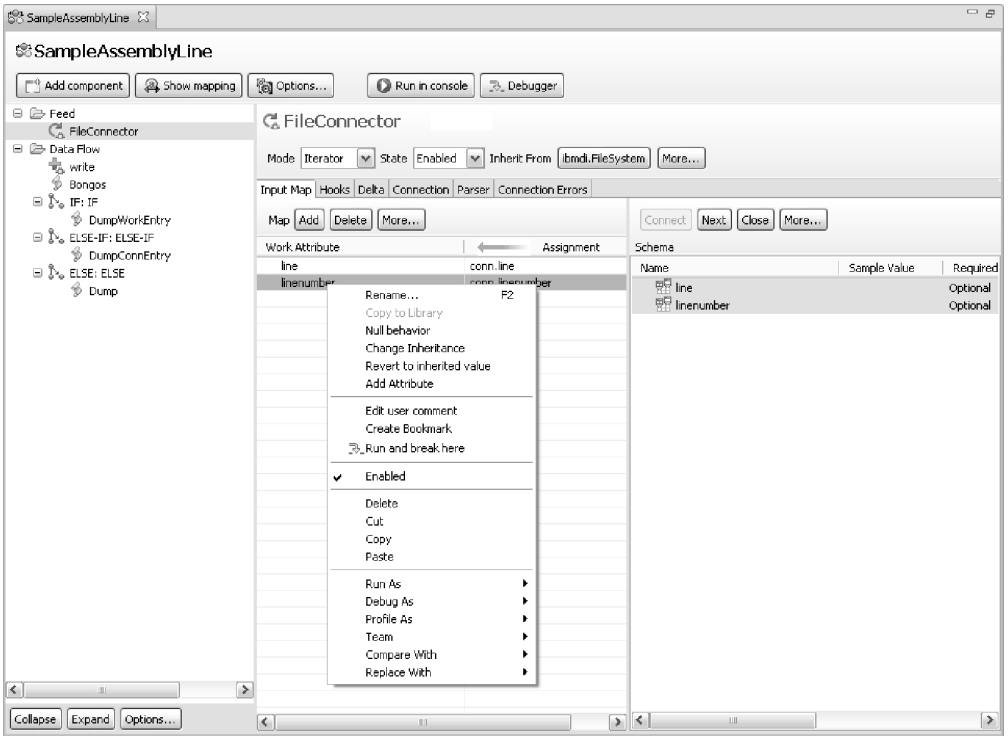


図 35. ディスカバーされた属性を表示した「属性マッピング」

タイトル・バーのドロップダウン・メニューを使用して、スキーマ構成の継承を変更することもできます。

スキーマが表示されたら、個々の項目を属性マップにドラッグ・アンド・ドロップするか、コンテキスト・メニューから「**属性のマップ**」機能を使用して、必要に応じてマッピングを変更することができます。

図 36. 属性マップの継承の変更

注: ドラッグ・アンド・ドロップの機能性は、ある程度まではご使用のウィンドウ操作環境に依存します。特に、UNIX システムの場合、CDE (Common Desktop

Environment) ではドラッグ・アンド・ドロップを実行できません。したがって、マッピングをセットアップするためには、コンテキスト・メニューから「**属性のマップ**」機能を使用する必要があります。

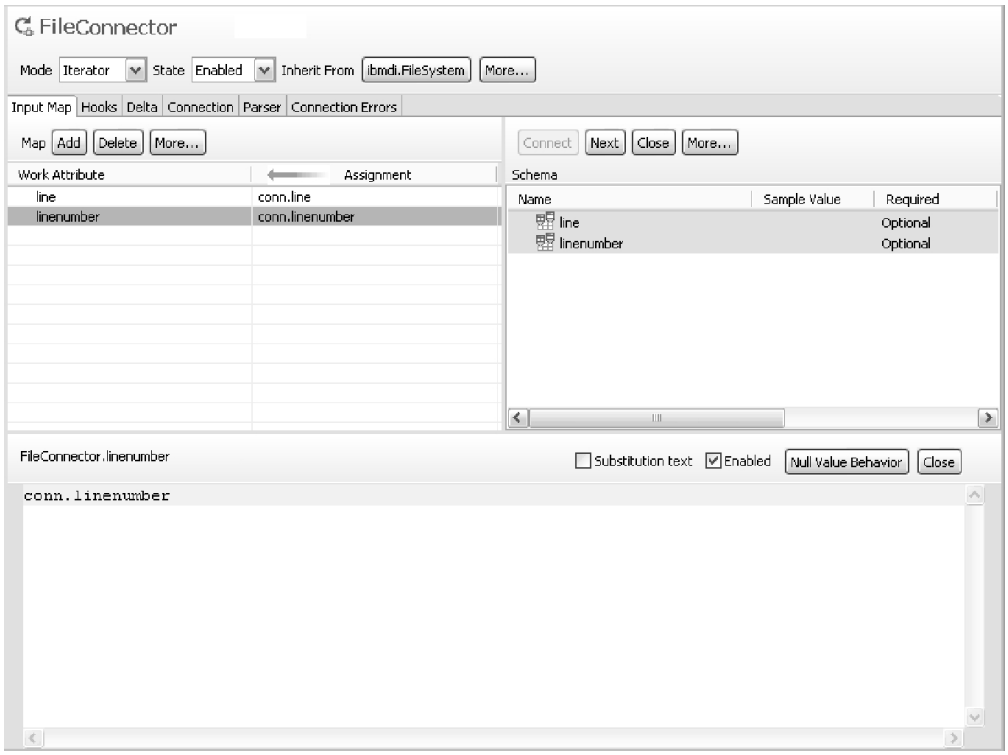


図 37. 個々の属性用の JavaScript 編集ウィンドウを表示した「属性マッピング」

スキーマがない場合、またはスキーマに関係のない属性を追加したい場合は、当然それも可能です。「**追加**」ボタンを使用して、新しい属性をマップに追加します。その属性に名前を付けます。新しい属性には、「conn.*attribute-name*」または「work.*attribute-name*」のいずれかの式が割り当てられます。これは、AssemblyLine エディター、およびコネクター・エディターのどちらのウィンドウで行うこともできます。

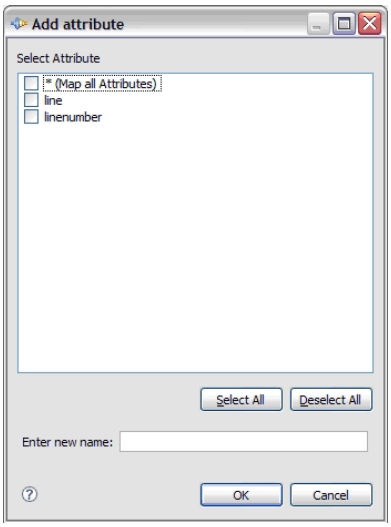


図 38. 「属性の追加」ダイアログ

新しい属性の名前を入力できる編集可能なテキスト・フィールドがあるダイアログが表示されます。上記のリストには、スキーマからの既知の属性名すべてが表示されます。属性マップに追加したい属性名を選択することができます。

AssemblyLine にさらにコンポーネントを追加するときに、コンポーネント間で属性をドラッグすることができます (ドラッグが意味を成す場合)。コンポーネントを別のコンポーネントにドラッグすると、マップ済みの属性すべてが宛先のコンポーネントにマップされます。属性マップから、AssemblyLine のすべてのコンポーネントを表示する左のパネル内のコンポーネントへ、属性をドラッグすることもできます。これにより、ドラッグするこれらすべての項目の単純マッピングが行われます。これは、「属性マップ」パネル内のコンポーネントにこれらの項目をドロップすることと同様です。

「属性マッピング」の概念は、「スタートアップ・ガイド」でかなり広範囲にわたって、豊富な例を使って説明しています。

コネクター、およびコネクターが構成されたモードに応じて、「コネクター構成」ウィンドウにはさまざまなタブが表示されます。

- 接続されたシステムからの入力をサポートするモードのコネクターには、「**入力属性**」というセクションがあります。
- 接続されたシステムへの出力をサポートするモードのコネクターには、「**出力属性**」というセクションがあります。
- 一部のコネクターは、入力と出力の両方を行うモードをサポートします。そのように構成されている場合は、「**入力属性**」セクションと「**出力属性**」セクションの両方が表示されます。

外部の属性マップ

属性マップは、外部の属性マップ・ファイルから継承できます。外部の属性マップ・ファイルとは、実際のマッピング画面に表示されているのと同じ属性マップ項目が含まれているテキスト・ファイルです。異なるのは、外部ファイルに使用されている形式が、内部の XML 構造と違うということです。これにより、構成エディ

ターを使用しなくても、コネクター用の属性マップを簡単に構成できます。このオプションは、次のように、構成エディターの属性マップの継承ダイアログで使用できます。

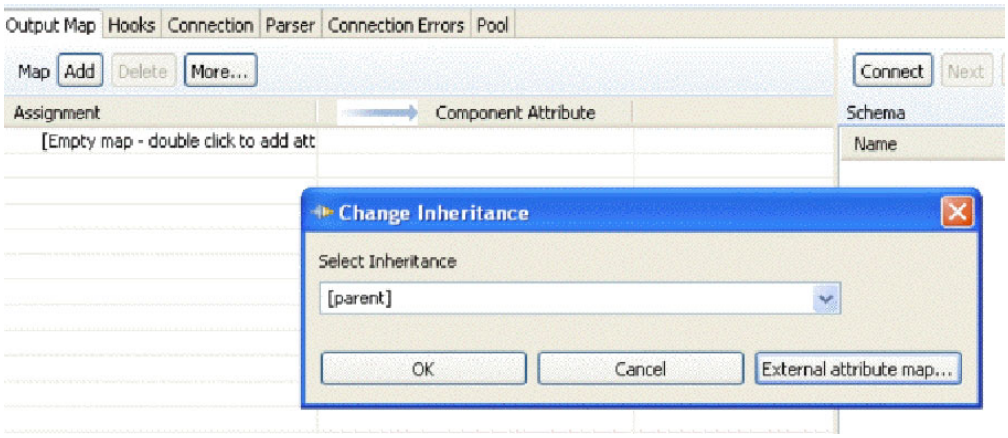


図 39. 属性マップ: 継承ダイアログ

「**外部の属性マップ**」ボタンをクリックして既存のファイルを選択するか、または「`file:`」に続けて属性マップ・ファイルへの絶対パスを入力します。相対パス名を使用する場合は、ファイル名の前にドットとスラッシュ (*./*) を追加します。

入力属性マッピング:

入力属性マッピングは、入力ソースから AssemblyLine 内の作業項目へのデータ移動を遂行するプロセスです。入力属性マップは、コネクター・エディターに呼び出されると、コネクターの「属性マップ」ウィンドウに表示されますが、この場合、「[ソース]」と呼ばれるエンティティーからコネクターを指す矢印が付いています。これらの入力属性マップは、「入力属性マップ」の下の「スキーマ」ウィンドウにも表示されます。

始める前に

入力属性マップをセットアップ可能にするには、コネクターのツールバーで、コネクターを入力がサポートされるモードに設定する必要があります。入力がサポートされるモードは通常、イテレーター、ルックアップ、およびサーバーです。

次に、「**入力マップ**」セクションで、入力ソースから AssemblyLine で処理する属性を選択します。

このタスクについて

入力属性マッピング用にセットアップするコネクターは、`<workspace>/Resources/Connectors` に置くことも、AssemblyLine 内の指定の位置に置くことも可能です。

手順

1. 「**入力マップ**」をクリックします。
2. 「**接続**」、「**次へ**」の順にクリックして、多数のデータ・ソースのスキーマを取得します。コネクター、またはコネクターとパーサーの組み合わせによっては、

事前定義されたスキーマがある場合もありますが、データ・ソースからサンプル項目を読み取り、それを調べて属性をディスカバーするよう、プロンプトが出される場合もあります。

3. 最後に、「**スキーマ**」リストから属性を選択して属性マップまでドラッグするか、「追加」および「削除」ボタンを使用して手動で属性を追加します。属性マップは、変換の指定と同様に、処理のために AL に提供する属性を制御します。

次のタスク

これらのマップされた属性は、データ・ソースから取り出され、作業 項目に入れられた後、AssemblyLine 内の「フロー」セクションの後続のコネクターに渡されます。

コネクターを直接 AssemblyLine 内に作成しなかった場合に、このコネクターを AssemblyLine で使用するためには、コネクターを `<workspace>/Resources/Connectors` 内のその場所から AssemblyLine の「**フィード**」セクションにドラッグします。

出力属性マッピング:

出力属性マッピングとは、AssemblyLine の作業項目から接続システムの出力宛先へデータを移動する処理です。コネクター・エディターでコネクターの「属性」ウィンドウを表示すると、コネクターから「[Target]」と示されているエンティティーを指す矢印として出力属性マップが示されます。これらはまた、「スキーマ」ウィンドウの出力属性マップの下にも表示されます。

始める前に

出力属性マップをセットアップできるようにするため、コネクターのモードとして、コネクターの出力とツールバーをサポートするモードを設定する必要があります。出力の標準的なモードは AddOnly です。CallReply などの一部のモードでは、入力と出力の両方がサポートされます。

次に「出力属性」セクションの「**出力マップ**」で、接続システムに出力する AssemblyLine の作業項目の属性を選択します。

このタスクについて

`<workspace>/Resources/Connectors` または AssemblyLine 内の指定の場所に存在しているコネクターの出力属性マッピングをセットアップできます。ただしコネクターが `<workspace>/Resources/Connectors` にのみ存在し、AssemblyLine のメンバーではない場合、作業項目属性を出力属性マップに容易にドラッグすることができません。この場合、コネクターを AssemblyLine にドラッグするか、またはマッピングを手動で作成します。マッピングを手動で作成するには、「属性マップ」ウィンドウの「**追加**」をクリックするか、または「属性マップ」ウィンドウのコネクターを右クリックして「**属性マップ項目の追加 (Add attribute map item)**」を選択します。

手順

1. 「**出力マップ**」をクリックします。

2. 「**接続**」をクリックして、データ・ソースのスキーマを取得します。一部のコネクターまたはコネクターとパーサーの組み合わせには事前定義のスキーマがあり、これが表示されます。ただし、多数のコネクターには事前定義のスキーマはありません。
3. コネクターが AssemblyLine に含まれている場合は、以前にマップした作業項目属性を、AssemblyLine エディターの「属性マップ」ウィンドウのコネクターにドラッグします。あるいは、属性を手動で作成します。名前の突き合わせは実行時に行われます。例えば、`some_attribute` として作成された出力マップ属性マップの項目の場合、some_attribute という名前の作業項目属性が、同名の接続システム属性にマップされます。

次のタスク

AssemblyLine のフローでこのコネクターが呼び出されると、これらのマップされた属性が作業 項目から取得され、接続システムに出力されます。

AssemblyLine にコネクターを直接作成しない場合は、AssemblyLine でこのコネクターを使用するため、コネクターを `<workspace>/Resources/Connectors` 内の位置から AssemblyLine の「**フロー**」セクションにドラッグします。

コネクター・エディター

コネクター・エディターは、コネクター・ファイルを編集するとき、および AssemblyLine 内部のコネクターに対して「**編集**」機能を使用するときに使用されます。

コネクターの作成方法の概要については、126 ページの『コネクターの作成』で説明します。

このエディターでは、コネクターのウィザードおよびポップアップ・ダイアログと同じウィジェットが使用されています。エディターには 6 つのタブがあります。タブには接続のさまざまな設定の構成パネルが表示されます。上部には、コネクターの主要属性 (モード、状態、およびその他の一般的なコネクター・オプション) が表示されます。

次のタブがあります。

1. 127 ページの『入力属性マップおよび出力属性マップ』
2. 127 ページの『フック』
3. 128 ページの『接続』
4. 129 ページの『パーサー』
5. 129 ページの『リンク基準』
6. 131 ページの『接続エラー』
7. 133 ページの『デルタ』
8. 135 ページの『プール』
9. 136 ページの『コネクターの継承』

コネクターの作成

コネクターの作成とは、コネクターの配置場所と、そのコネクターに割り当てる初期パラメーターを決めることです。

始める前に

コネクターの配置場所を、以下の 2 つの場所のいずれかに決めます。

1. 再利用およびリソース共有のためのコネクターは、<workspace>/Resources/Connectors ディレクトリーに配置します。一般に、これがコネクターの作成と保守に最適の場所です。このように定義したコネクターを AssemblyLine に追加するには、該当する場所にドラッグします。

   その後は、AssemblyLine で指定の役割を果たすようにコネクターの設定の一部を変更できますが、大半の設定はそのまま継承されるため、「リソース」セクションでの定義から変わることはありません。
2. コネクターを AssemblyLine で直接作成することもできます。この方法で定義したコネクターは、その特定の AssemblyLine のコンテキストでのみ有効な暫定定義となります。

このタスクについて

コネクターは、IBM Security Directory Integrator で作成されるソリューションのバックボーンを形成し、データの交換相手のシステムとの接続を確立します。

手順

1. ワークスペースで右クリックして「**リソース**」 > 「**コネクター**」 > 「**新規コネクター...**」を選択するか、「**ファイル**」 > 「**新規**」 > 「**コネクター**」を選択します。
2. 新規コネクターを作成するロケーションに移動して、新規コネクターに名前を付けます。
   a. 推奨ロケーションは、<workspace>/Resources/Connectors です。
   b. あるいは、新規コネクターを AssemblyLine で直接作成することもできます。ターゲットの AssemblyLine に移動します。イテレーター・モードのコネクターとサーバー・モードのコネクターの場合は「フィード」セクション、それ以外のすべてのモードでは「フロー」です。
3. 「**完了**」をクリックすると、コネクターが作成されます。
4. 「**接続**」タブで、新規コネクターのモードを必要なモードに設定します。
5. 「**接続**」タブで、このコネクターの接続パラメーターを設定します。必須パラメーターにはアスタリスク (*) のマークが付いています。コネクターの種類によっては、さらに「**パーサー**」タブでパーサーの構成が必要になることがあります。
6. 「コネクター構成」ウィンドウの「**属性マップ**」ウィンドウに移動して、このデータ・ソースのスキーマをディスカバーまたは定義します。データ・ソースのスキーマを取得するには、「**属性のディスカバー**」をクリックします。コネクター同士、またはコネクターとパーサーの組み合わせでは、スキーマが事前に定義されている場合があります。そうでない場合は、データ・ソースからサンプル項目を読み取り、それを調べて属性をディスカバーするようにというプロンプトが出されます。

次のタスク

コネクターの定義が終われば、AssemblyLine との間でやり取りする情報 (これを*属性* といいます) をセットアップする準備が整ったことになります。このプロセスは*属性マッピング* と呼ばれ、AssemblyLine からの視点になっています。つまり、接続されたシステムから入力コネクターを経由して AssemblyLine の作業項目に至る属性のマッピングの定義は、*入力属性マップ* で行われます。この逆の、作業項目から出力コネクターを経由して接続されたシステムに至る属性のマッピングは、*出力属性マップ* で行われます。

入力属性マップおよび出力属性マップ

「属性マップ」タブには、1 つのコンポーネントの入力属性マップと出力属性マップ、およびスキーマが表示されます。

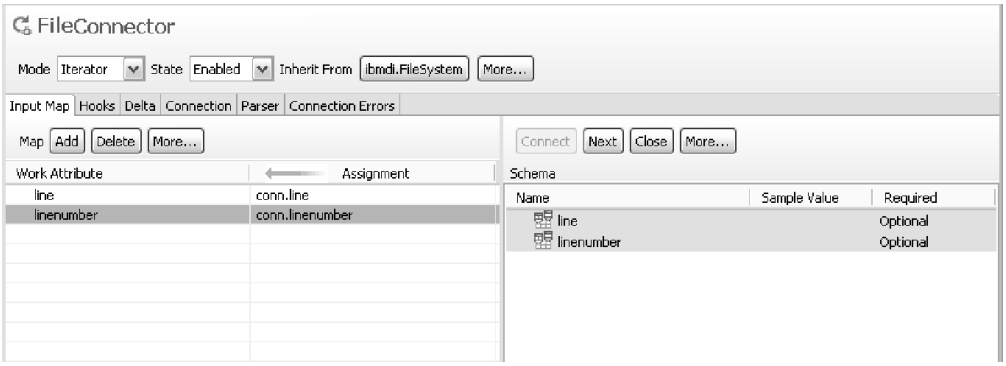


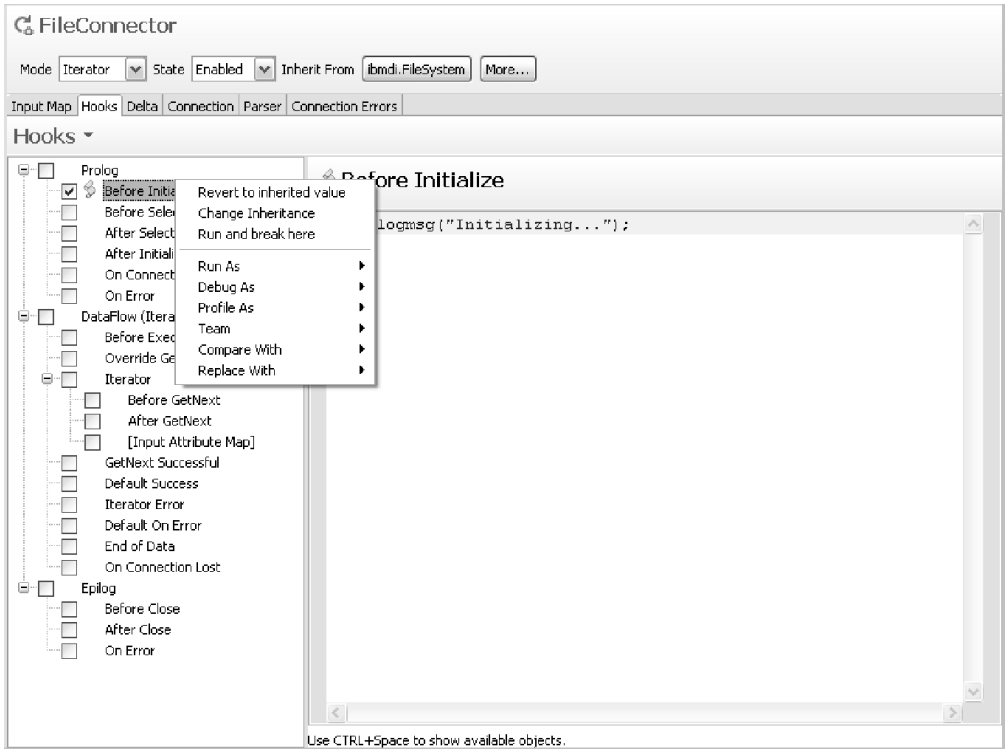
図 40. 「属性マップ」ウィンドウ

このウィンドウの説明については、AssemblyLine エディターの 118 ページの『属性マッピングおよびスキーマ』のセクションを参照してください。

フック

「フック」タブには、コネクターのすべてのフックが表示されます。

チェック・ボックスを使用してフックを使用可能または使用不可にしたり、フックを選択してその内容を編集したりします。フックの内容を変更すると、そのフックは自動的に使用可能になります。スクリプト・コードが含まれているフックでは、ツリー・ビュー内にスクリプト・アイコンが表示されるため、そのフックに内容があるかどうかをすぐに見分けることができます。使用可能なフックは、実行フロー内に到達すると (スクリプトを含むかどうかにかかわらず) 実行されることに注意してください。



AssemblyLine レベル、および個々のコンポーネント・レベルの各種のフックの説明については、 34 ページの『AssemblyLine フローおよびフック』のセクションを参照してください。

接続

図 41. 「接続」タブ

「接続」タブ内のパラメーターは、特に構成するコンポーネントに特異的なものです。「リファレンス」にあるコンポーネントの個々の仕様を参照してください。

パーサー

コネクターがパーサーを使用できる場合 (またはパーサーを必要とする場合) は、パーサーのためのタブも表示されます。

コネクターに対するパーサーを変更するには、「**パーサーの選択**」ツールバー・ボタンを使用します。

例えば、行リーダー・パーサーには、以下に示すような構成画面があります。

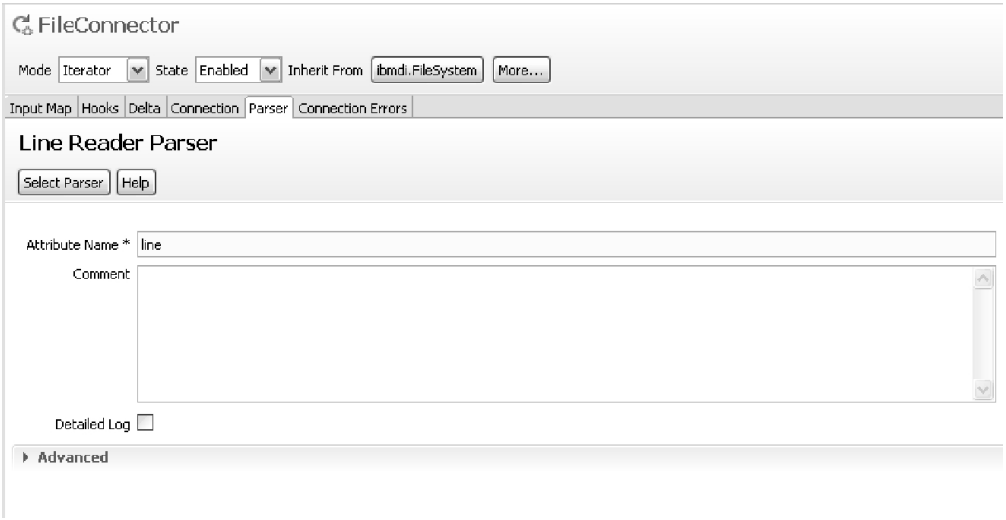


図 42. 行リーダー・パーサー

リンク基準

コンポーネントにリンク基準が必要な場合は、「リンク基準」タブを表示します。

コンポーネントで実際に「リンク基準」タブが表示されるかどうかは、コンポーネント・タイプだけではなく、そのモードによっても異なります。詳しくは、21 ページの『リンク基準』を参照してください。

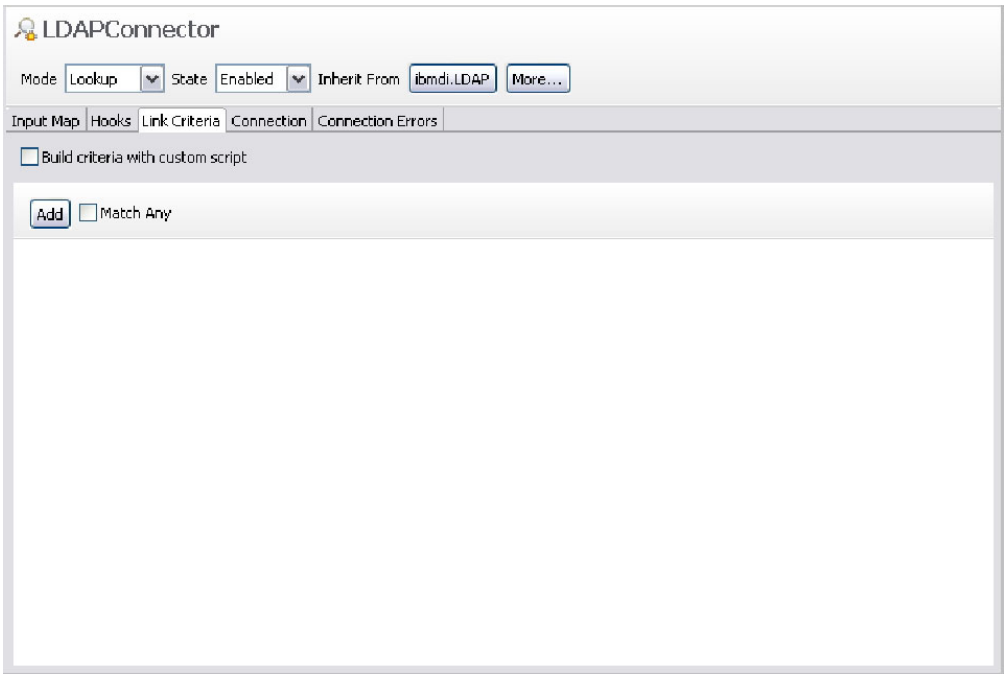


図 43. 「リンク基準」タブ

新しい制御の行をビューに追加するには、「**追加**」ボタンを使用します。個々の行を除去するには、「**削除**」ボタンを使用します。このビューで、属性名、オペランド (例えば、等しい、含む) および一致する値を指定します。一致する値は、定数または式のいずれでもかまいません。以下のように、  ボタンを使用して、式エディターを表示します。

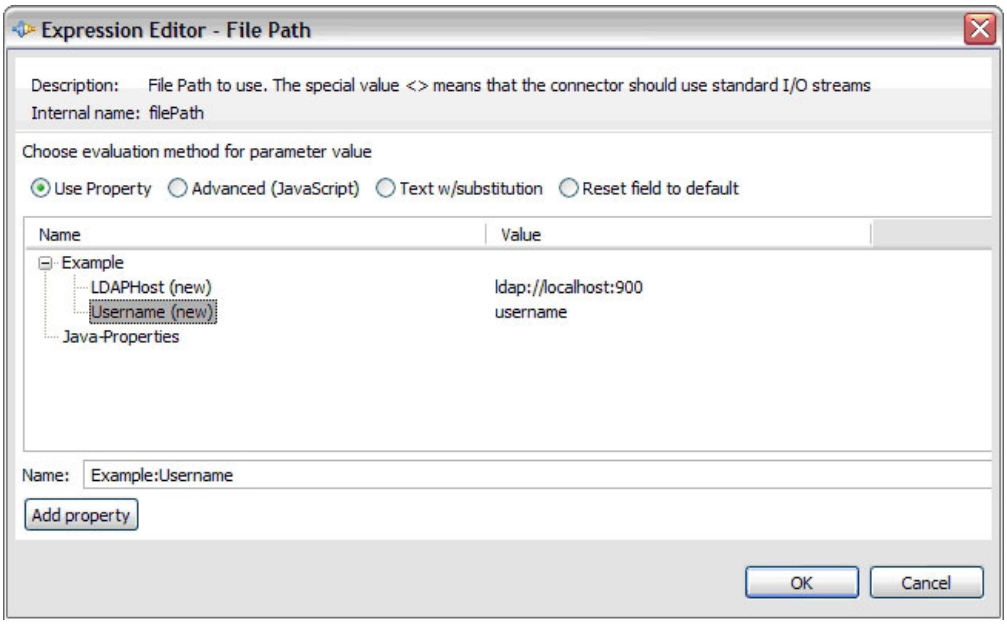


図 44. 「式エディター」ウィンドウ、単純モード

式エディターでは、ご使用のプロパティー・ファイルの中から簡単にプロパティーを選択したり、JavaScript コードに基づいて値が戻される拡張モードを使用したりすることができます。プロパティー選択と JavaScript コードとの切り替えを行うには、「**拡張 (JavaScript)**」チェック・ボックスにチェック・マークを付けます。code.使用できるのは、これらのいずれか 1 つだけです。

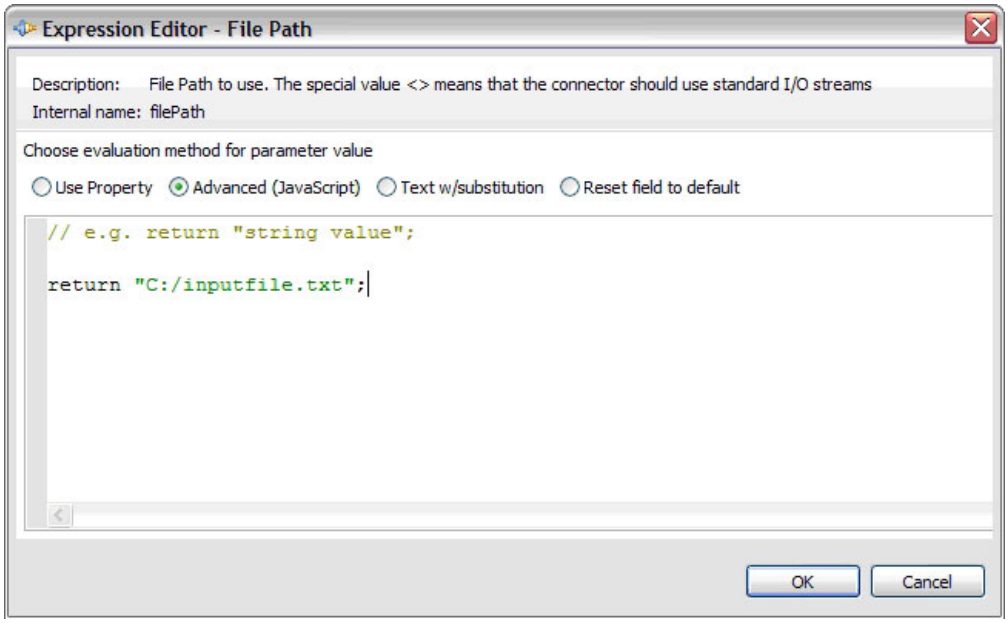
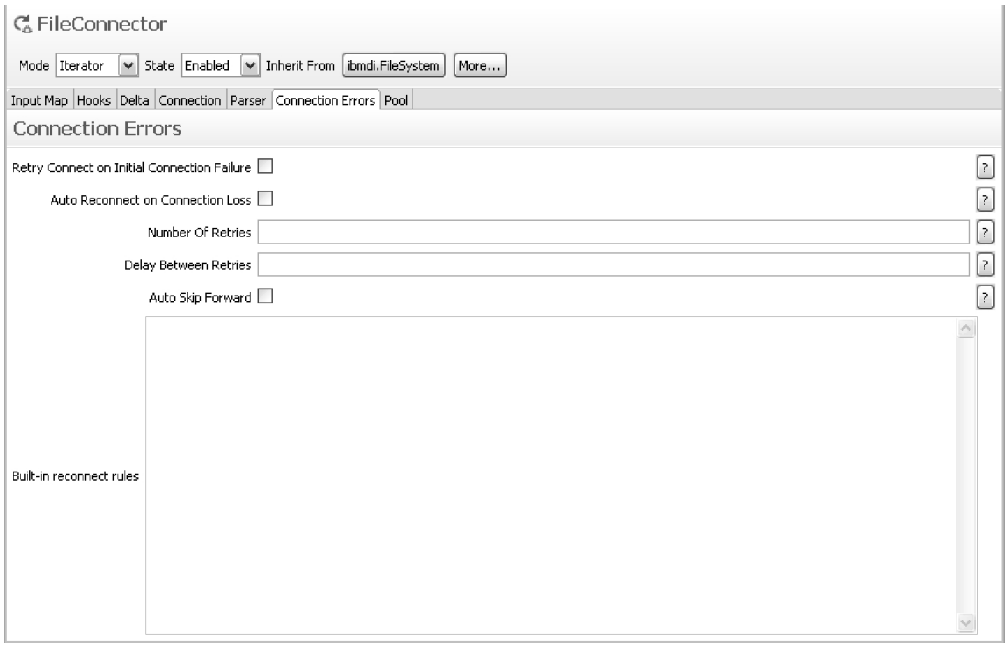


図 45. 「式エディター」ウィンドウ、拡張 (JavaScript) モード

接続エラー

「接続エラー」タブでは、接続回復力 (コンポーネントから行った接続が失敗したときに実行する自動化動作) を構成できます。

「**初期接続失敗時に接続を再試行する**」を選択するときに、オプションとして、開始時に接続に失敗したときに例外をスローするかどうか、または再試行するかどうかを構成することができます。このフラグを設定すると、コネクターの初期化時に接続を確立できなかった場合、「再接続」が試行されます。まず初めの段階で接続が確立されていないため、実際にはこれは再接続ではありませんが、一般に確立済みの接続が失われたときに発生する可能性のある状態に対するものと同じメカニズムです。



確立済みの接続では、残りのパラメーターには以下の意味があります。

接続損失時に自動再接続する
このフラグを設定すると、コネクターが初期化された後に接続が失われた場合に、再接続が試行されます。

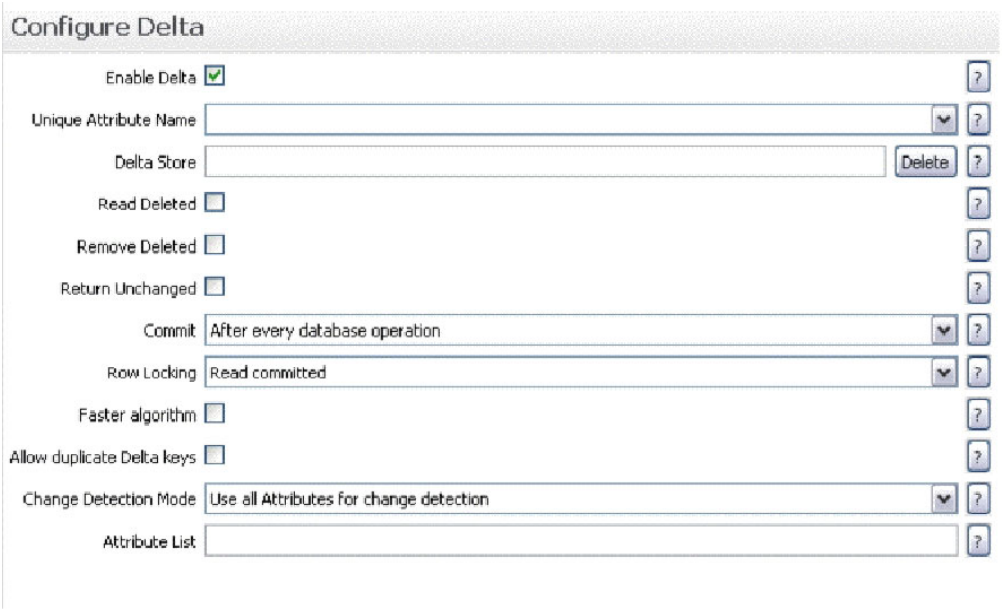
再試行回数
問題の発生時に再接続が試行される回数。この回数試行した後、再接続は断念されます。後で新しい問題が発生した場合、この同じ回数の試行が行われます。

再試行間隔
再接続が試行されてから次にまた試行されるまでの待機秒数、および最初に再接続が試行されるまでの秒数です。

自動前方スキップ
再接続の後に、正常な読み取りの回数と同じ数だけ自動的に前方にスキップします。

組み込み再接続ルール
これらは、再接続ルール・エンジンに関連付けられます。詳しくは、「*インストールと管理*」の対応するセクションを参照してください。

デルタ



このタブは、イテレーター・モードでのみ使用できます。

このタブ内のパラメーターには、以下の意味があります。

デルタを使用可能化
これは、このコネクター用のデルタ・エンジンのマスター・スイッチです。これを選択しないと、以下のパラメーターは使用可能になりません。

固有属性名
これは、特定のデータ・ソース内に固有値を持つ属性の名前、または、選択された複数の入力属性を「+」で区切ったものです。「**重複デルタ鍵を許可**」が有効になっている場合を除いて、重複鍵を持つデータ・ソースをデルタ関数の対象とすることはできません。詳しくは、17 ページの『デルタ検出』を参照してください。

デルタ・ストア
後続の実行での相違点を検出できるようにするために、このコネクターに関する以前の実行からのデルタ情報を格納するシステム・ストア内のテーブル。

削除済み項目の読み取り
このパラメーターにチェック・マークを付けた場合、イテレーターが繰り返しを完了したとき、つまり入力を終了したときに、AssemblyLine が削除済み項目を AssemblyLine の実行に投入します。命令コードは、この項目が入力ソースから削除されたことを示します。削除タグが付加された項目は、「削除済み項目の除去」フラグも有効にしない限り、デルタ・ストアから除去されないことに注意してください。

削除済み項目の除去
このパラメーターにチェック・マークを付けた場合、入力ソースから削除された項目がデルタ・ストアから削除され、それ以降の実行では検出されなくなります。

未変更項目を戻す

このパラメーターにチェック・マークを付けた場合、この実行で変更されなかった項目が AssemblyLine に投入されます。

コミット

入力の繰り返しの結果としてデルタ・ストアに行った変更をコミットするタイミングを選択します。次の項目を選択できます。

- 各データベース操作後
- AL サイクル終了時
- コネクターのクローズ時
- 自動コミットなし

デフォルトは、「**各データベース操作後**」です。

行のロック

デルタ・ストアへの接続に使用するトランザクション分離レベルを選択します。このパラメーターは、複数の DB クライアントが同じデータにアクセスするときに、*トランザクション分離レベル* を設定することにより、デルタ・ストア・テーブル内で行のロックを行う必要がある場合に対応します。高い分離レベルを設定すると、行とテーブルのロックを使用することにより、「ダーティー読み取り」、「非再現読み取り」、および「ファントム読み取り」として知られるトランザクション異常が削減されます。

次の項目を選択できます。

- READ_UNCOMMITTED
- READ_COMMITTED
- REPEATABLE_READ
- SERIALIZABLE

デフォルトは READ_COMMITTED です。詳しくは、セクション 227 ページの『行のロック』を参照してください。

属性リスト

これは、変更の計算の処理中に、変更を検出するか無視する対象の属性のコンマ区切りリストです。リスト内の属性の変更は、変更を無視または検出するかどうかを指定する「**変更検出モード**」パラメーターによって影響されます。このパラメーターおよび次のパラメーターの詳細については、セクション 229 ページの『特定の属性内の変更のみの検出または無視』を参照してください。

変更検出モード

「**属性リスト**」パラメーター内にリストされている属性の変更を検出するか、無視するかを指定します。値は以下のとおりです。

- IGNORE_ATTRIBUTES
- DETECT_ATTRIBUTES
- DETECT_ALL

高速アルゴリズム

このパラメーターにチェック・マークを付けた場合、AssemblyLine に対して、使用するメモリー量が増えることを代償として、変更の計算に高速アルゴリズムを使用するように指示します。

重複デルタ鍵を許可

長時間実行する AssemblyLine で変更ログ/変更検出コネクターに対してデルタ機能が使用可能になっていると、項目が複数回変更される可能性があります。変更が複数回行われるとその項目を 2 度受け取ることになるため、重複デルタ鍵の例外がスローされます。このパラメーターにチェック・マークを付けた場合、重複鍵属性を持つ項目 (「**固有属性名**」パラメーターで指定) をデルタが有効なイテレーター・コネクターで処理できるようになります。

プール

コネクターのプール定義は、コネクターがコネクター・ライブラリー内 (つまり、プロジェクト/リソース/コネクター内のファイル) に存在する場合にのみ表示されます。 AssemblyLine から開いたコネクターでは、このタブは表示されません。

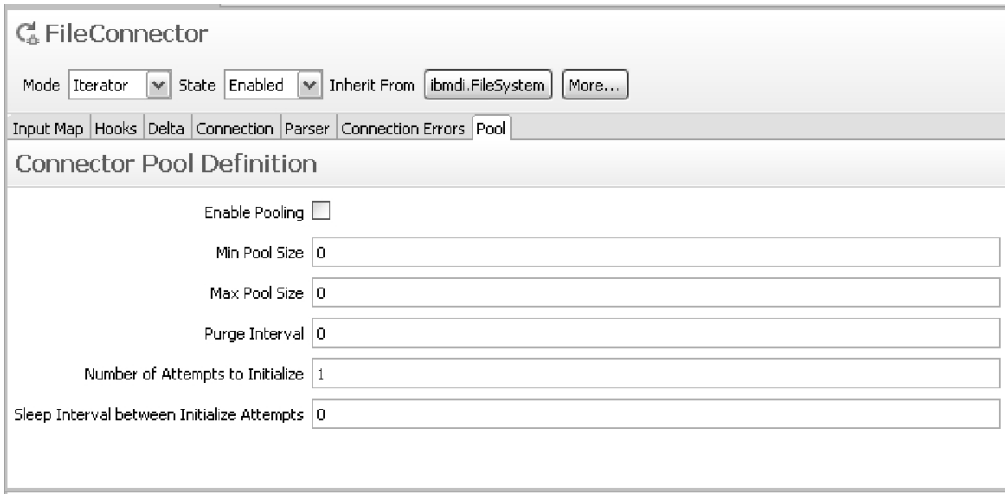


図 46. 「プール」タブ: 「コネクター・プール定義」

AssemblyLine でプールされたコネクターが使用された場合、そのコネクターでは代わりに次のタブが表示されます。

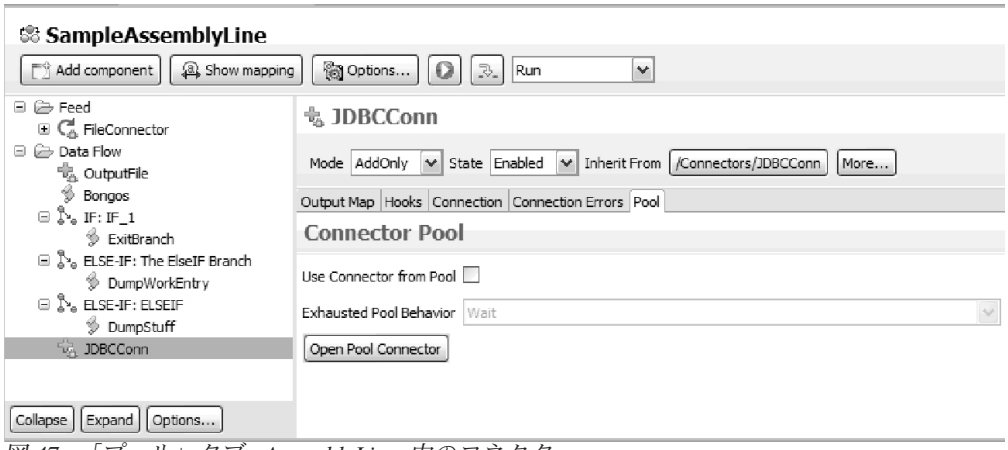


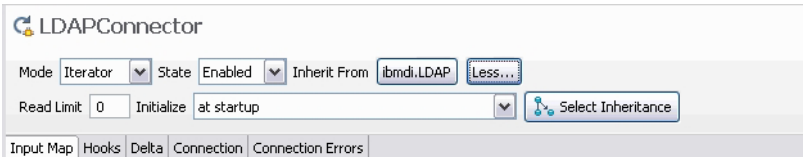
図 47. 「プール」タブ: AssemblyLine 内のコネクター

プールされたコネクターを開くには、「**プール・コネクターのオープン**」ボタンを使用します。

コネクターの継承

エディター内のさまざまな場所で継承を変更することができますが、すべての継承設定のリストが必要な場合は、コネクターの継承ボタンを使用することもできます。

「**その他...**」ボタンを使用して、コネクター・エディターのヘッダーを展開し、「**継承**」ボタンをクリックします。



「**継承の選択**」ボタンをクリックすると、そのコネクターのすべての継承設定を示す、次のようなダイアログが表示されます。

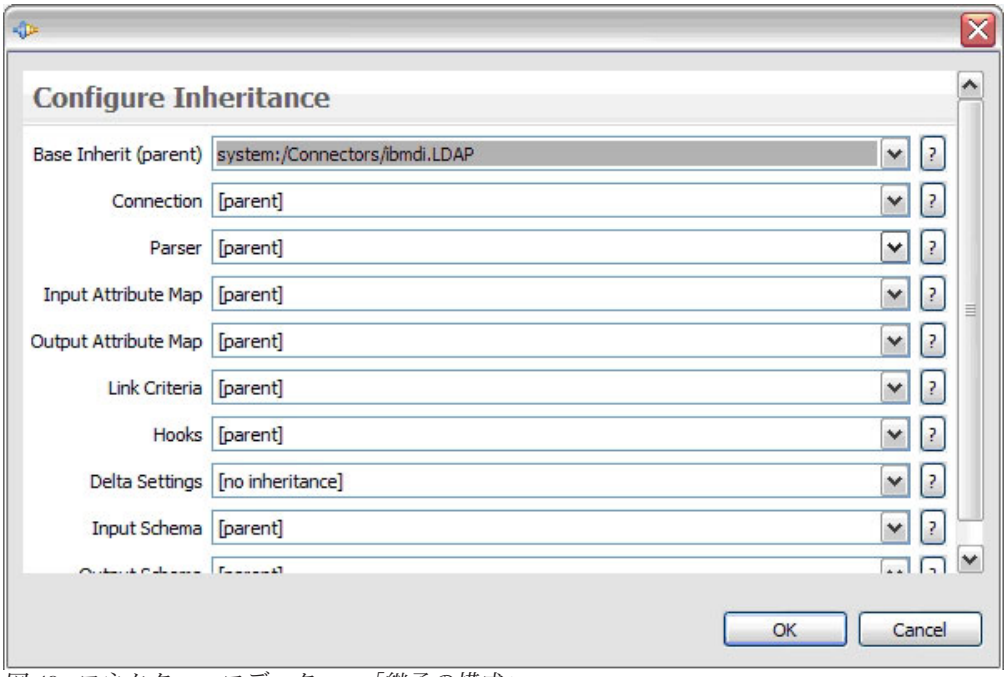


図 48. コネクター・エディター: 「継承の構成」

[親] という表記は、その項目が「**基本継承 (親)**」フィールドに示されているコンポーネントから継承されていることを意味します。

サーバー・エディター

サーバー・エディターを使用して、IBM Security Directory Integrator サーバーへの接続方法を定義します。

CE では常に「Default」という名前のサーバーが 1 つ定義されています。 IBM Security Directory Integrator のサーバー・ビューで、プロジェクトに新規サーバーを追加できます。

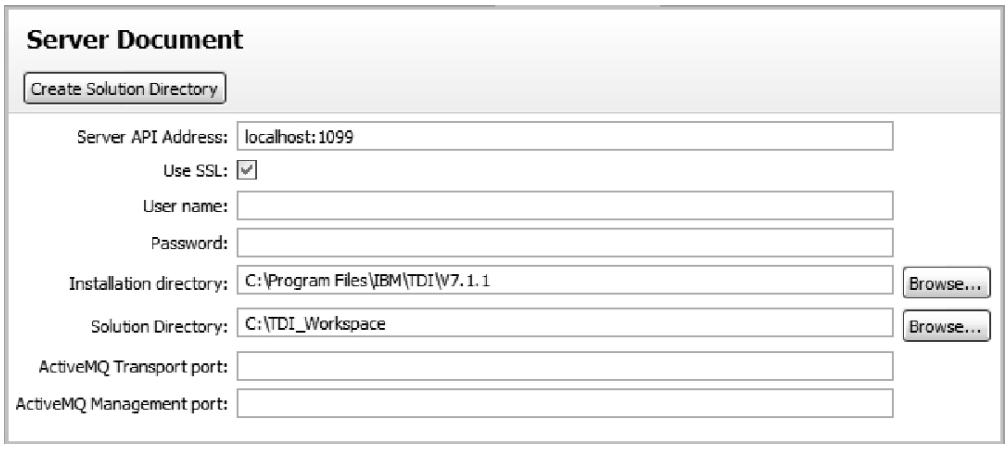


図 49. サーバー文書エディター

サーバー API アドレスは、IBM Security Directory Integrator サーバーの `host:port` アドレスです。インストール・ディレクトリーを指定すると、CE はサーバーを始動できます。新しい IBM Security Directory Integrator サーバーを始動する前に、すべてのパラメーターを構成し、「**ソリューション・ディレクトリーの作成**」ボタンを使用してサーバーに必要なファイルを作成します。Apache ActiveMQ のトランスポートおよび管理用に固有のポートを指定することもできます。

スキーマ・エディター

スキーマ・エディターでは、設計スキーマ・ファイルを管理します。

設計スキーマ・ファイルは、その他のエディターの入出力マップで使用されます。スキーマは、設計時間スキーマのみであり (つまり、CE 内でのみ)、一般的に、ランタイム構成ファイルに出現させたくない巨大なスキーマがある場合に使用されます。

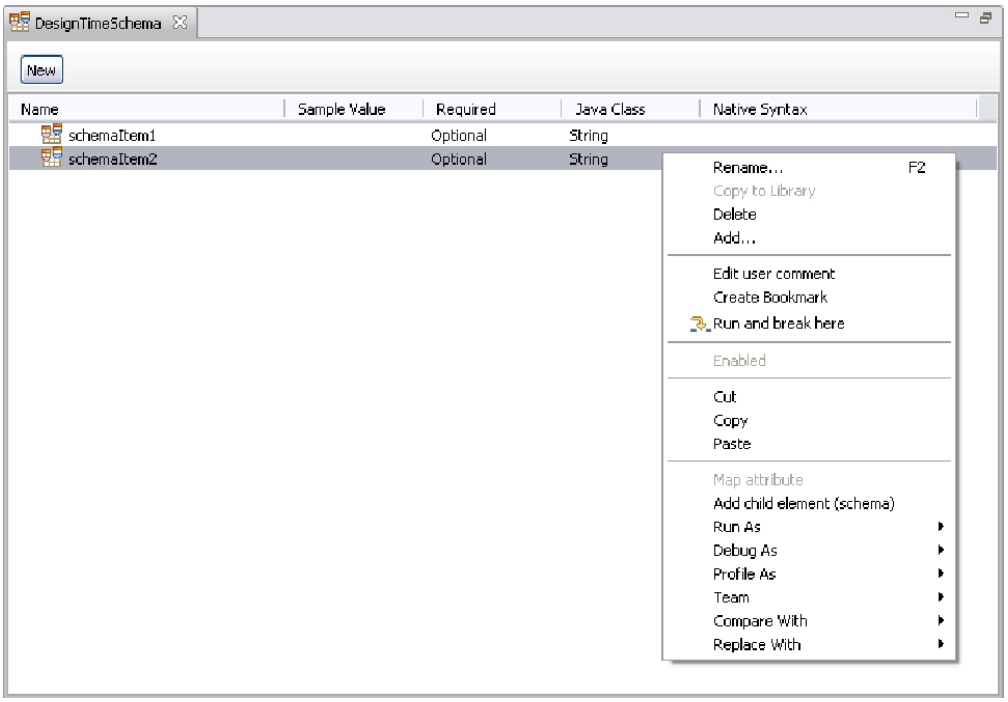


図 50. スキーマ・エディター

スキーマ・エディターには、最上位のスキーマ項目を新しく追加するための「**新規**」ボタンと、既存のスキーマ項目上で操作するコンテキスト・メニューがあります。

データ・ブラウザー

データ・ブラウザーは、ターゲット・システムを詳細に調べるための機能です。現在は、コネクターの詳細を提供する LDAP コネクターと JDBC コネクターがあるだけです。データ・ブラウザーを開くには、ライブラリーまたは AssemblyLine でコネクターを右クリックします。

ナビゲーターで右クリックして「**データのブラウズ**」を選択すると、新規エディター・ウィンドウが開き、コネクターでセットアップした現行接続内のデータを参照できます。

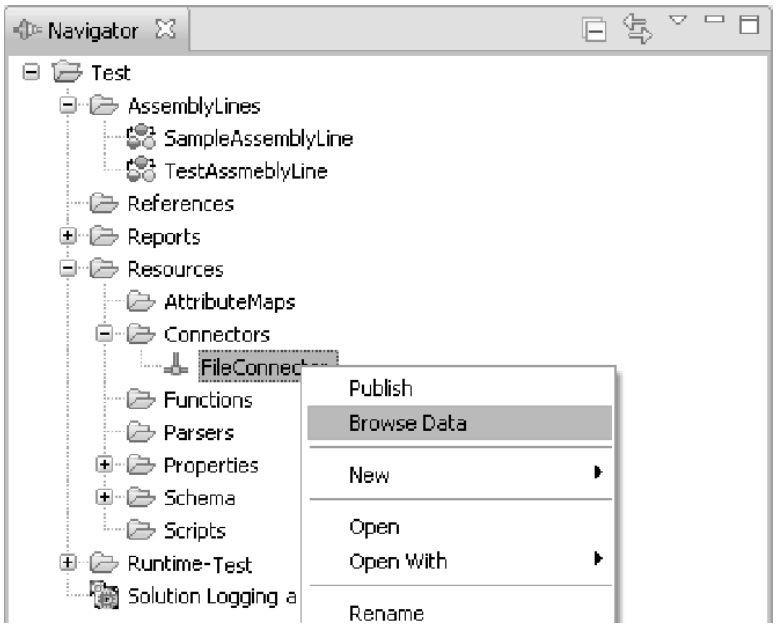


図 51. データ・ブラウザー

AssemblyLine でも同じようにして、データ・ブラウザーで新規ウィンドウを開くことができます。

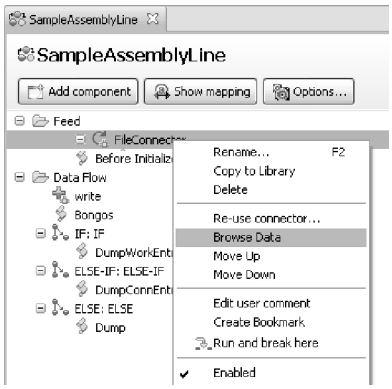


図 52. データ・ブラウザー

汎用データ・ブラウザー

これは、構成エディターに明示的な知識がないコネクターで使用されるデータ・ブラウザーです。これによって、データ・ソースからの結果セットを簡単にブラウズすることができます。

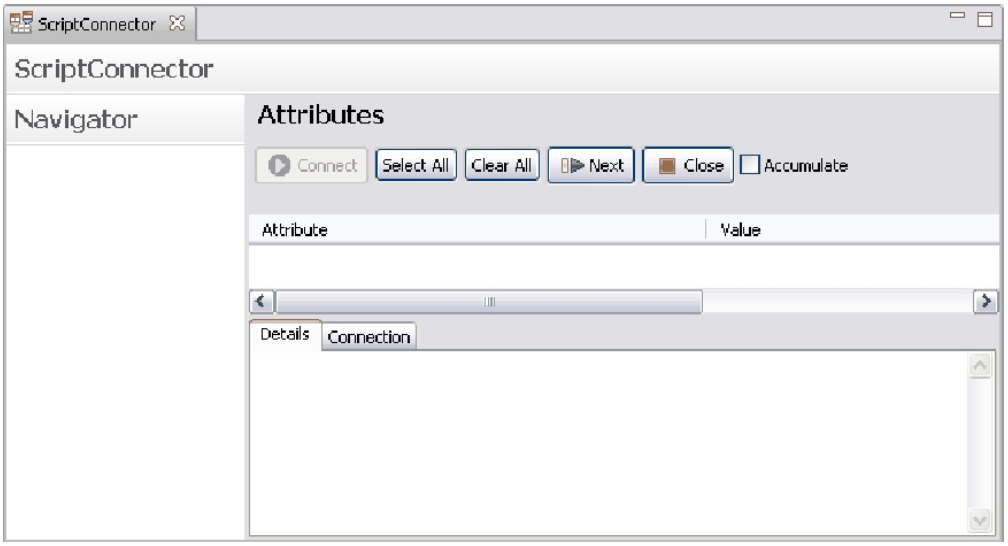


図 53. 汎用データ・ブラウザー

上部には、コネクターから読み取った属性のリストとツールバーが表示されます。リストの属性にはチェック・ボックスがあります。属性にチェック・マークを付けると、単純な `conn.attributename ->work.attributename` 式を使用して、その属性に属性マッピングが作成されます。属性のチェック・マークを外すと、その属性の属性マッピングは除去されます。

ツールバーには以下の機能があります。

表 8. データ・ブラウザーのツールバー

すべて切り替え	このコマンドで、リスト内のすべての属性のチェック・ボックスを切り替えます。これを実行すると、リストにあるすべての属性の属性マップが変更されます。
累算	このコマンドで、ディスカバーされた属性のリストを累算するかしないかを切り替えます。属性を累算すると、コネクターから読み取られるすべてのレコードが、既存の属性リストとマージされます。累算しない場合は、すべての属性を除去してから、次のレコードがリストに表示されます。
	接続を閉じるには、このボタンをクリックします。ブラウザーのエディター・ウィンドウを閉じると、接続も自動的に閉じられます。このエディターで接続設定を変更している場合は、通常、接続を閉じてから次のレコードを読み取ります。
	コネクターから次のレコードを読み取るには、このボタンをクリックします。コネクターから戻されるレコードがなくなった場合は、ツールバーの左側に、コネクターからの項目がなくなったことを示すメッセージが表示されます。この状態になったときにもう一度このボタンを押すと、コネクターは、その結果セットの先頭から読み取りを開始します。

画面の下部には 2 つのタブがあります。最初のタブは「**詳細**」タブで、ここには現在の選択の詳細が表示されます。汎用データ・ブラウザーでは、このタブは常に空です。

2 番目のタブは「**接続**」タブです。このタブには、コネクターの接続構成が表示されます。コネクター・エディターを開いたときと同様に、接続パラメーターを変更して保存することができます。

ストリーム・データ・ブラウザー

コネクターがパーサーを使用する場合に、ストリーム・データ・ブラウザーが使用されます。このストリーム・ベースのデータ・ブラウザーでは、最初にコネクターが初期化され、入力ストリームの取得が試行されます。その後、詳細タブに入力データの最初の 20K が表示されます。

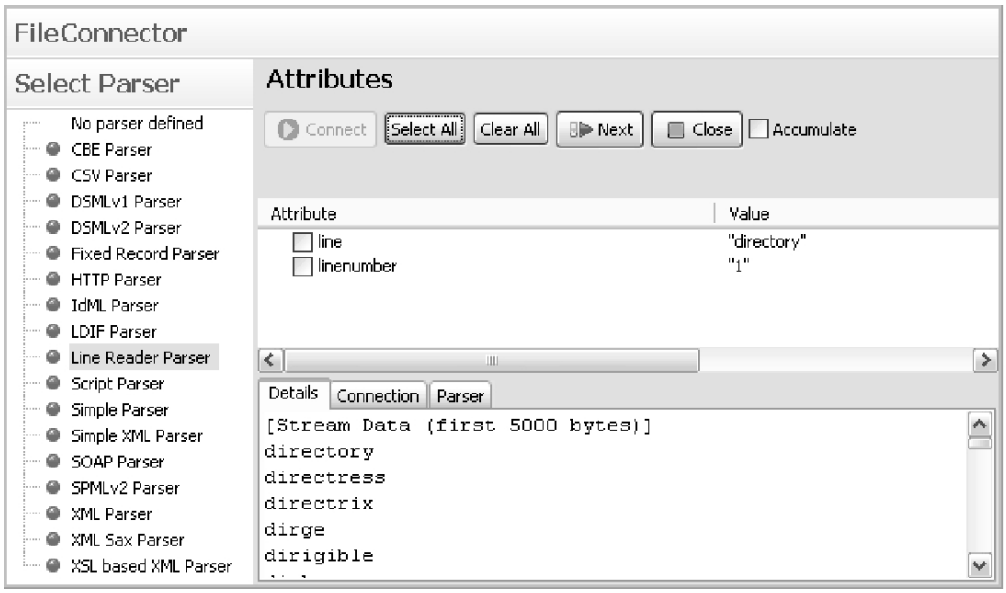


図 54. ストリーム・データ・ブラウザー

入力ストリームの内容を読み取り、入力ストリームが予期している内容であるかどうかを確認するため、左側の「**パーサーの選択**」リストからパーサーを選択できます。パーサーを選択すると、「**パーサー**」タブの内容が更新され、選択されたパーサーの構成フォームが表示されます。パーサーを変更するたびにコネクターがクローズされるため、パーサーの選択と read-next を連続して実行することで、パーサーが入力を解釈できるかどうかを容易に確認できます。

注: テーブルでパーサーを選択するときには、そのパーサーを使用するようにコネクター構成も変更します。クローズして保存する (または「**ファイル**」 > 「**保存**」機能を使用する) と、設定したパラメーターで構成が実質的に更新されます。

JDBC データ・ブラウザー

JDBC データ・ブラウザーには、左側にすべてのテーブルとビューが表示されます。「詳細」タブには、そのリストで選択した項目の情報が表示されます。

図 55. JDBC データ・ブラウザー

「**詳細**」タブには、JDBC 接続オブジェクトから取得したシステム情報が表示されます。左方にあるツリー表示にも、すべてのテーブルとビューが表示され、それらの列が子項目として表示されます。テーブルまたはビューを選択すると、「詳細」タブにテーブル全体の構文が取り込まれます。例えば、「IDI_PS_DEFAULT」というテーブルを選択すると、次のようになります。

図 56. JDBC テーブルの詳細

テーブルまたはビューの列を選択すると、その列の詳細のみが表示されます。

図 57. 「テーブルとして使用」オプション

テーブル名を右クリックして、「**テーブルとして使用**」機能を使用し、 JDBC コネクター構成の **表名** パラメーターを更新することもできます。

「**JDBC テーブル**」ヘッダーのツールバーの ボタンを使用すると、接続を再ディスカバーできます。これを使用する必要があるのは、最初のディスカバリーが失敗した場合、または「接続」タブで JDBC URL を変更した場合だけです。

LDAP データ・ブラウザー

LDAP データ・ブラウザーには、LDAP サーバーが提供するスキーマとコンテキスト接頭部 (検索ベース) が表示されます。

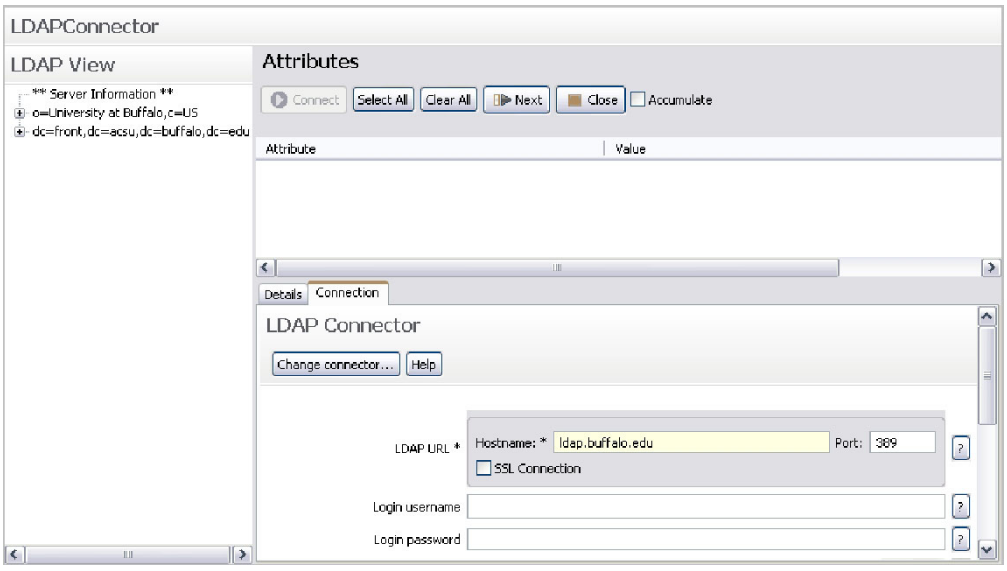


図 58. LDAP データ・ブラウザー

「LDAP ビュー」には、LDAP サーバーが提供するサーバー情報と検索ベースの両方が含まれています。表示される結果は、このツリーで選択した項目によって異なります。スキーマ以外のいずれか 1 つのノードを選択すると、以下のように、その特定項目の詳細なダンプが「詳細」タブに表示されるはずです。

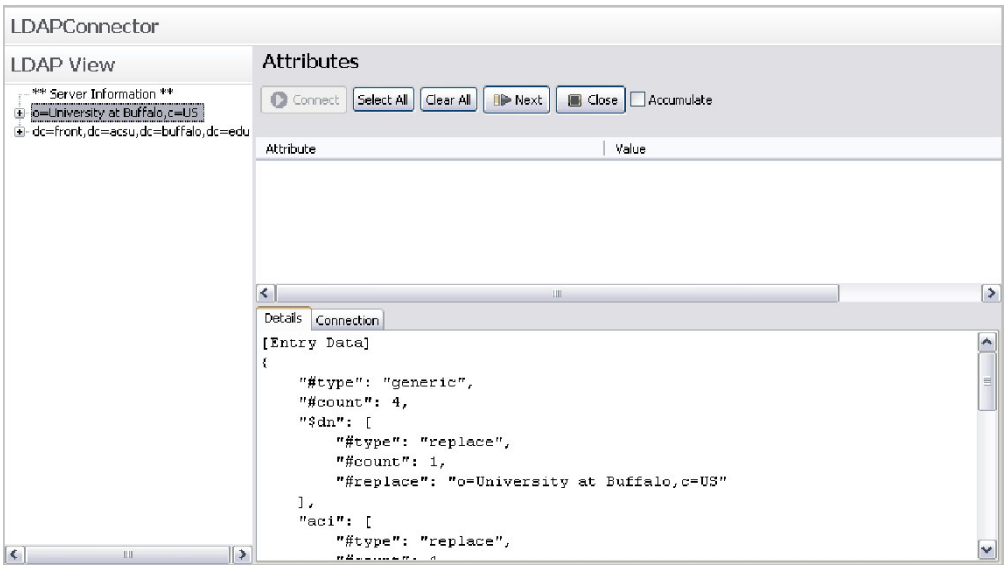


図 59. LDAP データ・ブラウザー項目

スキーマ項目を選択すると、「詳細」タブに詳細が表示されるとともに、ご使用の属性リストがスキーマ項目からの情報で更新されます。これは、特定のスキーマの読み取りや書き込みを行う場合に、大いに役立ちます。

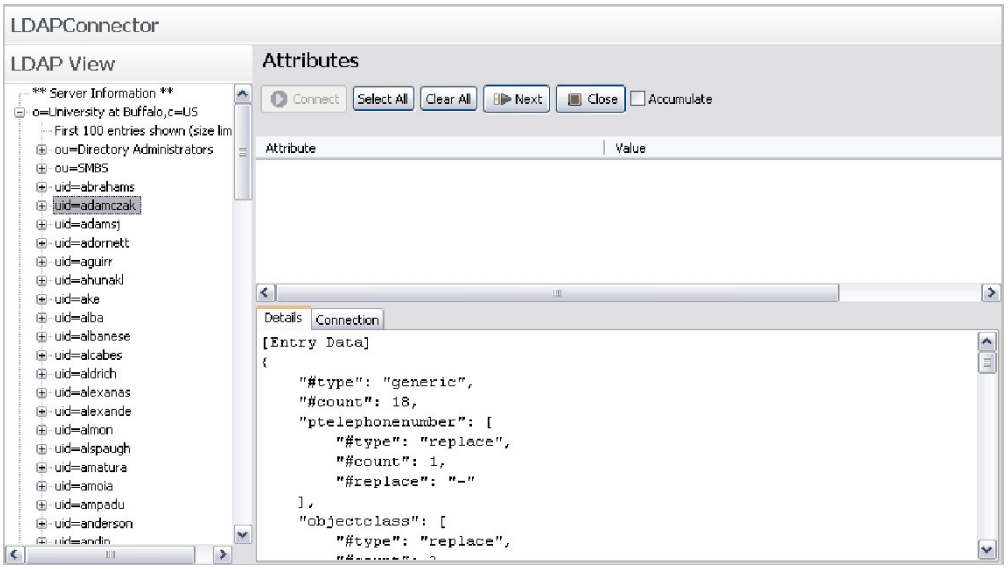


図 60. LDAP データ・ブラウザー・スキーマ項目

スキーマ・ノード内のオブジェクト・クラスを選択すると、そのクラスの属性マッピングを即座に作成できます。値の列には現在、その属性に関する情報が含まれています。値「MAY」はオプション、「MUST」は必須であることをそれぞれ示しています (項目を追加する場合)。括弧内の値は、属性を定義するオブジェクト・クラスを示しています (LDAP オブジェクト・クラスは階層になっています)。

コンテキスト・メニューから「**検索ベースとして使用**」を選択すると、LDAP コネクターの **検索ベース** パラメーターを即座に更新することもできます。

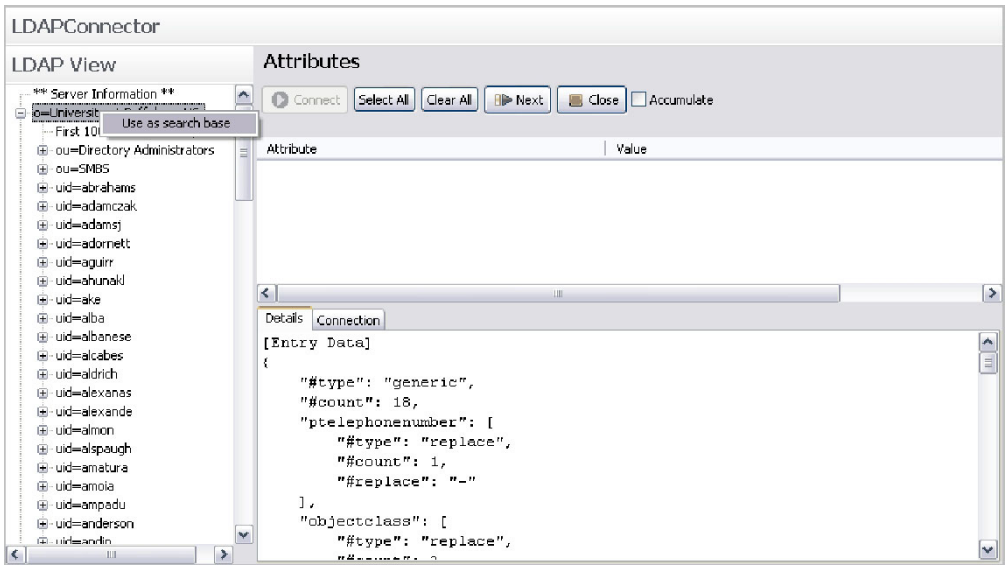


図 61. 「検索ベースとして使用」コンテキスト・メニューの選択

フォーム・エディター

フォーム・エディターは、コンポーネントの接続パラメーター・フォームをカスタマイズするのに使用します。これは、リソース・フォルダー内のコンポーネントにのみ適用されます (プロパティー・ファイルを除く)。

フォームをカスタマイズするには、フォーム・エディター でコンポーネントを開く必要があります。

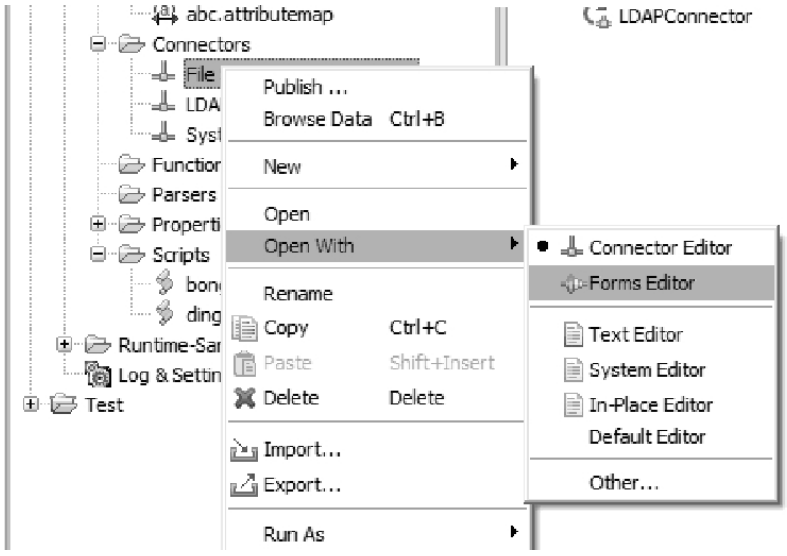
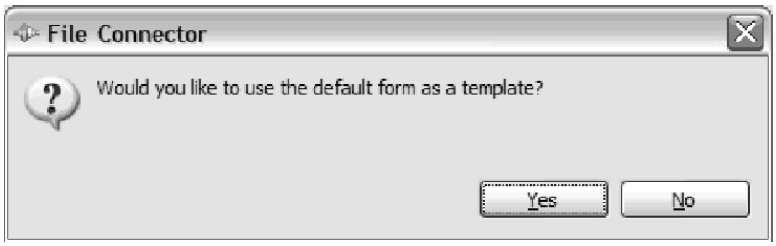


図 62. コンテキスト・メニュー - フォーム・エディターの選択

あるファイルにデフォルト以外のエディターを選択した場合、CE では、選択されたエディターが記憶されているため、次回にそのファイルをダブルクリックすると、前回使用したエディターでファイルが開かれることに注意してください。デフォル

ト・エディターでコンポーネントを開くには、このメニューで該当するエディター (通常は一番上のエディター) を選択するだけです。

開くコンポーネントにカスタム・フォームがない場合は、以下のように、フォームにデフォルト・フォームを取り込むよう、プロンプトが出されます。



「**はい**」を選択すると、コンポーネントのデフォルト・フォームに基づいて、初期フォーム定義が作成されます。この場合の例では FileSystem コネクターが使用され、次のような画面が表示されます。

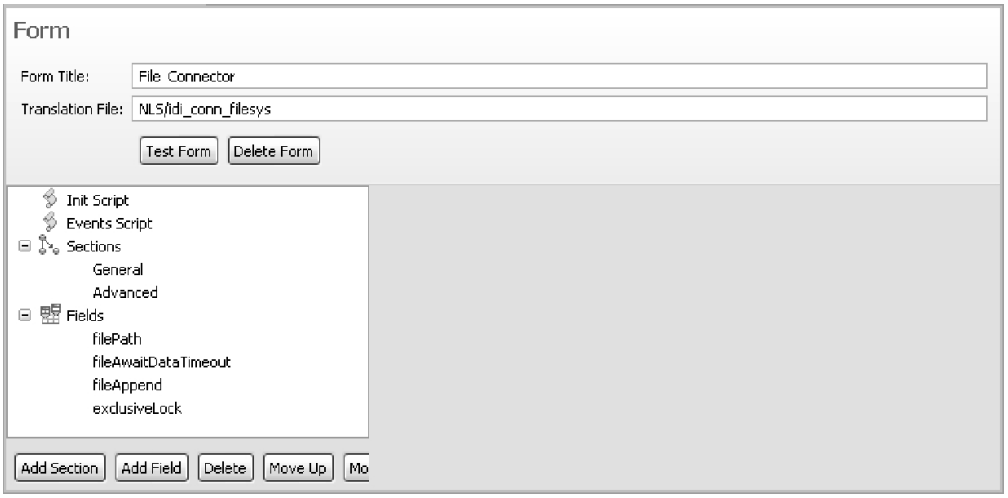


図 63. FileSystem コネクターのデフォルトのフォーム・エディター画面

このフォームに表示されるさまざまなエレメントには、以下の機能があります。

フォーム・タイトル
コンポーネント・フォームのメイン・タイトルになります (タイトルがない場合は、ブランクのままにします)。カスタム・フォームの先頭に、これが表示されます。

変換ファイル
フォーム内のラベルとツールチップの変換に使用するファイルです。変換ファイルを定義する際に、すべてのラベルとツールチップの変換が試行されます。「file_name」というラベルをテキストとして作成すると、変換が試行され、「file_name」がキーとして使用されて変換ファイルからストリングが検索されます。変換ファイル自体は、各行に「key=value」があるシンプル・プロパティー・ファイルです。

フォームをローカライズするには、ロケール ID の付いたファイルを作成する必要があります。したがって、フランス語の変換の場合は、*base-name*_fr.properties というファイルを作成します。

フォームのテスト

このボタンを押すと、現行のフォーム定義が指定されたダイアログ・ウィンドウが表示されます。

フォームの削除

このボタンを押すと、コンポーネントからフォームが削除されます。これを永続的に有効にするには、クローズしてから「**保存**」を選択する必要があります。

初期化スクリプト

初期化スクリプトは、フォームのロード時に実行されます。ここには、フォームの状態を初期化するコードと、フォームのグローバル・スクリプト変数を入れます。

イベント・スクリプト

フィールド値が変更されると、フォームによって、このスクリプトに定義されたイベント・ハンドラーが実行されます。すべてのフィールドには、コンポーネントが使用する内部名が付いています。例えば、FileConnector は「filePath」を「**ファイル・パス**」パラメーターの内部名として使用します。ご使用のフォームに「フィールド」セクション内のデフォルト・フォームを取り込むようにすると、すべてのコンポーネント・パラメーター名を表示できます。フォーム内の変更に対応するには、「_changed」という接尾部の付いた内部名をスクリプト関数名として使用して、イベント・スクリプト・エディターでイベント・ハンドラーを作成します。以下に、認証メソッドに基づいて 2 つのフィールドを使用不可にする LDAP コネクターの例を示します。

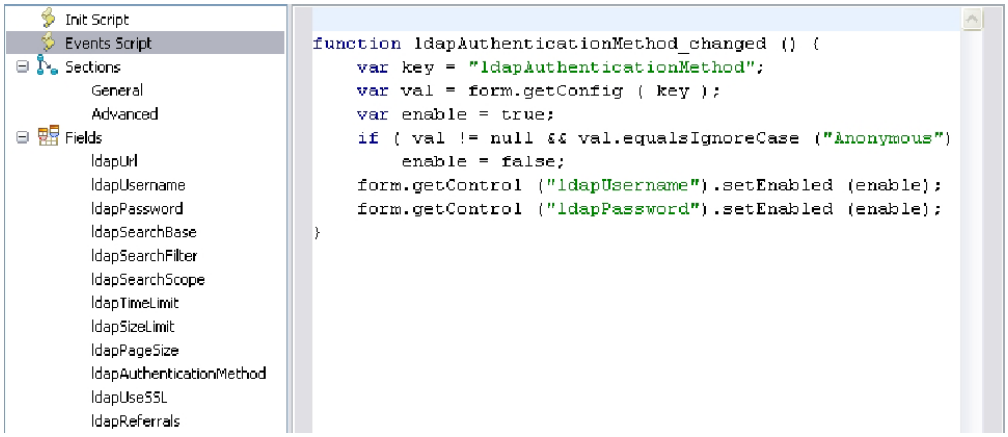


図 64. フォーム・エディター、LDAP コネクターのイベント・スクリプト

セクションとフィールド

セクションとフィールドは、フォームを構成するものです。セクションとフィールドは、ツリーの下にあるツールバーを使用して、追加、削除、および再度並べ替えることで管理します。

- 「**セクションの追加**」 – フォームに新規の空のセクションを追加します。
- 「**フィールドの追加**」 – フォームに新規フィールドを追加します。フィールド名は固有でなければなりません。
- 「**削除**」 – フォームからセクションやフィールドを除去します。
- 「**上へ移動/下へ移動**」 – セクションとフォームを再度並び替えます。セクションが定義されている場合、フィールドの順序は、フィールドのリストではなく、セクションによって定義されます。セクションが定義されていない場合は、このツリー内の順序によってフォーム内のフィールドの順序が決まります。

セクション

この部分はオプションです。セクションを指定しない場合、フォームにはそのフォーム内のすべてのフィールドが一度に表示されます。

セクションを使用して、フォーム内にフィールドを配置します。セクションは、フォルダーと同様に、展開または縮小表示して、内容を表示または非表示にすることができます。セクションを定義する際は、最初にセクションを展開した状態にするかどうか、その中にどのフィールドを表示するかを指定します。 FileSystem コネクターの例では、2 つのセクションがあります。

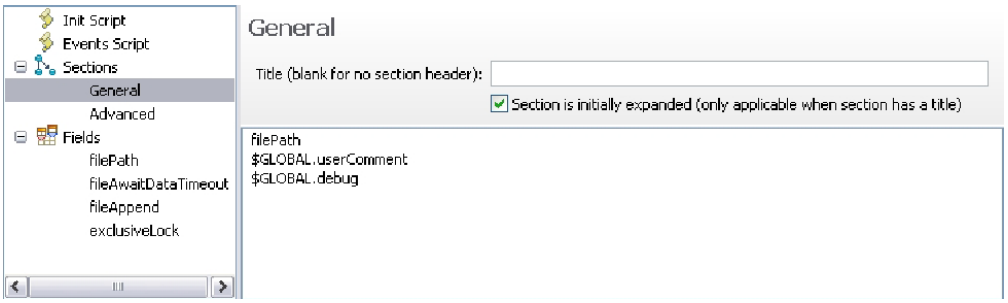
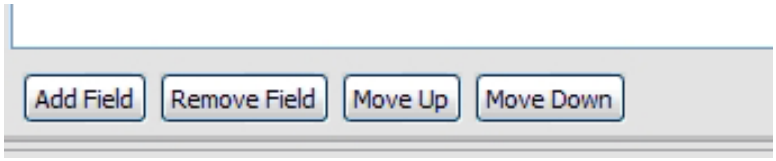


図 65. フォーム・エディター - 「General」セクション

最初のセクションは、「*General*」セクションです。このセクションにはタイトルがありません。つまり、クリックして内容を展開、縮小表示できるセクション・ヘッダーがないことになります。縮小表示や展開が不可能になるため、セクションは静的セクションとなります。フィールドのリスト内のフィールドは、パネル下部にあるツールバーを使用して追加、除去、再配列が可能です。



2 番目のセクションは、「*Advanced*」セクションです。

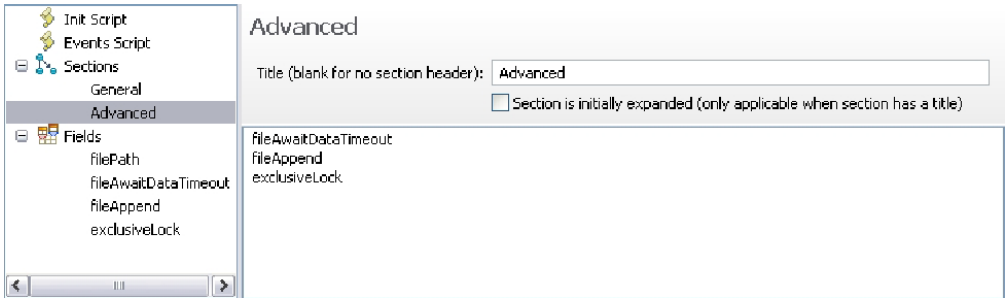


図 66. フォーム・エディター - 「Advanced」セクション

このセクションにはタイトルがありますが、最初は展開されていません。これにより、フォームは表示されますが、このセクション・タイトルは縮小表示されています。また、その中のフィールドは、ユーザーがセクションを展開するまで非表示になります。

フィールド

フィールドは、コンポーネントのパラメーターです。 FileSystem コネクターと同様にコンポーネントを再利用する場合は、フォームに必須パラメーターを指定するか、「初期化スクリプト」セクションにパラメーターを設定して、コンポーネントが正しく機能するようにする必要があります。各フィールドを選択すると、その定義が表示されます。

図 67. フォーム・エディター - フィールド定義

このパネルには、コネクター・パラメーター「filePath」の定義が表示されます。これは、次の順序で表示されます。

表 9. フォーム・エディター - パラメーター定義

フィールド	説明
ラベル	フォームに表示されるラベル

表 9. フォーム・エディター - パラメーター定義 (続き)

フィールド	説明
ツールチップ	入力フィールドにマウス・ポインターを移動したときに表示されるツールチップ
フィールド・タイプ	入力フィールドのタイプは、以下のとおりです。 • ストリング (単一行テキスト入力用) • ドロップダウン (編集可能) (編集可能なドロップダウン入力フィールド用) • ドロップダウン (編集不能) (固定された一連の選択項目があるドロップダウン用) • ブール値 (チェック・ボックス用) • テキスト域 (複数行テキスト入力フィールド用) • 静的テキスト (単純なテキスト表示用 (つまり、入力なし)) • パスワード (単一行のパスワード保護入力フィールド用) • スクリプト・エディター (スクリプト編集用) • カスタム・コンポーネント (ユーザー定義の SWT/JFace 制御用)
モードの選択	このオプション・フィールドには、フィールドの除外や組み込みを行う際のコンポーネントのモードを指定できます。除外する場合は、負符号 (-) を付け、コンマで区切ってモードを指定します。 「Iterator」- イテレーター・モードでのみ表示します 「-Iterator」- イテレーター・モードでは表示しません 「Iterator,Lookup」- イテレーター・モードと ルックアップ・モードでのみ表示します

下部にある 3 つのタブでは、ボタン、ドロップダウン・リストのドロップダウン値、およびカスタム・コンポーネントの Java クラス名を指定できます。

ウィザード

IBM Security Directory Integrator 構成エディターにはさまざまなウィザード (グラフィカルな操作支援ステップ方式の手順) があります。これらの症状を以下に示します。

1. 『構成のインポート・ウィザード』
2. 153 ページの『新規コンポーネント・ウィザード』
3. 159 ページの『「コネクター構成」フォームの特徴』

構成のインポート・ウィザード

構成のインポート・ウィザードを使用して、以前のバージョンの構成ファイルをインポートできます。

このウィザードでは、ターゲット・プロジェクトとインポートするコンポーネントを選択します。

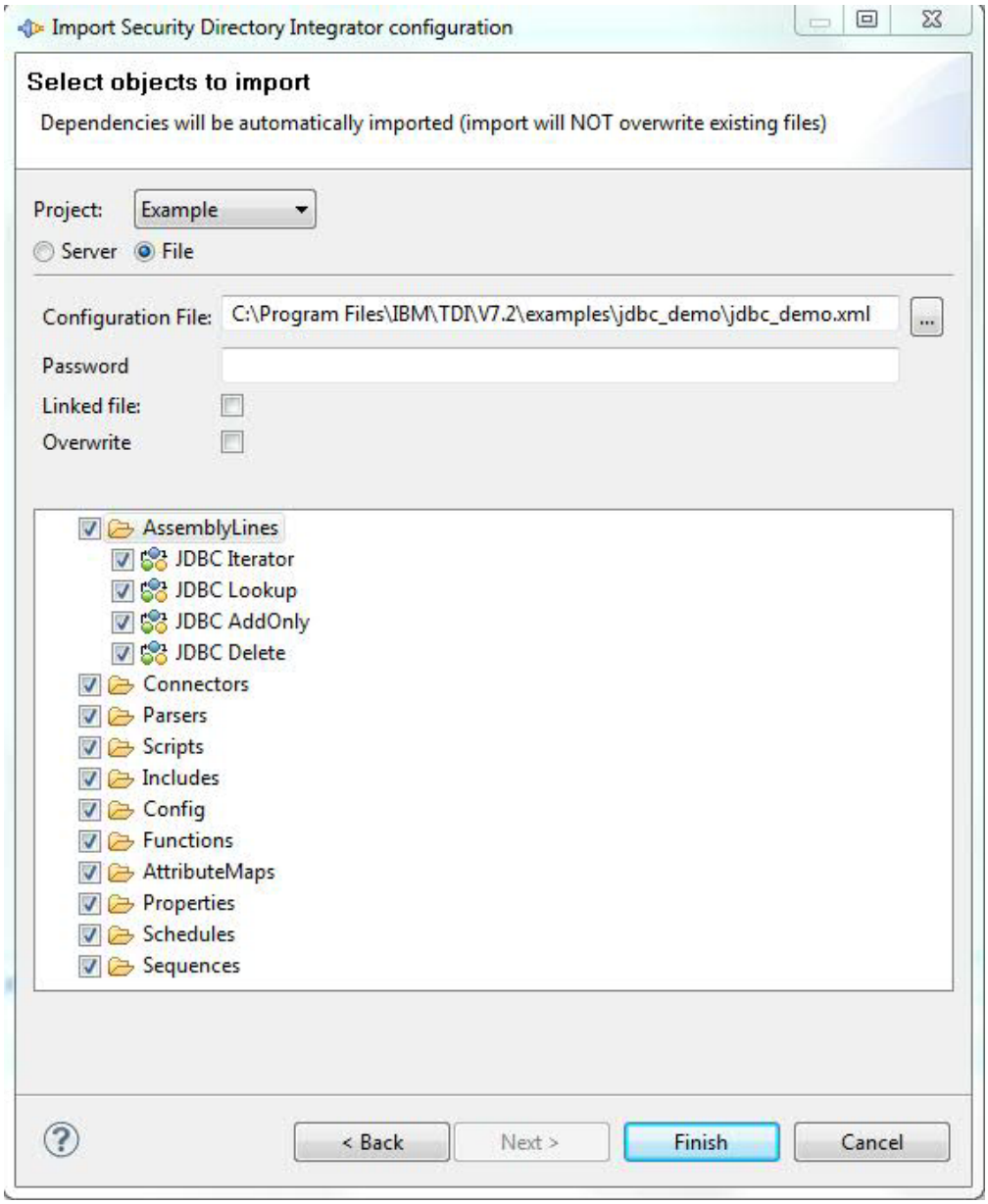


図 68. 構成のインポート・ウィザード

デフォルトでは、インポート対象としてすべてのコンポーネントが選択されています。ただし、インポートするコンポーネントのみを選択できます。構成ファイルのコネクターを使用する 1 つの AssemblyLine を選択すると、これらのコネクターもインポートされます。

「**プロジェクト**」入力フィールドでは、構成のインポート先のターゲット・プロジェクトを指定します。新規プロジェクトを作成するには、空白オプションを選択します。

「**構成ファイル**」入力フィールドには、インポートする構成ファイルを指定します。構成がパスワードで保護されている場合は、「**パスワード**」入力フィールドにパスワードを入力します。

「**リンクしたファイル**」フィールドにチェック・マークを付けると、インポートされたプロジェクトに対して行われた変更がすべて、プロジェクトのインポート元のファイルに書き戻されます。この設定を変更するには、次の図に示すように、プロジェクトのプロパティーで「**リンクしたファイル**」フィールドのファイル名を消去または変更します。

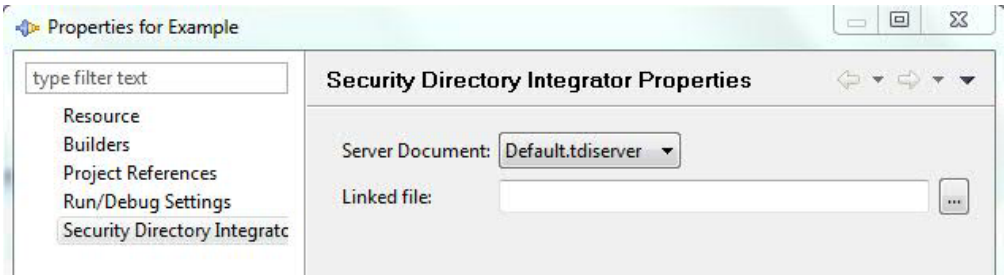


図 69. 「リンクしたファイル」フィールド

サーバーから構成をインポートすることもできます。「**サーバー**」ラジオ・ボタンにチェック・マークを付けて、サーバー・ビューに切り替えます。

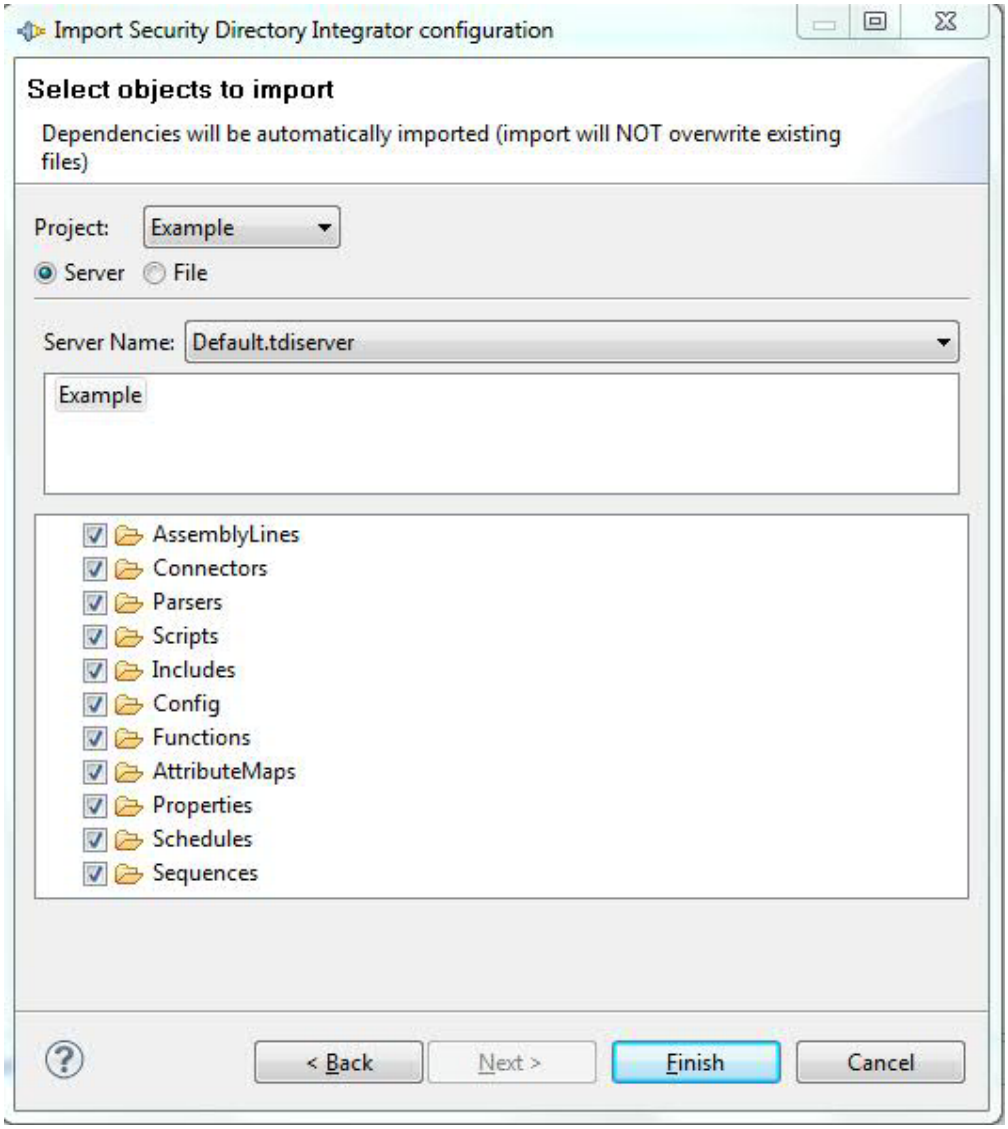


図 70. 「サーバーからのインポート」ウィザード

サーバーからのインポートを行うには、まず「**サーバー名**」フィールドのサーバー・リストからサーバーを選択します。サーバーを選択すると、そのサーバーにある構成のリストがサーバー名の下のリストに表示されます。これはネットワーク経由での操作であるため、リストが最新表示されている間に一部のコントロールが使用不可になる場合があります。構成のリストから、インポートする構成を選択します。構成を選択すると、サーバーから構成がダウンロードされてその下のツリーに内容が表示され、インポートに含めるコンポーネントを選択できるようになります。

新規コンポーネント・ウィザード

AssemblyLine で「**コンポーネントの追加**」を使用するか、またはメインメニュー・バーから「**ファイル**」 > 「**新規 ...**」を選択すると、このウィザードが起動されます。

Resources フォルダーに新規コンポーネントを作成する場合、ウィザードのレイアウトが多少異なります。例えば新規コネクターを作成する場合のウィザードは次のようになります。

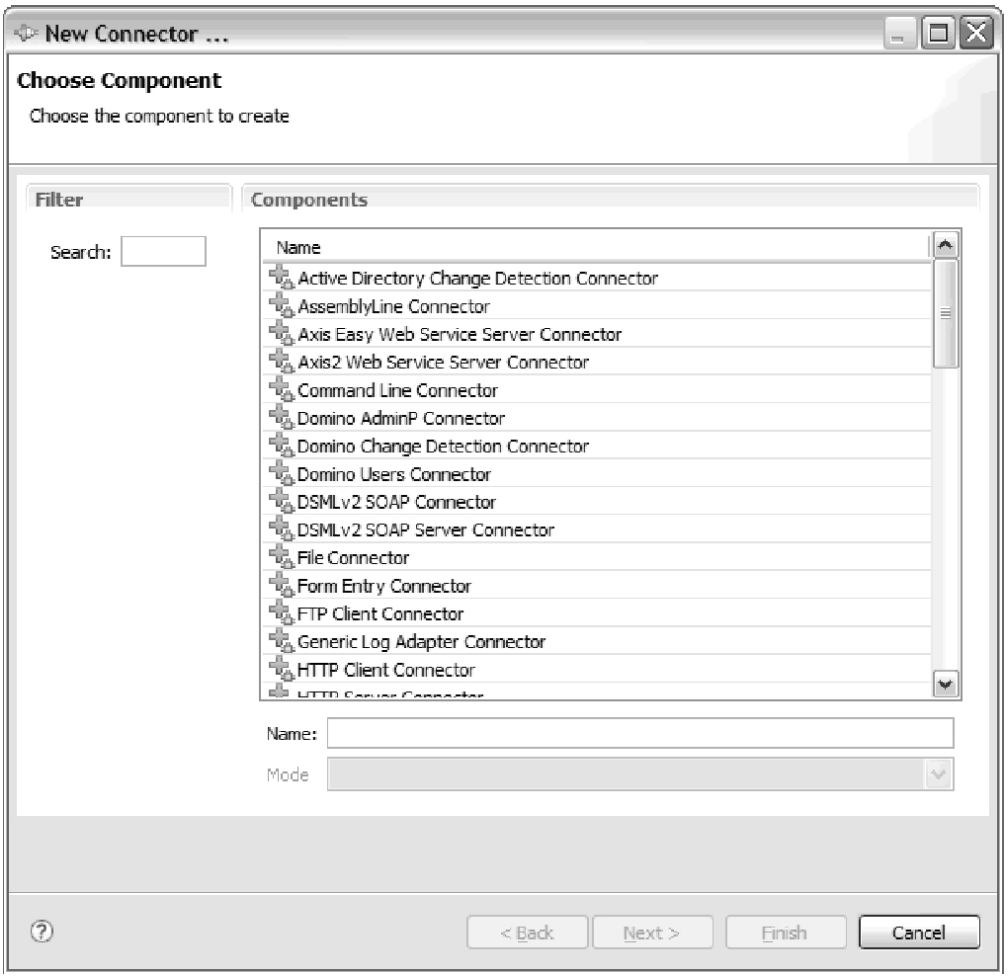


図 71. Resource フォルダーに新規コネクターを作成する場合のウィザード

コンポーネント名は、フォルダーで提示されるファイル名として使用されます。この名前をわかりやすい名前に変更することをお勧めします。

AssemblyLine に新規コンポーネントを作成する場合、このウィザードにはこれよりも多くのオプションが表示されます。

ウィザードの最初のページは、コンポーネント・タイプ選択ページです。このページでは、追加または作成するコンポーネントのタイプを選択します。左側にはフィルターのリストが表示されます。これらのフィルターは、リストのラベルに基づいて関連するコンポーネントを選択します。

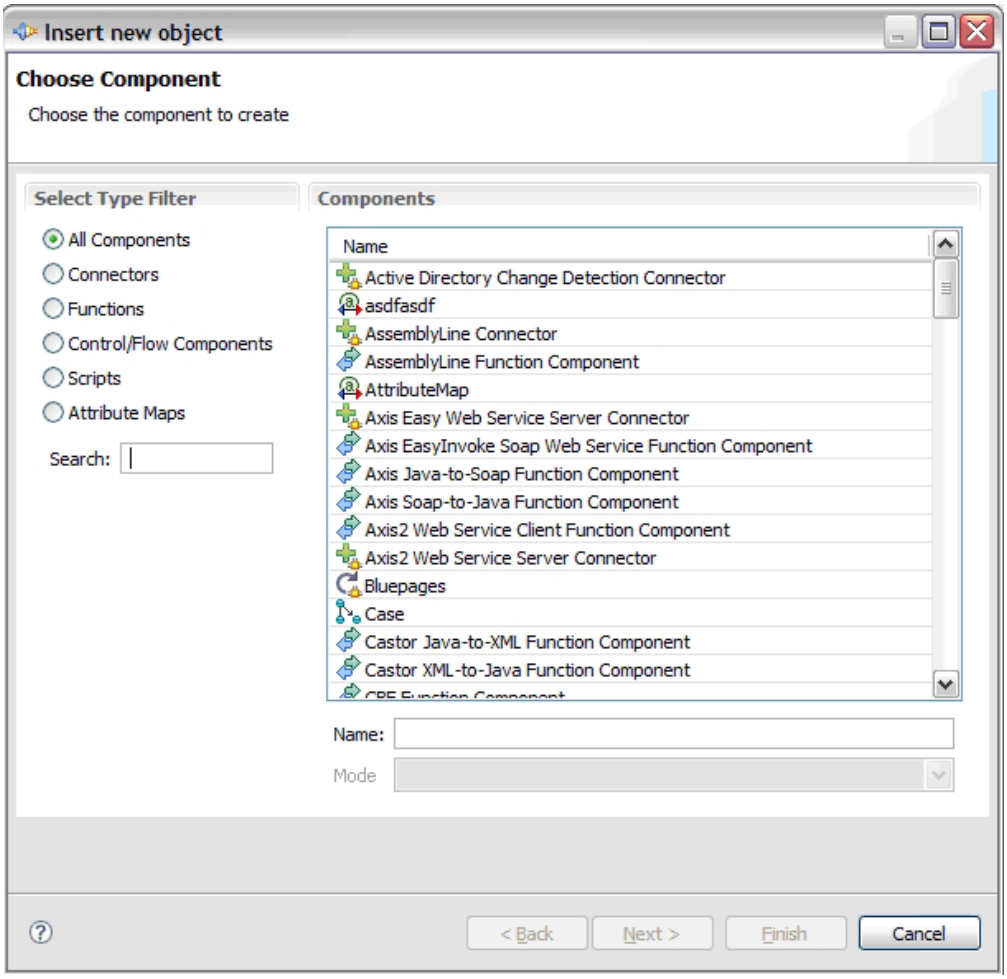


図 72. 新規コンポーネント・ウィザード

コンテンツをフィルタリングするには、「検索」フィールドに値を入力します。文字を入力すると、検索フィールドに入力した文字に一致するものがリストに示されます (大/小文字が区別されない部分一致)。

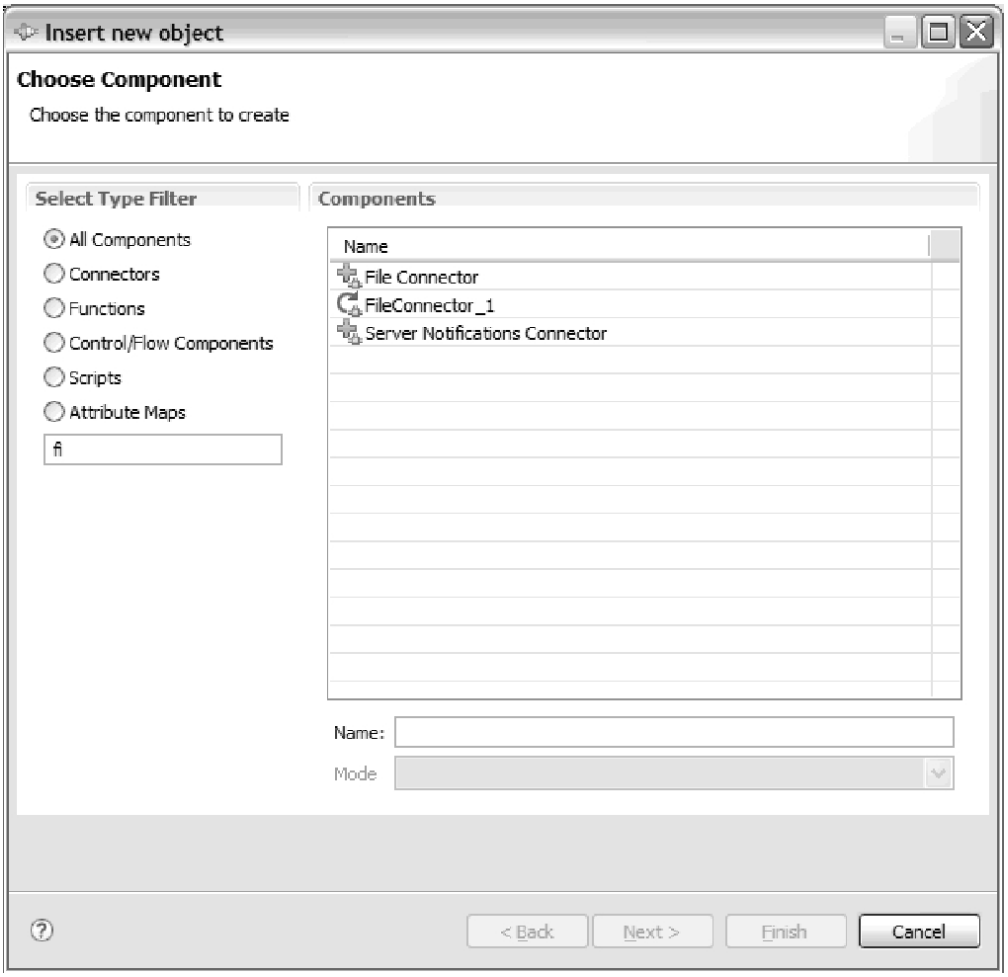


図 73. フィルタリングされた新規コンポーネント・ウィザード

この時点で、ウィザードを完了することができます。ウィザードを完了すると、コンポーネントが AssemblyLine に挿入されます。

コネクターを選択すると、コネクターを適切に定義するための一連のフォームが表示されます。その他のタイプの場合は、ウィザードを終了して、AssemblyLine でコンポーネントを構成します。

タイプを選択すると、「コネクター構成」パネルが表示されます。

図 74. 「コネクター構成」パネル

このタイプのフォームの入力に関する特別事項については、 159 ページの『「コネクター構成」フォームの特徴』も参照してください。

コネクターでパーサー構成を使用できる場合、次にパーサー構成が表示されます。

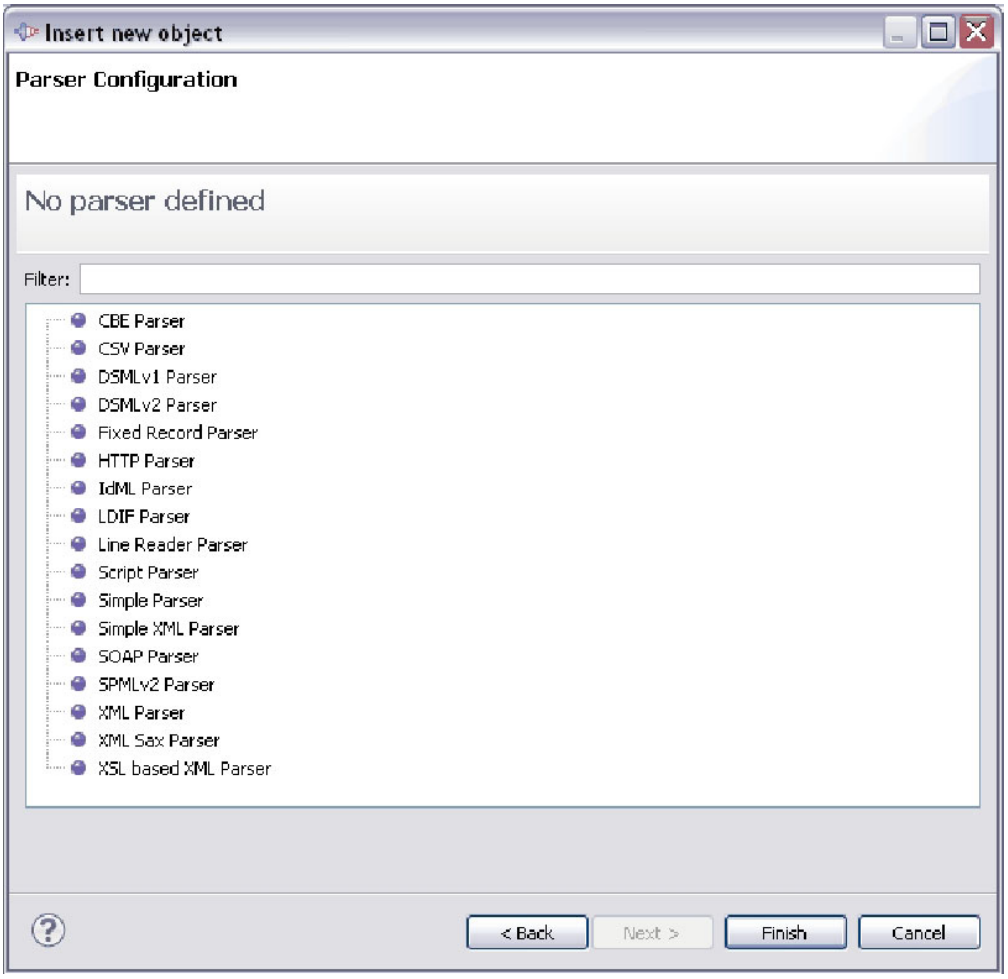


図 75. 「パーサー構成」パネル

「**パーサーの選択**」ボタンを使用して、コンポーネントのパーサーを選択します。

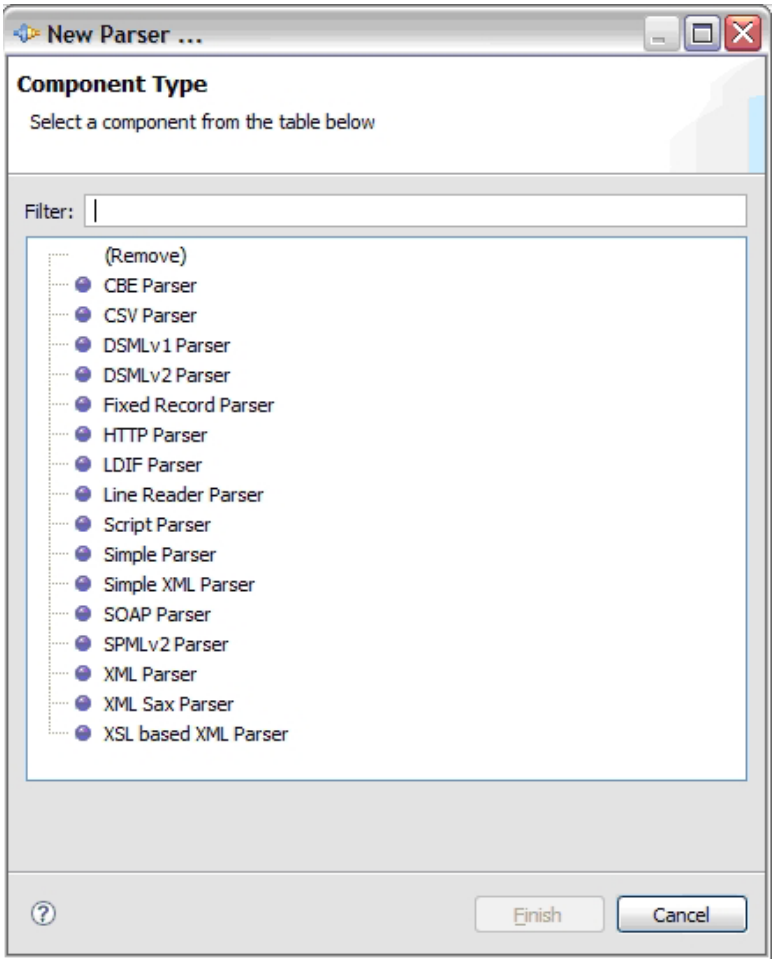


図 76. パーサー選択ダイアログ

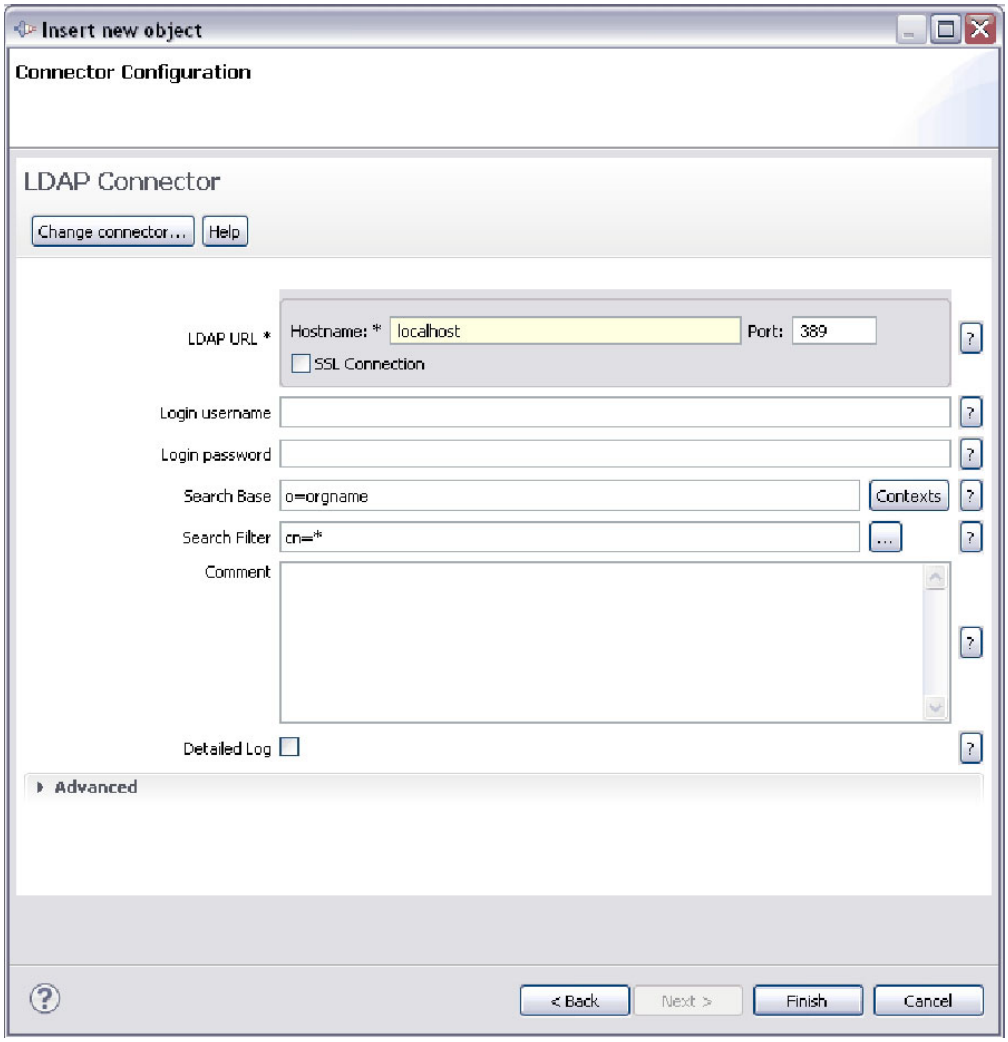
コンポーネントから現行のパーサーを除去するには、「(除去)」オプションを選択します。

「コネクター構成」フォームの特徴

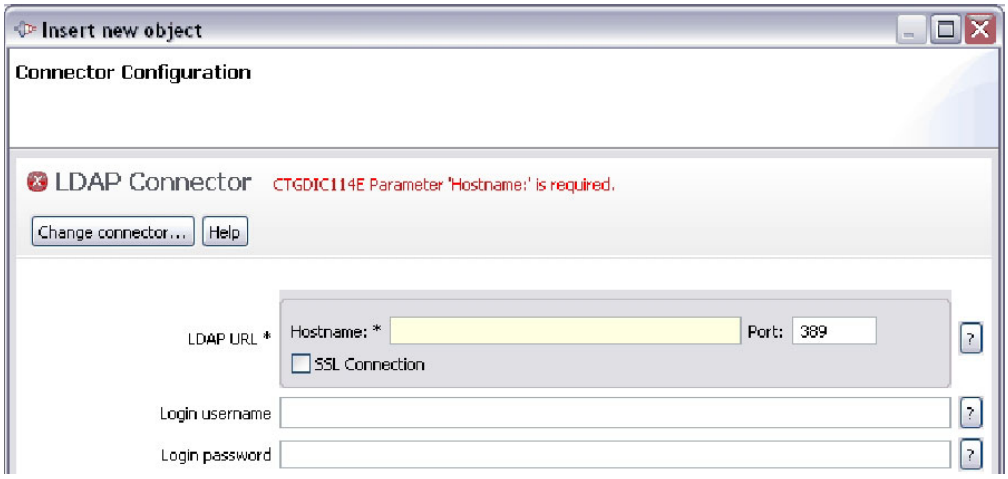
フォーム内の各フィールドを、必要に応じてタイトル付きのセクションに分類することができます。各フィールドは、そのフィールドが必須フィールドであるかどうかを示すプロパティーを持っています。

「コネクター構成」ウィンドウでは、必須フィールドにはフィールド名の後に * の印が付いています。

継承される値には、青色のラベルがあります。



上の例では、LDAP コネクターの構成に、2 つのセクションと 1 つの必須フィールドが示されています。必須フィールドに値が指定されていない場合、フォームのタイトルにこのことが示されます。



赤色のアイコンまたはテキストの上にマウス・ポインターを移動させると、エラー・メッセージと、エラーが当てはまるフィールドが表示されます。これは、フォーム内に複数のエラーがある場合に便利です。

AssemblyLine の実行とデバッグ

構成エディターには、AssemblyLine のロジックのテストおよびデバッグの機能など、AssemblyLine の開発を支援するさまざまなメカニズムが組み込まれています。

AssemblyLine レポート

AssemblyLine レポートは、ナビゲーターのコンテキスト・メニューから実行できます。

AssemblyLine を右クリックして、「**AssemblyLine レポートの作成**」サブメニューを選択します。このメニューには、*TDI_Install_dir*/XSLT/ConfigReport ディレクトリー内にあるすべてのレポート・テンプレートが表示されます。

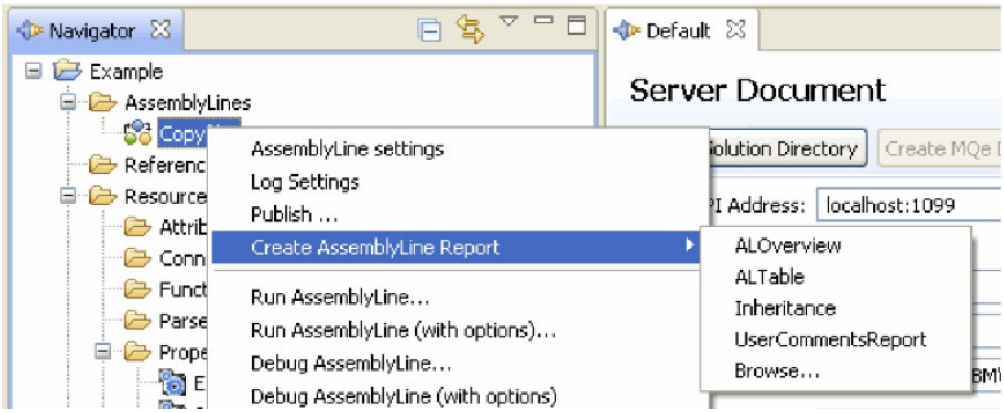


図 77. 「AssemblyLine レポートの作成」コマンド

「**ブラウズ...**」オプションを選択すると、レポート・テンプレートのローカル・ファイル・システムを参照することができます。

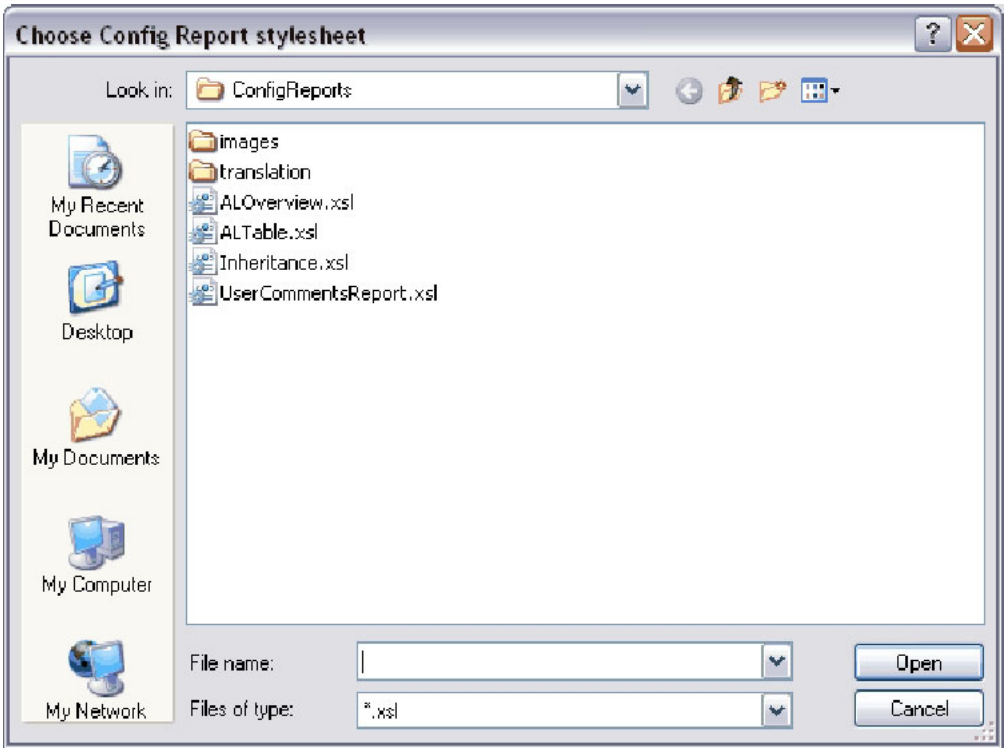


図 78. 「構成レポート・スタイルシートの選択」ダイアログ

1 つのファイルを選択し「**開く**」をクリックすると、レポートが生成されて、次の図に示すように、プロジェクトの「**レポート**」ディレクトリーに格納されます。「*.html*」のファイル拡張子に関連付けられているエディター (通常はデフォルトのインターネット・ブラウザー) が開いて、レポートが表示されます。

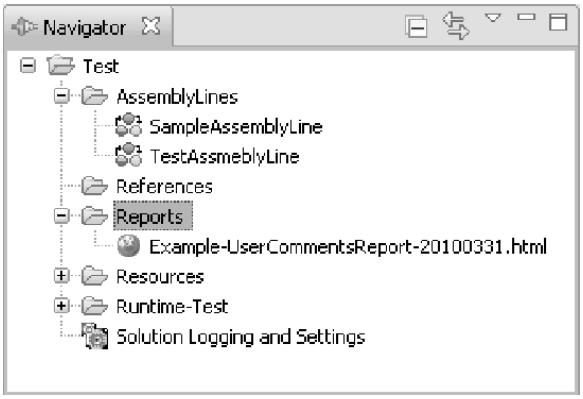
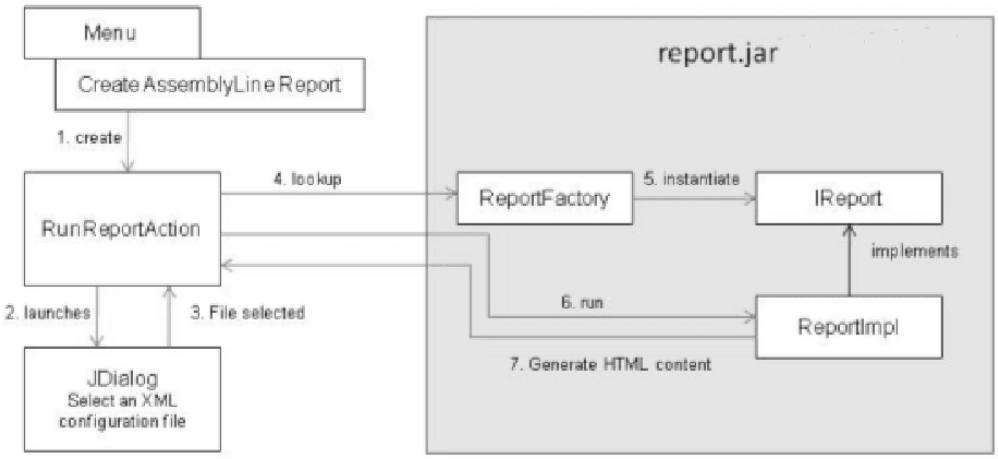


図 79. 「プロジェクト」階層内の「レポート」フォルダー

AssemblyLine レポートでは、AssemblyLine に基づいたレポート・ファイル名が生成され、その名前に現在の日付が挿入されます。

XML ベースの AssemblyLine レポートの概要

選択した構成エレメントのレポートを、レポート・スタイル・シートに基づいて、または指定したレポート構成 XML ファイルに対して、生成できます。以下の図に、XML ベースの AssemblyLine レポートのアーキテクチャーを示します。



レポート構成 XML ファイル・フォーマット

AssemblyLine レポート構成 XML ファイルのフォーマットは、以下のとおりです。

```
<tdiReport>
<reportClass)com.ibm.di.report.aloverview.AssemblyLineOverview</reportClass>
    <reportConfig>
        <report specific configuration>
    </reportConfig>
</ tdiReport>
```

構成 XML ファイルのエレメントは、以下のとおりです。

- **reportClass** - レポートの Java クラス名を指定します。
- **ReportFactory** - レポートのインスタンスをインスタンス化します。
- **reportConfig** - レポート固有のパラメーターを含みます。
-

AssemblyLine の実行

AssemblyLine を開発しながら、AssemblyLine をテストすることができます。これを行うには、AssemblyLine を最後まで実行するか、またはコンポーネントを 1 つずつステップ実行します。

AssemblyLine を実行するには、2 つのボタンがあります。1 つ目のボタン (再生アイコン ) を使用した場合、AssemblyLine が開始されて、出力がコンソール・ビューに出力されます。2 つ目のボタン (「**デバッガー**」) を使用した場合、デバッガーとともに AssemblyLine が開始されます。

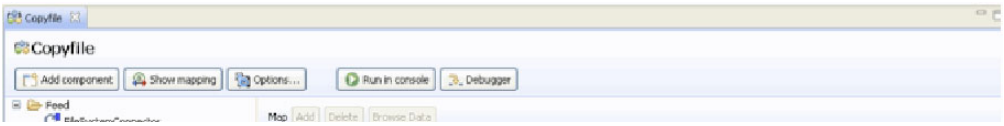
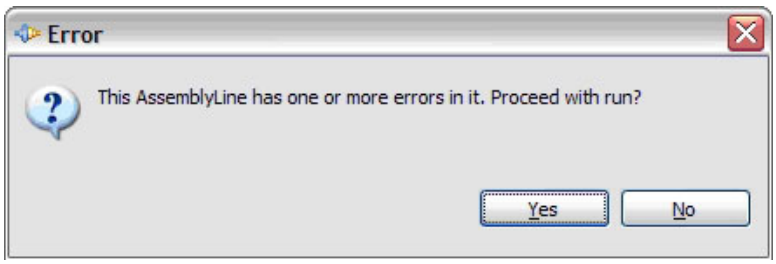


図 80. AssemblyLine を開始する 3 つのオプション

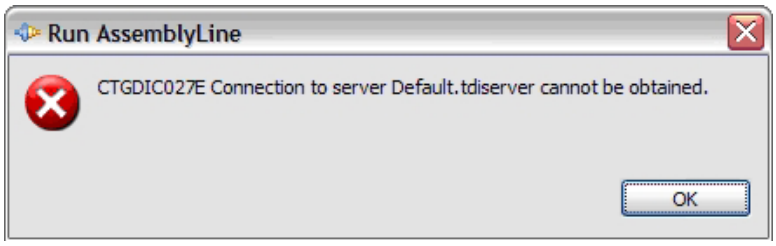
AssemblyLine を開始するプロセスは、3 つのステップで実行されます。

AssemblyLine にエラーが含まれている (出力マップ等が欠落しているなど) 場合は、AssemblyLine の実行を確認する次のようなプロンプトが出されます。

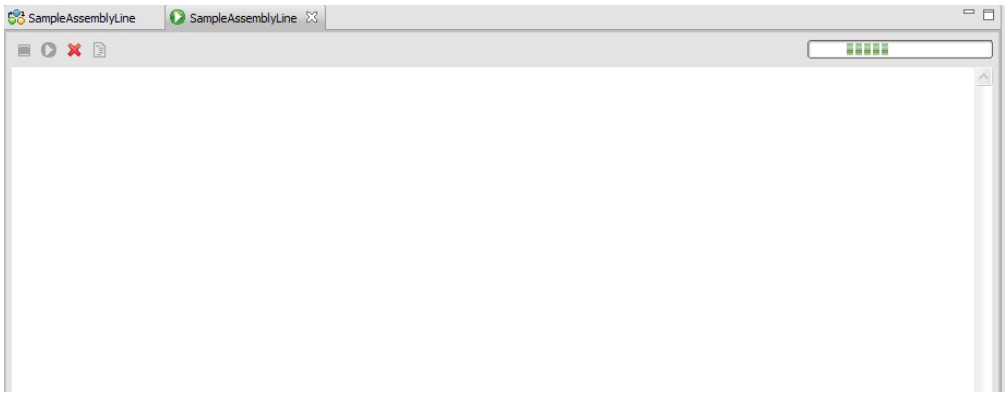


このダイアログが表示されたら、「問題」ビューで、AssemblyLine を中断する可能性があるエラーを確認する必要があります。ただし、開発中にこれらのエラーに気付いても、それでも AssemblyLine を実行したい場合がよくあります。その場合は、Enter キーを押すか、「**はい**」ボタンをクリックして、AssemblyLine を実行してください。

次に、IBM Security Directory Integrator サーバーが使用可能であるかどうかを確認します。サーバーにアクセスできない場合、次のメッセージが表示されます。

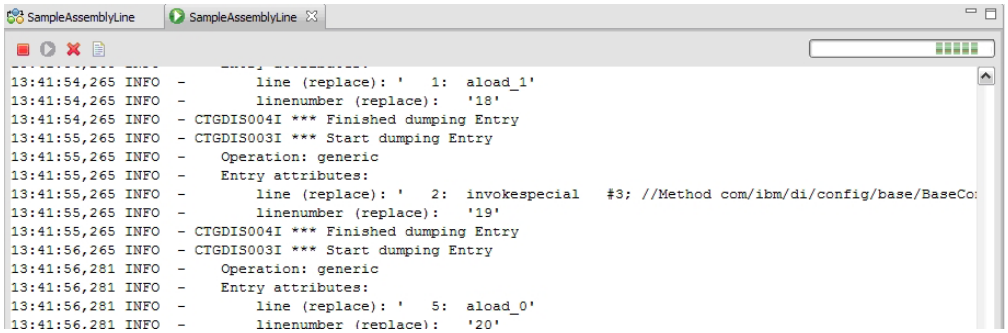


CE から AssemblyLine を実行する場合、最初のステップとして CE は、サーバーにランタイム構成を転送し、AssemblyLine が開始されるのを待ちます。このステップでは、ウィンドウの右上部分に進行状況表示バーが表示されます。AssemblyLine を停止するためのツールバー・ボタンも表示されます (まだ開始していないため、ぼかし表示になっています)。



2 番目のステップでは、AssemblyLine は実行中です。進行状況表示バーがスピンしている場合は、ログ・ウィンドウでメッセージを確認し始める必要があります。この時点では、ツールバーの (左端にある)「停止」ボタンを押して、AssemblyLine を停止することができます。

注: 「停止」ボタンが機能するのは、スレッドの実行時にサーバーが制御を取得している場合のみです。スレッドが IBM Security Directory Integrator の外部の対象を実行しているときには、「停止」ボタンをクリックしてもほとんど効果がない場合があります。



複数の AssemblyLine ウィンドウを開いている場合は、実行中の AssemblyLine の名前の前に「*」(アスタリスク) が付くため、該当する AssemblyLine のうちのどれが実行中であるかを見分けることができます。

AssemblyLine が停止すると (正常に停止した場合でも、「停止」ボタンを押した場合でも)、進行状況表示バーが消えて、AssemblyLine を再実行するためのツールバー項目が使用可能になります。このとき、AssemblyLine は実行中ではないため、「停止」ボタンは使用不可です。

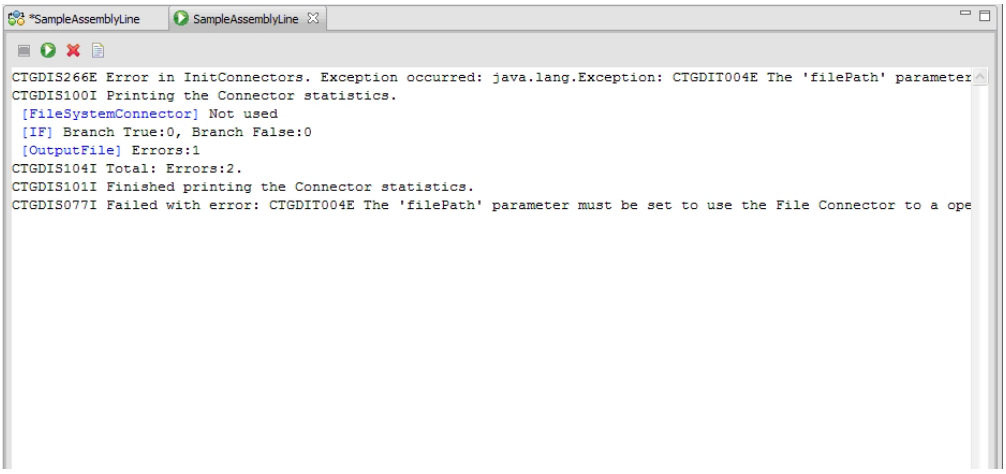


図 81. コンソール・ログ・ウィンドウ

ログ・ウィンドウは、どの時点でも消去することができます。ログ・ウィンドウには、AssemblyLine ログのうちの最新の数百行のみが表示されますが、すべてのログ・メッセージは一時ログ・ファイルに書き込まれます。このため、(右端にある) ツールバーの「ログの表示」ボタンを使用して、このログ・ファイルを別のエディターで開くことが可能です。

注: ログ・ウィンドウの基礎となるログ・バッファーのサイズを、デフォルトの 300 行から別の値に変更することができます。これを行うには、**「ウィンドウ」** > **「設定」** > **「Security Directory Integrator の設定」** > **「「AssemblyLine の実行」ウィンドウの最大行数」**を順に選択します。

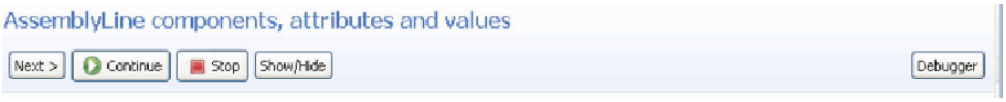
AssemblyLine が終了したら、「再実行」ボタンを使用して AssemblyLine を再実行することができます。

ログ・ウィンドウの青色のテキストに注意してください。このテキストが表示された場合、CTRL キーを押しながらその単語をクリックすると、AssemblyLine の該当部分に移動することができます。

ステッパーとデバッガー

ステッパーとデバッガーは構成エディターに組み込まれているツールであり、AssemblyLine を対話式に開発する際に役立ちます。

ステッパーは 2 つのモードで実行できます。1 つは通常の高機能なデバッガー (「**デバッガー**」ボタンでアクティブ化、 168 ページの『デバッガー』を参照) であり、AssemblyLine の全部分にアクセスできます。もう 1 つはステッパー (「**データ・ステッパー**」ボタンでアクティブ化、『データ・ステッパー』を参照) であり、コンポーネントとフローが単純なビューで表示されます。2 つのモードは、列ビューでボタンをクリックして切り替えます。



ステッパー・ビューでは、「**デバッガー**」ボタンをクリックしてデバッガーに切り替えられます。反対に、デバッガーからステッパーに切り替えるには、「**データ・ステッパー**」ボタンをクリックします。

データ・ステッパー

データ・ステッパーでは、AssemblyLine のすべてのコネクターが列で表示されます。AssemblyLine をステップ実行すると、各コンポーネントの読み取りまたは書き込みデータが表示されます。これらのテーブルに表示されるデータはすべて、コネクターが処理を完了した後のデータです。

図 82. 「データ・ステッパー」メインウィンドウ

データ・ステッパーでは、右側にコンポーネントが表示され、属性マップが縦に表示されます。各コンポーネントは、「閉じる」ボタンをクリックするか、または「**表示/非表示**」ダイアログから選択を外すと表示されなくなります。各コンポーネントの「閉じる」ボタンの左にあるボタン () は、「ここまで実行」のショートカットです。

「**次へ**」および「**実行**」ボタンはそれぞれ、コンポーネントを一度に 1 つずつステップ実行するか、または AssemblyLine を最後まで実行する場合に使用します。「**停止**」ボタンは、実行中の AssemblyLine を一時停止させるため、または一時停止している AssemblyLine を終了させるために使用します。データ・ステッパーの左側には、AssemblyLine の概要と、その下に作業項目が表示されます。概要のビューでは、コンテキスト・メニューから AssemblyLine の再起動を選択できます。これにより、新しいセッションでデバッガーが開始されて、選択したコンポーネントまで実行されます。この方法により、データ・ステッパーからデバッガーに素早く切り替えられます。

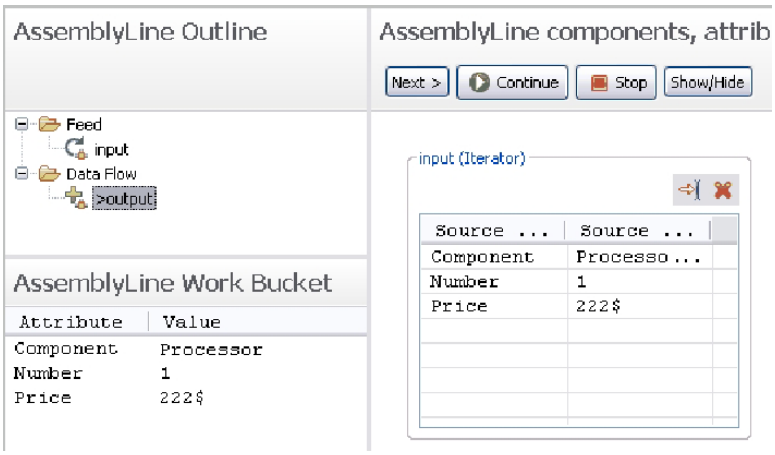


*図 83. データ・ステッパーの「**表示/非表示**」ボタン*

コンポーネントの「**表示/非表示**」ダイアログで、ビューに表示するコンポーネントを選択できます。

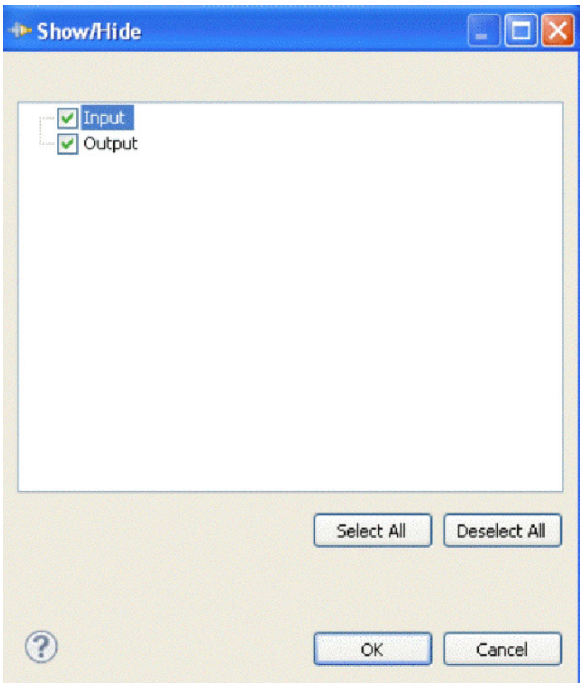


図 84. コンポーネントの「表示/非表示」ダイアログ

デバッガー

「**デバッガー**」ビューを選択すると、ステッパー・ビューによく似たレイアウトが表示されますが、デバッガー・ビューでは AssemblyLine のコンポーネントについて詳細が表示されます。また、監視ウィンドウも表示され、カスタムの式が利用できます。AssemblyLine のコンポーネント・ツリーでは各項目にチェック・ボックスがあり、チェック・マークを付けるか、またはチェック・マークを外して、ブレークポイントを設定または削除できます。また、コンポーネントおよびフックのステップイントゥを可能にするコマンド・ボタンがいくつかあります。

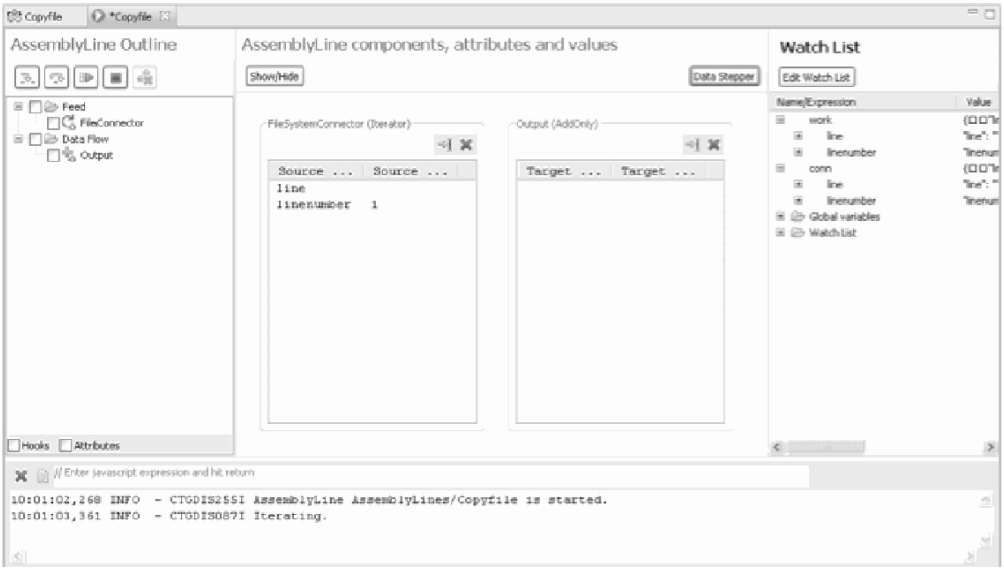


図 85. 「デバッガー」ウィンドウ

(デバッガー・ウィンドウでは、より良く全体像をつかむために、Ctrl+M を使用してエディターを最大化すると便利です。)

左側のツリーには、AssemblyLine のコンポーネントとフックが表示されます。項目ごとにチェック・ボックスの選択を切り替えて、その項目でブレークポイントを設定することができます。項目をダブルクリックして、スクリプトを表示したり、条件付き中断のためのスクリプトを入力することができます。

以下の図は、「**GetNext 前**」フックをダブルクリックした後の条件付き中断のタブを示しています。また、非アクティブのフックすべてを非表示にすることもできます。すべてのフックを表示しておくと、フックがアクティブであるかどうかに関係なく、ブレークポイントを設定することができます。「**属性**」チェック・ボックスでは、ツリー・ビュー内で属性マップを非表示にすることができます。

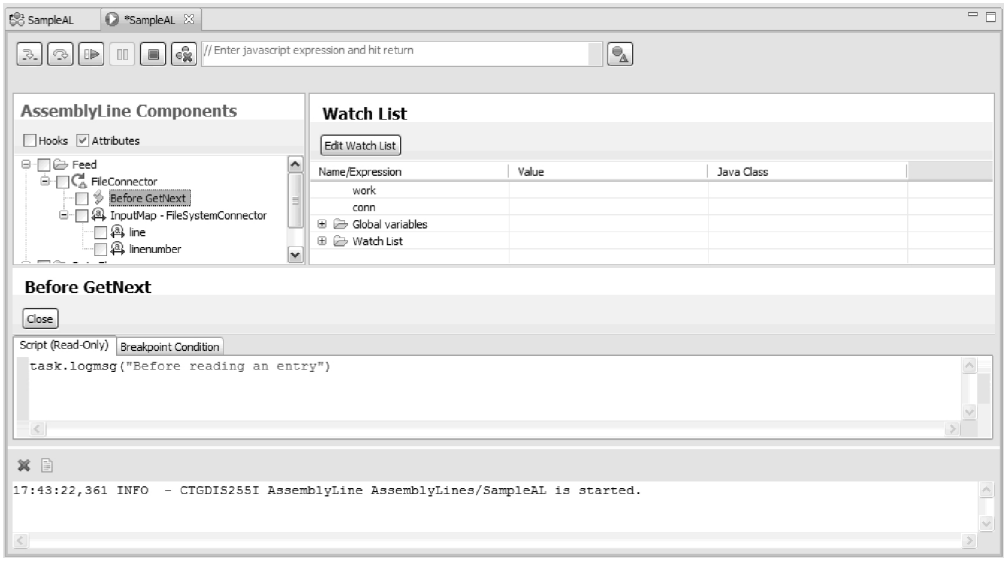


図 86. 「GetNext 前」のデバッガー

「属性マップ」パネルには、各コンポーネントとそれぞれの割り当て、および作業属性に割り当てられている値が表示されます。この値は、AssemblyLine の最後の中断からのスナップショットです。「**前の値**」は最後の中断の前のスナップショット値です。AssemblyLine をステップ実行する場合、各値は作業項目に影響を与えるマップやスクリプトを反映します。

エラーが発生すると、ステッパーは別のダイアログに、例外メッセージとスタック・トレースを示します。画面上のログ・ファイルには、このスタック・トレースは表示されません。チェック・ボックスを選択するか、構成エディターの「設定」ページでの選択によって、このダイアログを表示しないようにすることができます。

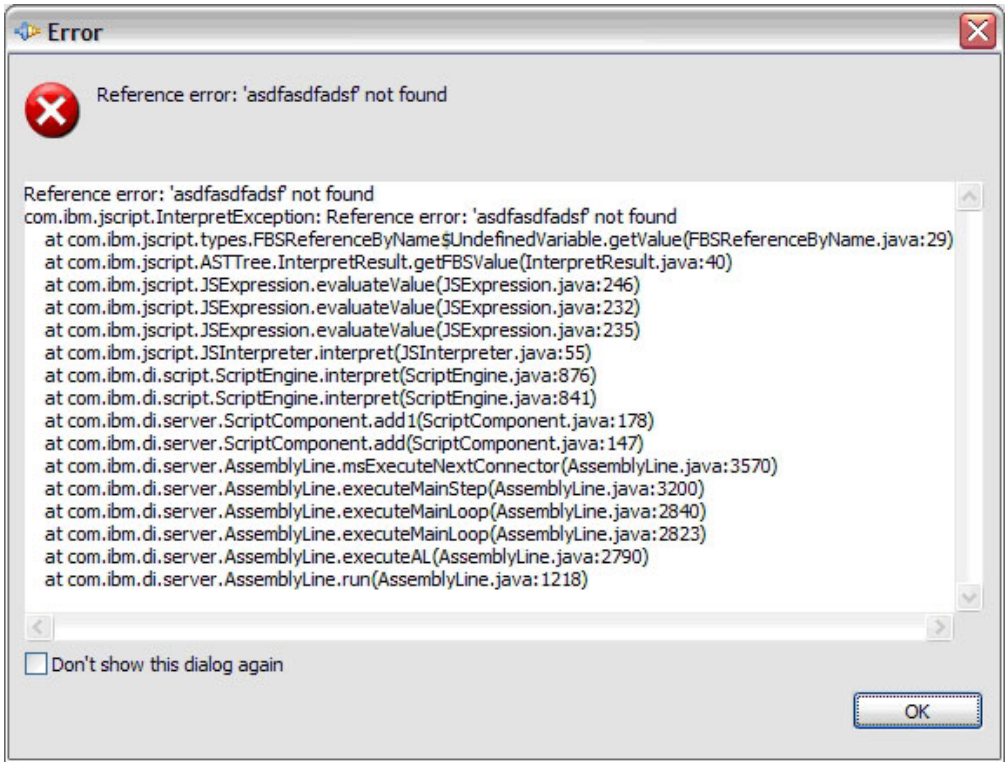


図 87. 「エラー」ダイアログ: スタック・トレース

スクリプトのステップ実行

JavaScript を含むブレークポイントに達したときに、スクリプトをステップイントゥして、1 行ずつ実行することができます。「スクリプト」タブには、実行しようとしているスクリプトが表示されます。「**ステップイントゥ**」コマンド (ステッパーの 2 つのボタンのうちの左側) を使用して、フローを実行することができます。

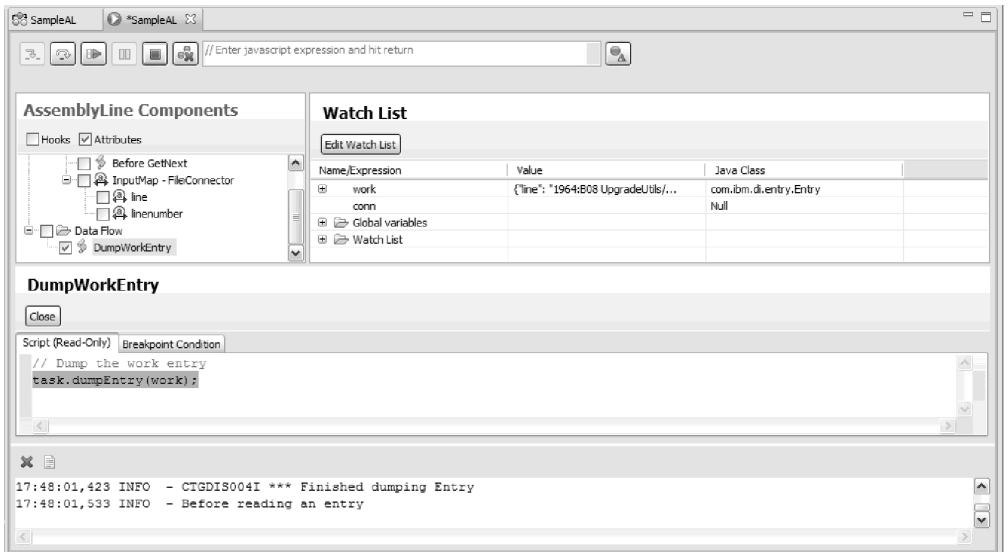


図 88. 「デバッガー」ウィンドウ: スクリプトを 1 行ずつステップ実行する

スクリプトのステップ実行中には、各行が実行される前に強調表示されます。「**評価**」機能を使用して、スクリプトをステップ実行している間に、スクリプト・エンジン変数を表示することもできます。

スクリプト関数が実行されようとしているときに、「**ステップイントゥ**」ボタンを使用して、JavaScript 関数のステップイントゥを実行できます。関数が現行コンテキスト外に定義されている場合 (例えば、フック、属性マップ・スクリプト) は、ウィンドウが置き換わります。

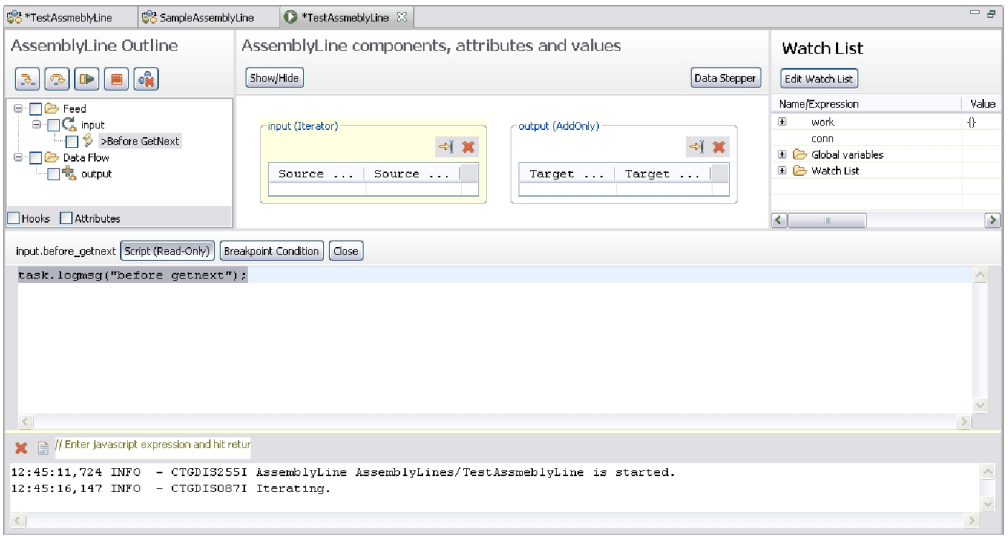


図 89. 関数へのステップイントゥ

この AssemblyLine では、開始前フックに `myfunc1()` という関数が定義されています。どのように表示されるかわかるように、ここではこの関数を Script1 コンポーネントから呼び出します。

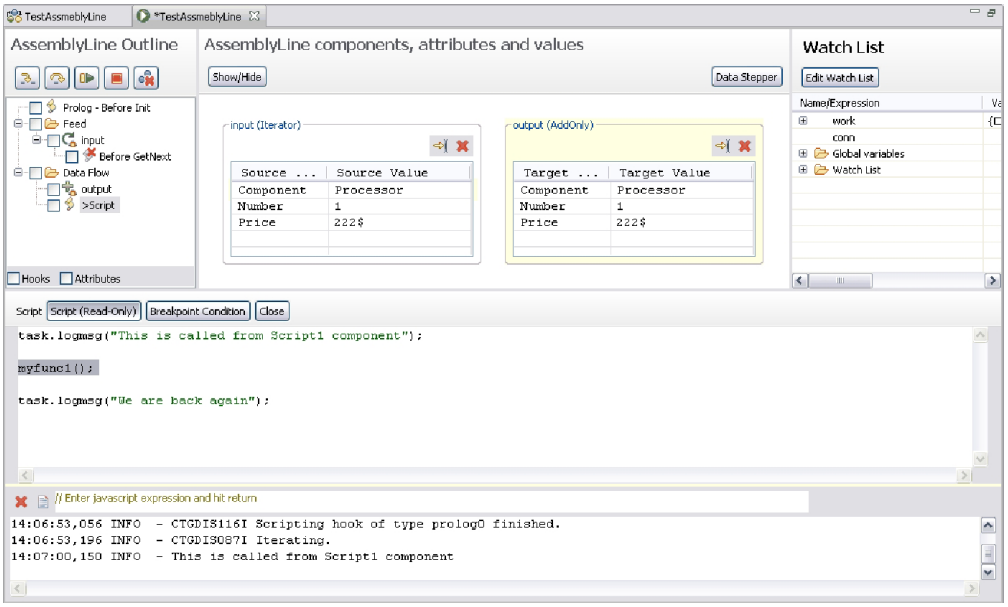


図 90. ステップイントゥされた関数

ここで、「**ステップオーバー**」を押して次の文を続行するか、「**ステップイントゥ**」を押して JavaScript 関数の呼び出しを追跡することができます。

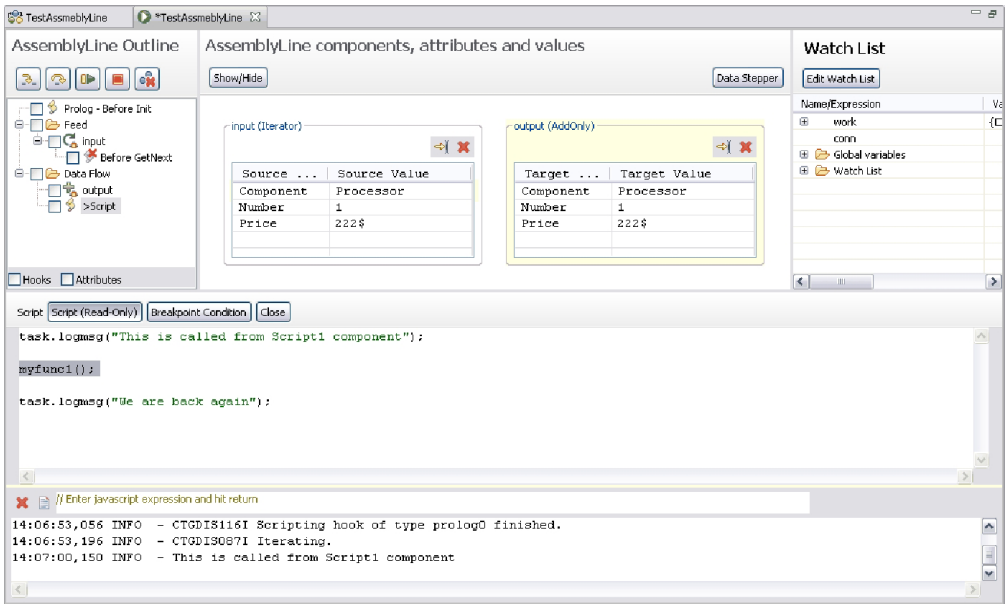
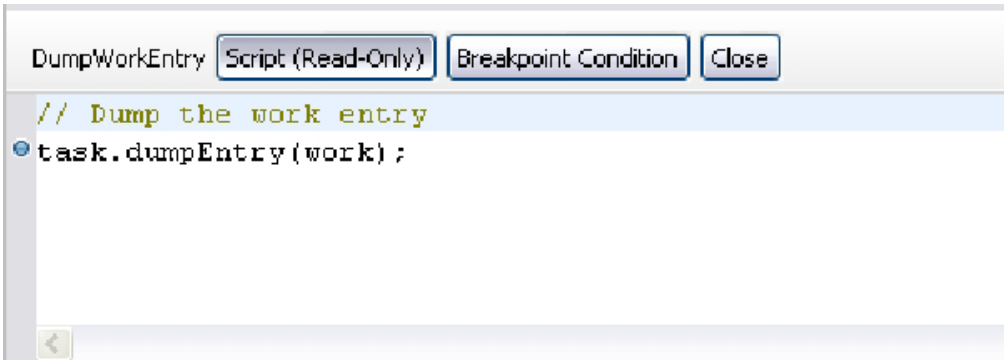


図 91. 関数呼び出しの追跡

ステップイントゥにより、スクリプト・エディターが*開始前*フックからスクリプトに変わります。スクリプトが定義されている場所を示すラベルに変わっています。

また、スクリプト内部にブレークポイントを設定することも可能です。スクリプトまたは属性マップ用のエディターを開いて、左余白をダブルクリックします。青色の中黒が表示されて、その場所に設定されているブレークポイントがあることを示します。もう一度ダブルクリックすると、その中黒が削除されます。左余白またはテキスト・フィールド内を右クリックして、ブレークポイントを切り替えることもできます。



サーバーのデバッグ

IBM Security Directory Integrator サーバーのデバッグとは、IBM Security Directory Integrator サーバー上で開始されたすべての AssemblyLine が、ステップ・モードで開始された場合と同様に、自動的に構成エディターとの間でデバッグ・セッションを開設することを意味します。

サーバーのデバッグをアクティブにするには、「サーバー」ビューで 1 つのサーバーに対するドロップダウン・メニューから「**サーバーのデバッグ**」を選択します。このオプションを選択すると、CE がサーバーに接続し、CE に対してデバッグを指示する Java プロパティーを設定します。

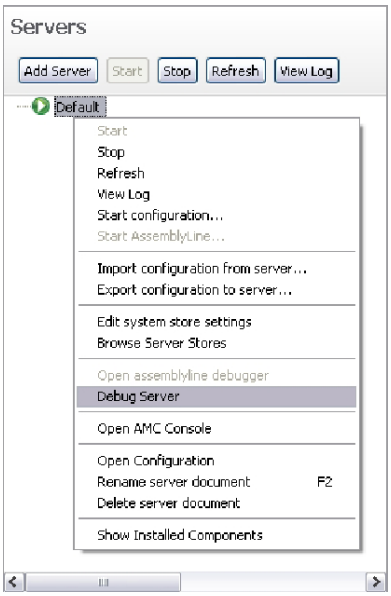
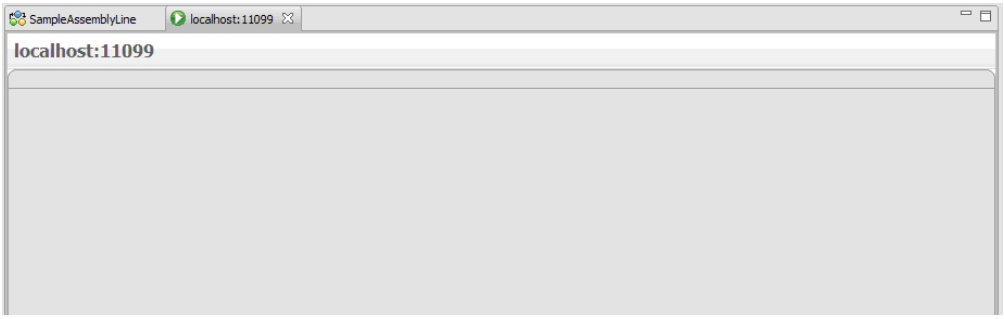


図 92. 「サーバーのデバッグ」オプション

「**サーバーのデバッグ**」を選択すると、新しいウィンドウが開始されて、AssemblyLine がそのサーバー上で開始されるたびに表示されます。これは、CE 内で AssemblyLine を能動的に開始する場合とは異なります。 CE からデバッグ・セッションを開始する場合には、デバッグ・セッション用のウィンドウがあり、このサーバー・デバッグ・ウィンドウには表示されません。



対象とされたサーバー上で別の CE またはコンポーネントによって開始された AssemblyLine では、このウィンドウ内にそれぞれ固有のステッパー・パネルが表示されます。ステッパー・パネルの説明については、 166 ページの『ステッパーとデバッガー』のセクションを参照してください。

実行オプション

AssemblyLine の実行時に指定できるオプションがあります。これらのオプションは保管され、AssemblyLine を実行するたびに使用されます。

実行モードと AssemblyLine 操作を指定できます。また、AssemblyLine に対して初期作業項目 (IWE) を指定できます。

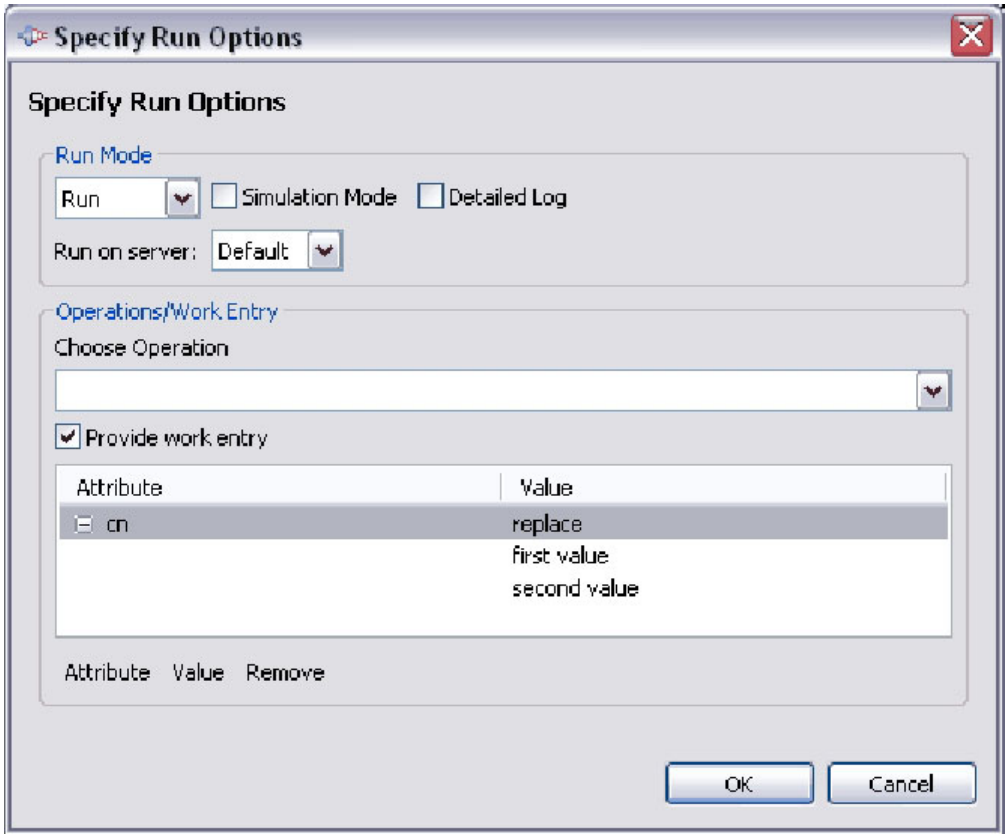


図 93. 「実行オプションの指定」ウィンドウ

初期作業項目に属性と値を追加するには「**属性**」ボタンと「**値**」ボタンを使用します。

サーバーの選択

AssemblyLine を実行すると、AssemblyLine はローカル開発サーバー上で実行されます。このサーバーは CE によって始動され、そのソリューション・ディレクトリーは *TDI* サーバー・プロジェクト内にあります。新しい IBM Security Directory Integrator サーバーを作成するときに、特定のプロジェクトの「**プロパティー**」ダイアログを使用して、そのプロジェクトの優先サーバーを変更することができます。

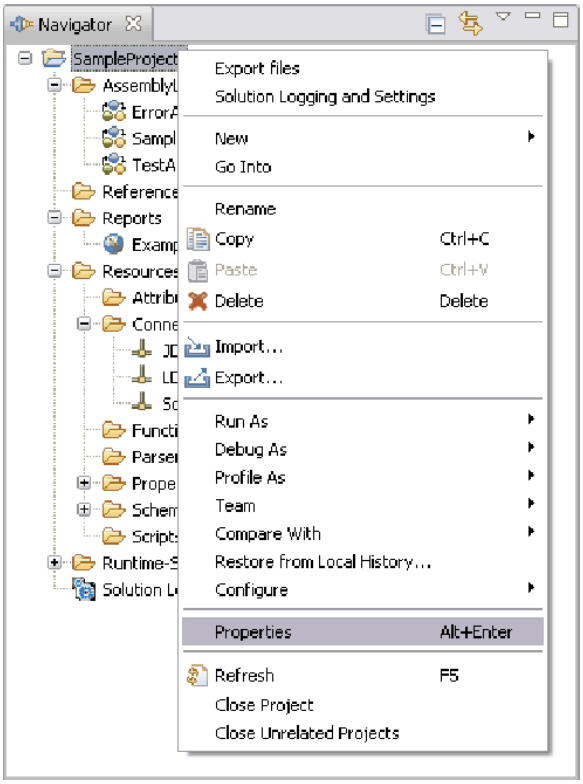


図 94. プロジェクトの「プロパティー」メニュー選択

この項目を選択すると、そのプロジェクトの「プロパティー」ダイアログが表示されます。「**Tivoli Directory Integrator のプロパティー**」を選択して、そのプロジェクトのデフォルトのサーバーを変更します。

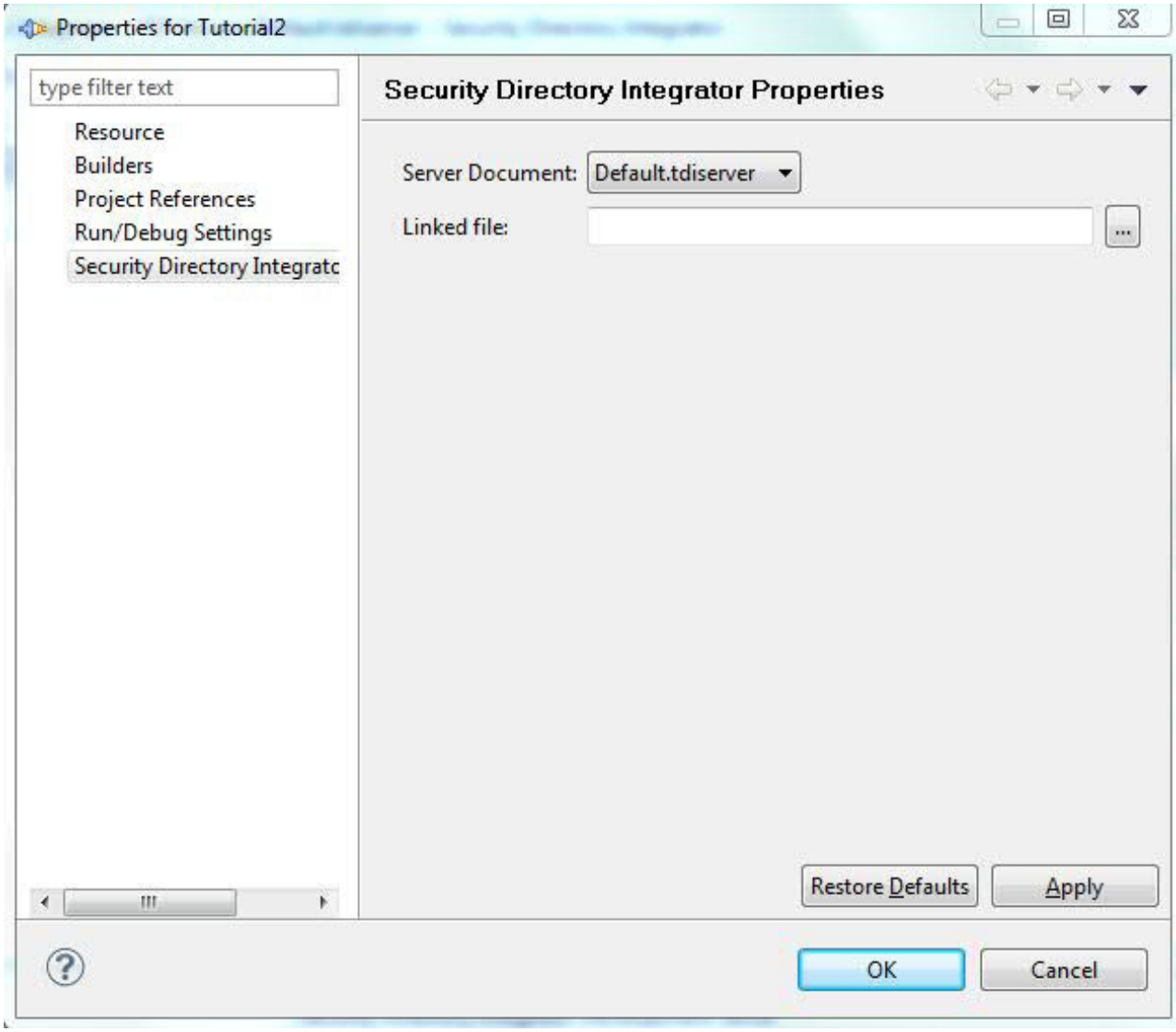


図 95. プロジェクトのプロパティー

チーム・サポート

IBM Security Directory Integrator 構成エディターには、ソース・コード制御リポジトリーを使用することにより、ユーザーの間でプロジェクトを共用できるようにする Eclipse プラグインが組み込まれています。

IBM Security Directory Integrator には、pserver、pserverssh2、ext、および extssh タイプの接続をサポートする CVS ライブラリーが付属しています。 Eclipse CVS プラグインの詳しい情報については、Eclipse CVS のサイト (http://www.eclipse.org/eclipse/platform-cvs) を参照してください。

チーム共有機能を使用するためには、CVS リポジトリーに対するアクセス権限が必要です。ほとんどのオペレーティング・システムは CVS リポジトリーをホストすることができ、また、バイナリー・パッケージはネットから一般的に入手できま

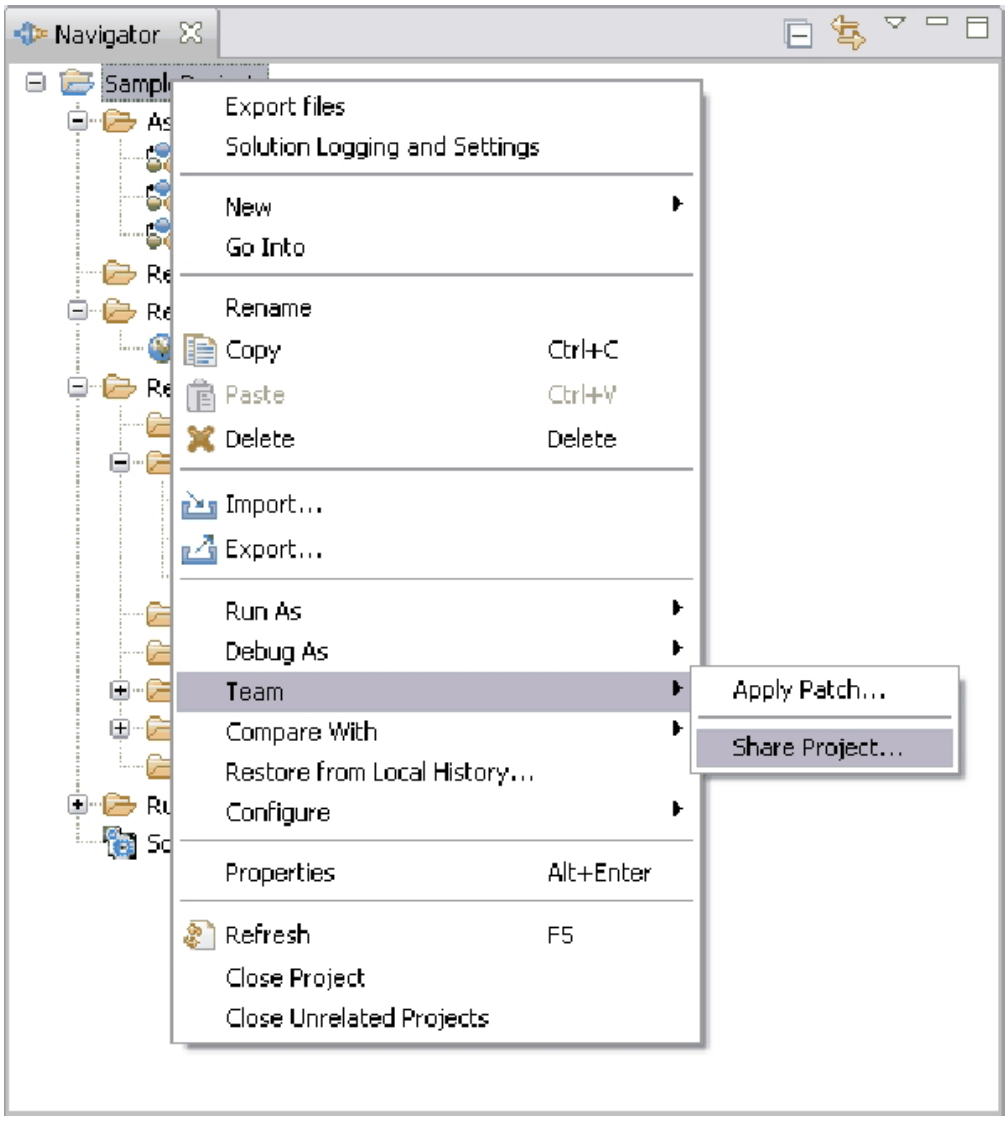
す。Eclipse CVS サイトには、よくある質問が掲載されており、ユーザーが直面する可能性のある多数の一般的な問題の解決に役立ちます。

CVS サーバーのインストールや構成でサポートが必要な場合は、CVS Wiki サイト (http://ximbiot.com/cvs) を参照してください。また、Web で「CVS サーバーのインストール方法 (how to install a cvs server)」を検索すると、さまざまなオペレーティング・システムやプラットフォーム上に CVS リポジトリーをインストールし構成する方法を詳しく説明する多数の Web サイトが表示されます。

プロジェクトの共用

CVS を使用して他のユーザーとプロジェクトを共用できます。

プロジェクトを共用するには、プロジェクトのドロップダウン・メニューから「**チーム**」 > 「**プロジェクトの共用...**」オプションを選択します。



これにより別の画面が開くので、ここでソース・コントロール・リポジトリーのパラメーターを指定できます。

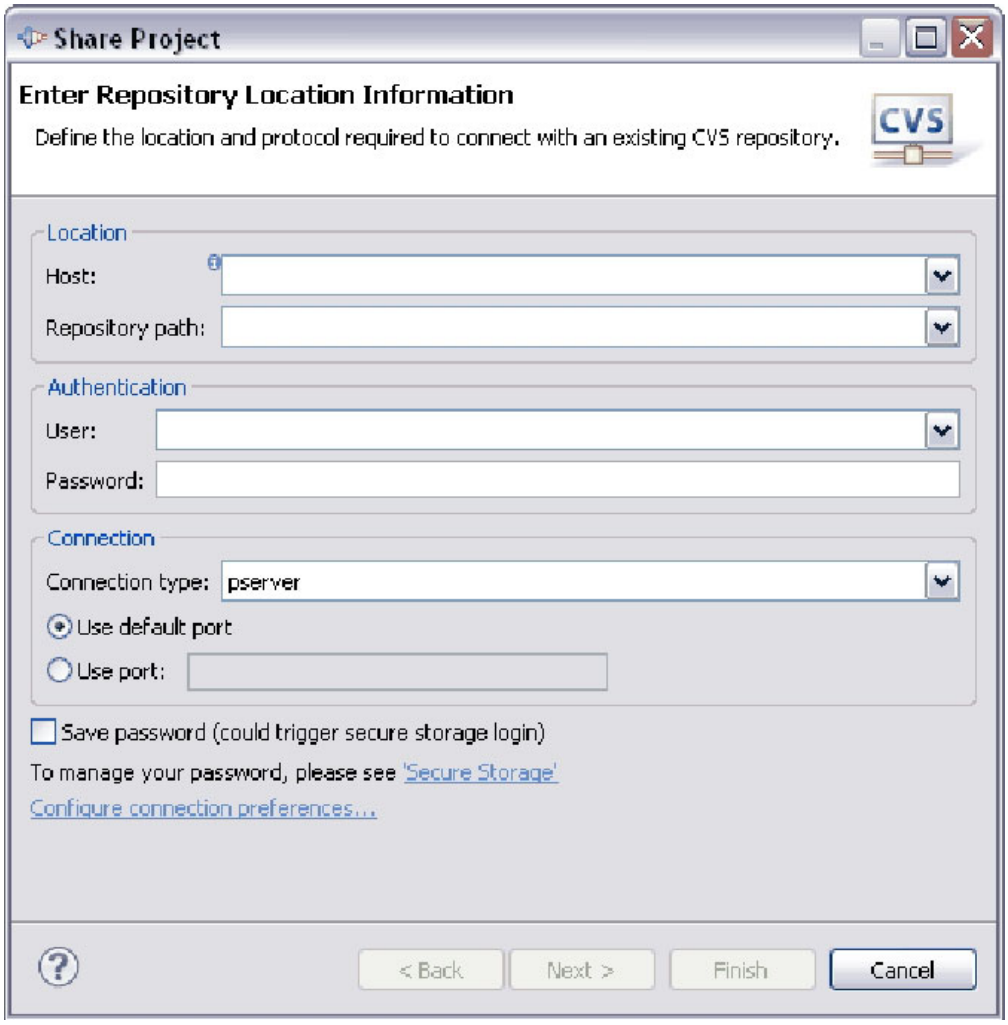
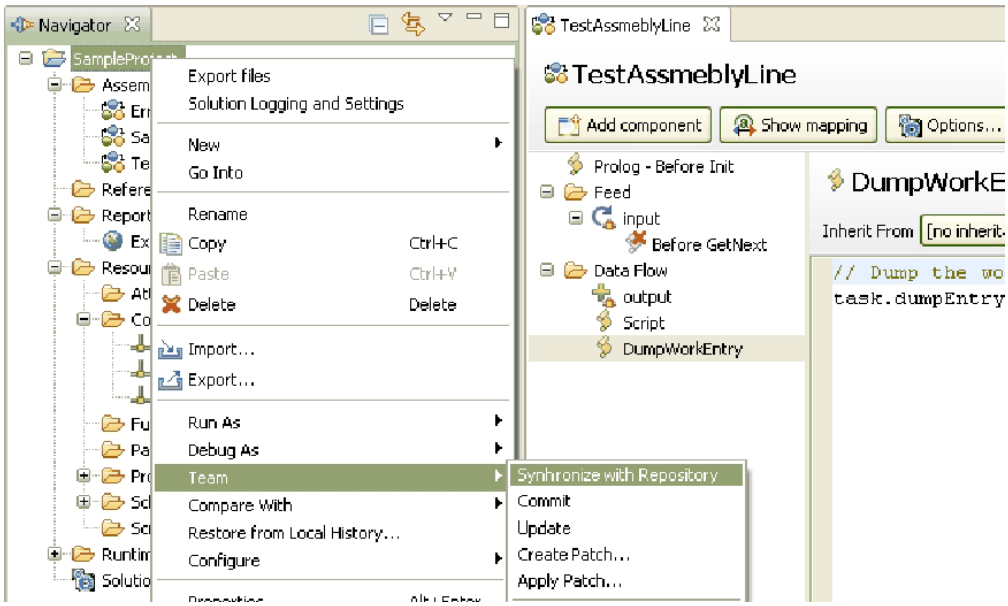


図 96. CVS プロジェクト共用ウィンドウ

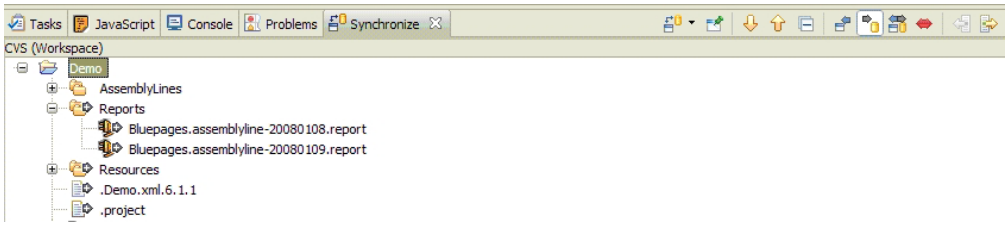
CVS サーバーでのプロジェクト共有を設定するウィザードを完了します。

注: プロジェクト内の実際のファイルと、これらのファイルが格納されているディレクトリーの構造は、リポジトリーによって適切に制御できます。空のディレクトリーは制御の対象外です。

プロジェクトの共用の設定が完了したら、リポジトリーとの同期を実行し、行った変更と他のユーザーによる変更をコミットできます。



「**チーム**」 > 「**リポジトリーと同期化**」を選択して、同期ビューを開きます。

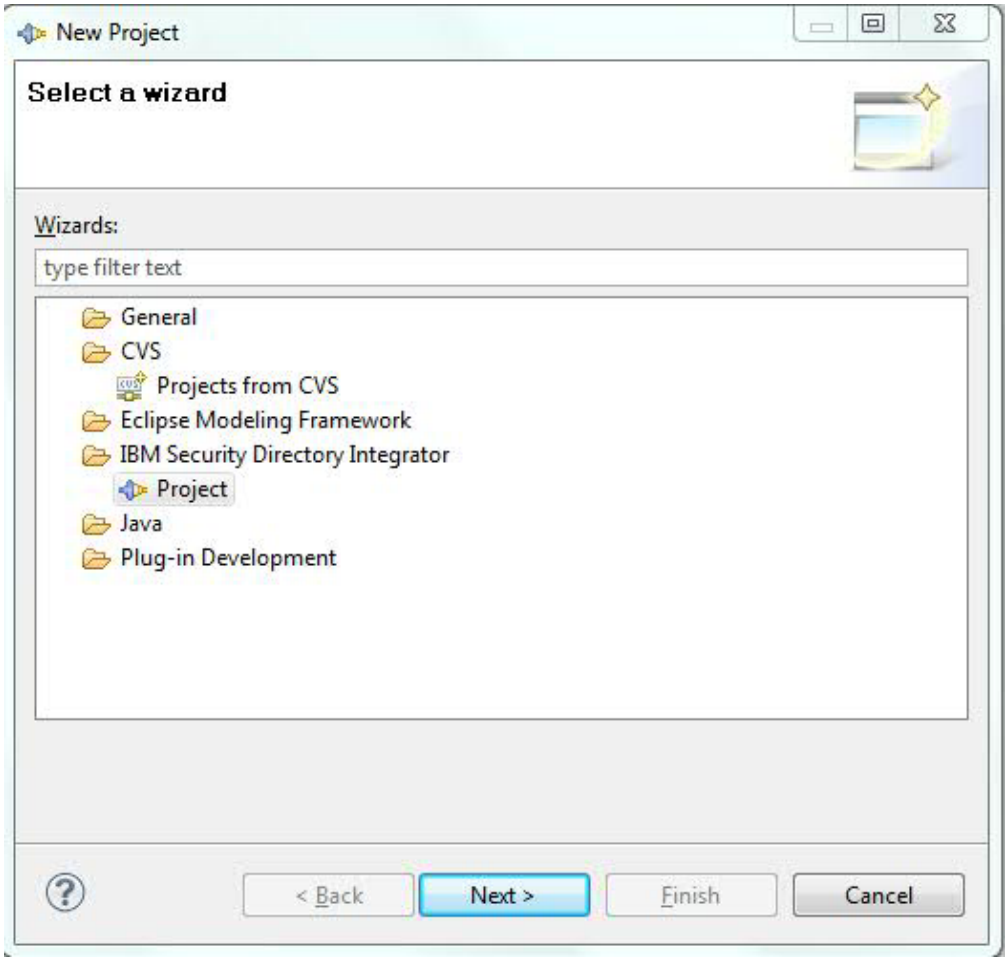


このビューには、更新する必要があるファイルと、更新が存在するファイルが表示されます。変更のコミット後、ナビゲーターにはファイルの追加情報が表示されます。

共用プロジェクトの使用

他のユーザーとの共用プロジェクトを使用するには、CVS プロジェクト・ウィザードを使用します。

メインメニューから「**ファイル**」 > 「**新規**」 > 「**新規プロジェクト...**」を選択して、CVS プロジェクト・ウィザードを選択します。



ウィザードを実行し、ワークスペースにプロジェクトを取り込みます。

「問題」ビュー

コンポーネントを保管すると、IBM Security Directory Integrator プロジェクト・ビルダーによりそのコンポーネントのランタイム構成ファイルが更新され、行った変更が反映されます。

プロジェクト・ビルダーはコンポーネントの妥当性検査も実行し、警告とエラーを「問題」ビューに表示します。「問題」ビューでは、警告は次のように表示されます。

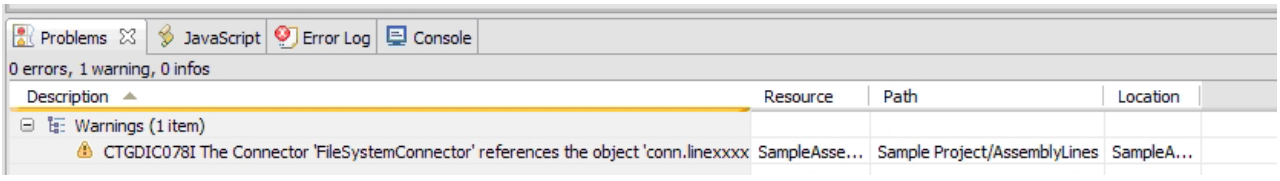


図 97. 「問題」ビュー・ウィンドウ

このビューで行をダブルクリックすると、問題の発生場所が表示されます。

「問題」ビューに表示される問題には、次のものがあります。

- 未定義のスキーマ項目への参照

  「conn.abc」などの構成体を使用するときに、スキーマに「abc」が定義されていない場合にこの警告が表示されます。
- 未定義の作業属性への参照

  「work.abc」などの構成体を使用するときに、「abc」の存在が不明な場合にこの警告が表示されます。
- スクリプトの構文エラー
- 複数のサーバー・モード・コネクターが含まれる AssemblyLine

JavaScript の機能拡張

構成エディターでスクリプトを編集する場合には、随時、JavaScript コードの編集用に特別に拡張されたテキスト・エディターを使用できます。

このエディターでは、コードの完了、構文のカラーリング、構文エラーのマーキングを行うことができます。

以下に、いくつかの機能拡張を示します。

- 『コードの完了』
- 184 ページの『構文のカラーリング』
- 185 ページの『構文検査』
- 185 ページの『ローカル評価』
- 186 ページの『外部エディター』

コードの完了

JavaScript エディターは、コードの完了をサポートしています。

入力を行うと、エディターは、特定のキー・ストロークに対応します。ドット (.) は、ポップアップ・メニューを表示するコードの完了をアクティブにします。このメニューには、IBM Security Directory Integrator に固有の関連するすべてのメソッド、フィールドおよび機能が表示されます。コードの完了は、`Ctrl-<Space>` キーを組み合わせて手動でアクティブにすることもできます。コードの完了では、標準の JavaScript スクリプト・タイプ (つまり、ストリング、数値など) を取り扱えます。これは、カスタム完了が IBM Security Directory Integrator 固有のオブジェクトを取り扱えるのと同様です。*conn* および *work* のようなオブジェクトは、使用可能な属性とともに、オブジェクトのフィールドやメソッドのリストを提供します。

コードの完了は、エディターが以下の式に Java クラス名を派生させることができる限り、機能します。

エディターは、単一または二重引用符、および中括弧 (「{}」) にも応答します。単一または二重引用符を入力すると、エディターは、引用符をもう 1 つ自動的に挿入し、2 つの引用符の間に脱字記号を配置します。これは、ストリング定数を入力しやすくするために行われます。

中括弧を入力すると、ブロックをインデントするための自動インデントが行われます。左中括弧を入力して Enter (キー) を押すと、タブが挿入され、適切なインデントとともに次の行に脱字記号が配置されます。反対に、右中括弧を入力すると、中括弧のインデントが解除されます。

マッピングとコードの完了

CE に属性を追加すると、属性名に基づいて単純式が生成されます。名前にドットや無効なスクリプト・オブジェクト ID であるその他の文字が含まれている場合、その名前は ["attribute name"] 内に格納されます (例えば、`conn["http.body"]` のようになります)。項目が階層になっているかどうかは、この表記を使用する場合には関係ありません。これは、この表記がいずれの場合でも有効であるためです。有効な JavaScript ID である属性は、そのまま使用されます (例えば conn.cn)。階層スキーマがある場合でも、マップの単純式が生成されます。例えば、>b->c という階層があり、「c」の部分をマップすると、属性を参照する ["a.b.c"] が生成されます。

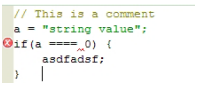
スクリプト・エディターで、コード完了が同様に機能することを確認できます。コード補完では、補完がプレーン・テキストで表示されます (例えば http.body) が、Enter (キー) を押して式を完了すると、無効なスクリプト・オブジェクト名である属性名が同じ方法で格納されます。

構文のカラーリング

エディターでの構文のカラーリングは、基本的な機能です。エディターにより、コメントとストリングが装飾されます。

構文検査

入力を行うと、エディターは、バックグラウンドでスクリプトの構文検査を実行します。スクリプトにエラーがあった場合は、エディターの余白にエラーが表示されます。また、該当するテキストに赤い波形の下線が引かれ、JavaScript インタープリターがエラーを検出した箇所にマークが付けられます。



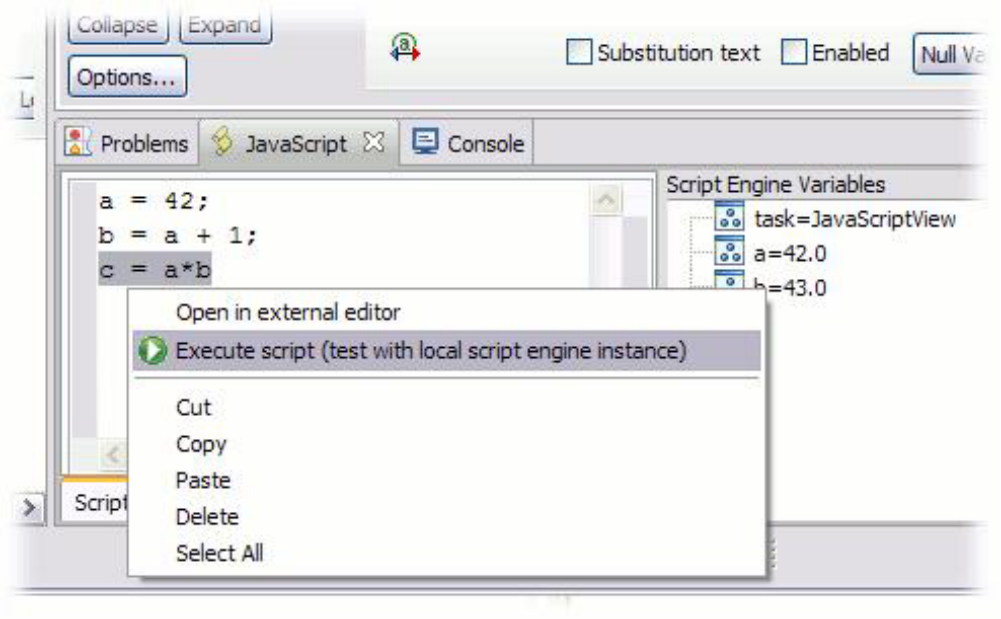
罫線 内 (左余白) のエラー・マークにマウス・ポインターを移動すると、ポップアップ・ウィンドウにエラー・メッセージが表示されます。

左側の罫線はテキスト・ウィンドウに同期し、マークはテキスト・ウィンドウ内の行に直接関連しています。ウィンドウに表示されるテキストにエラーがない場合、この罫線内にマークは表示されません。

しかし、右側の概要の罫線には、テキスト・エディターの位置に関係なく、スクリプト全体のすべてのエラーが表示されます。罫線内のマークにマウス・ポインターを移動すると、左の罫線内に表示されたものと同じエラー・メッセージのポップアップが表示されます。マーカーをクリックすると、問題が特定された行にエディターの位置が変わります。

ローカル評価

スクリプトを編集する場合は、右クリックして、ドロップダウン・メニューから「**スクリプトの実行**」を選択し、選択したテキストの簡易評価を行います。JavaScript ビューは、スクリプトに対するより詳細なテスト環境です。このビューでは、次のように、スクリプトを実行し、別のウィンドウにスクリプト・エンジン変数を表示することができます。



外部エディター

コンテキスト・メニューを使用すれば、好みの JavaScript エディターでスクリプトを編集することができます。

テキスト・フィールドで右クリックして、「**外部エディター内で開きます**」を選択します。エディターの構成は、「**ウィンドウ**」 > 「**設定**」 > 「**Security Directory Integrator**」の設定タブで行います。

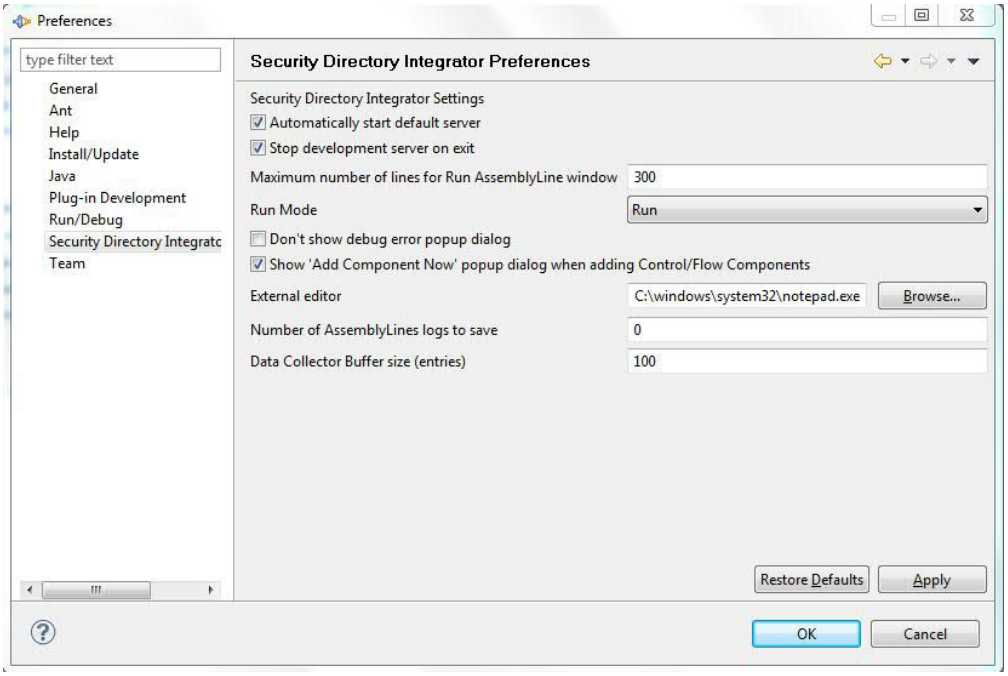
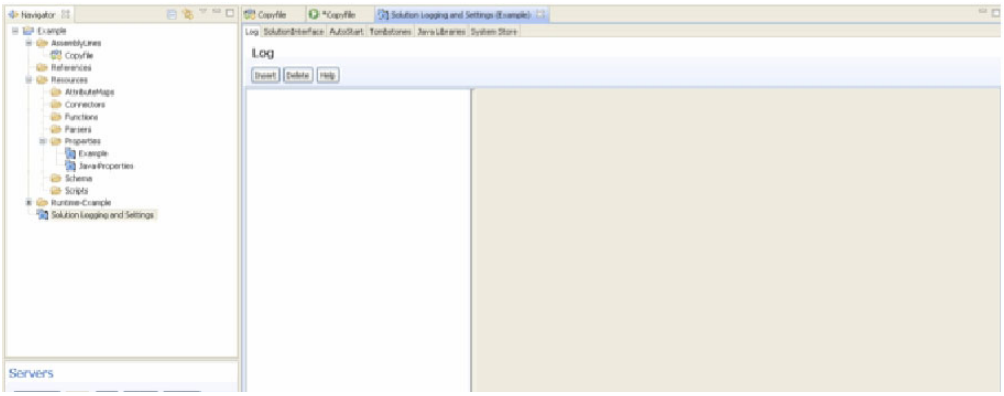


図 98. 構成エディターの「設定」ウィンドウ

ソリューションのロギングおよび設定

「ソリューションのロギングおよび設定」ウィンドウでは、プロジェクトのソリューション固有の領域を編集できます。

このウィンドウを開くには、プロジェクトまたはプロジェクト・ファイルを選択し、プロジェクト・ツリーの「**ソリューションのロギングおよび設定**」をダブルクリックします。



「ソリューションのロギングおよび設定」では、次の項目を変更できます。

- 『システム・ストア設定』
- 189 ページの『ロギング』
- 189 ページの『トゥームストーン』
- 190 ページの『Java ライブラリー』
- 190 ページの『自動開始』
- 191 ページの『ソリューション・インターフェースの設定』

システム・ストア設定

プロジェクトのシステム・ストア設定は、IBM Security Directory Integrator サーバーによって定義されたデフォルトのシステム・ストアを指定変更します。有効な場合、構成ではサーバーのシステム・ストアではなく構成されているシステム・ストアが使用されます。

システム・ストアの設定は、次の 2 つのレベルで行えます。

1. ワークスペースの IBM Security Directory Integrator プロジェクト・レベル
2. IBM Security Directory Integrator サーバー・レベル

プロジェクト・レベルでの設定は、サーバー・レベルでの設定よりも優先されます。つまり、プロジェクトに特定のシステム・ストアを定義した場合は、構成エディターでのコンポーネントの実行中にそのシステム・ストアが使用されます。プロジェクトにシステム・ストア設定を定義しなかった場合は、コンポーネントの実行に使用するサーバーのシステム・ストアが使用されます。

プロジェクト・レベルでの設定

タイトルのドロップダウン・メニューから事前定義テンプレートを選択できます。また、システム・ストア設定をロードしてワークスペースのファイルに保管することもできます。すべてのテーブル (デルタ、プロパティーなど) がこのデータベースに格納される点に注意してください。

「**Derby Embedded**」などのようにリストされているメニュー項目は事前定義テンプレートであり、構成パネルにロードできます。ロードしたテンプレートは必要に応じて変更し、続いて構成で更新できます。また、「**テンプレートのロード...**」でローカル・ファイル・システムからロードすることもできます。

このパネルに行われた変更は構成ファイルに保存され (プロジェクトのエクスポート時)、AssemblyLine が実行されているサーバーの設定に関わらず、そのすべての AssemblyLine で使用されます。

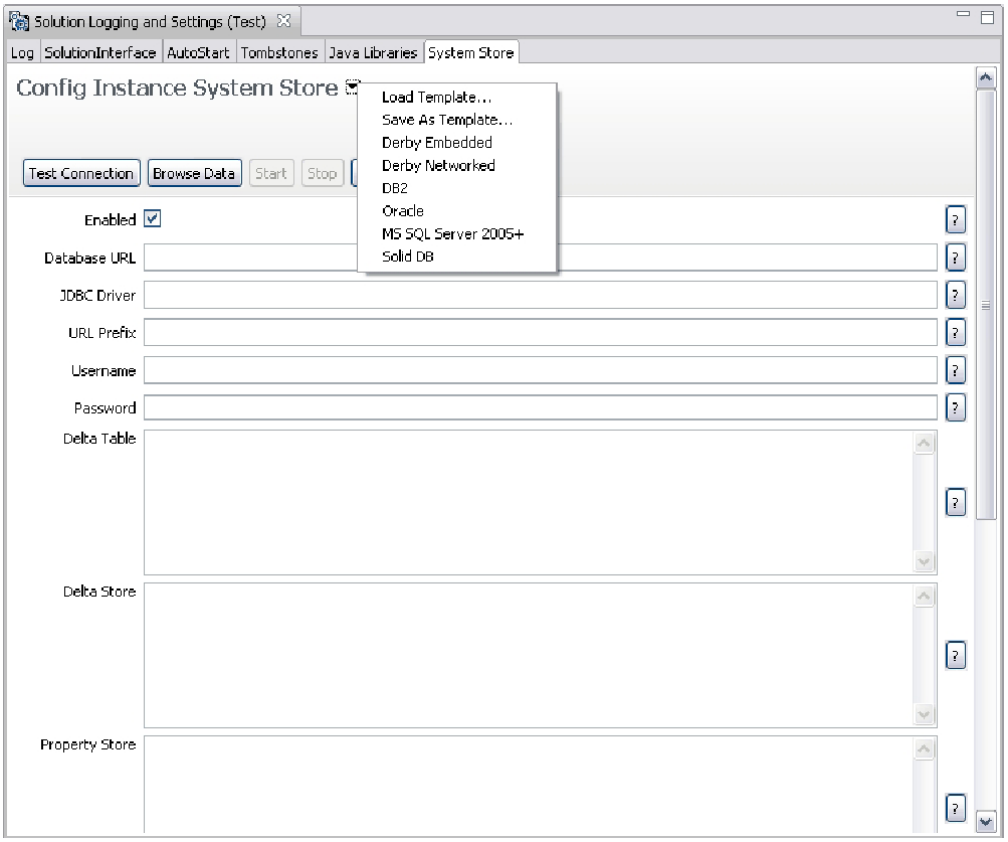


図 99. 構成システム・ストア設定

サーバー・レベルでの設定

「サーバー」ビューでサーバーのドロップダウン・メニューから「**システム・ストア設定の編集**」を選択しても同じ画面が表示されます。これにより、システム・ストア設定変数が更新され、ソリューション・プロパティー・ファイルに保管されます。新規設定をアクティブにするため、サーバーを再始動する必要があります。

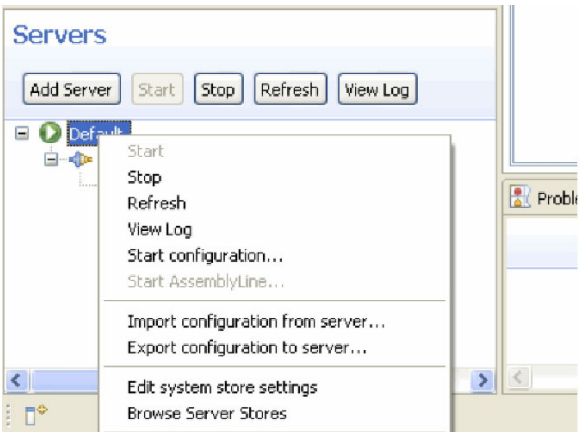
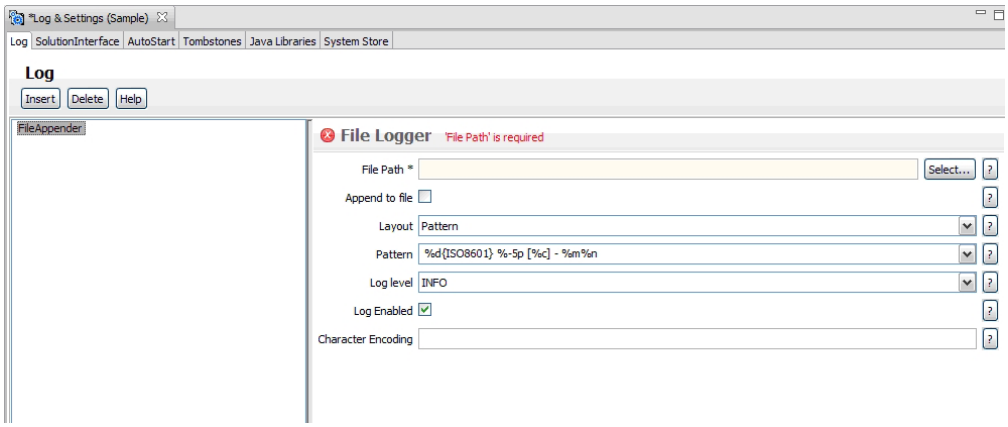


図 100. サーバー文書のコンテキスト・メニュー

ロギング

「**ロギング**」ビューには、ソリューションのロガーが表示されます。

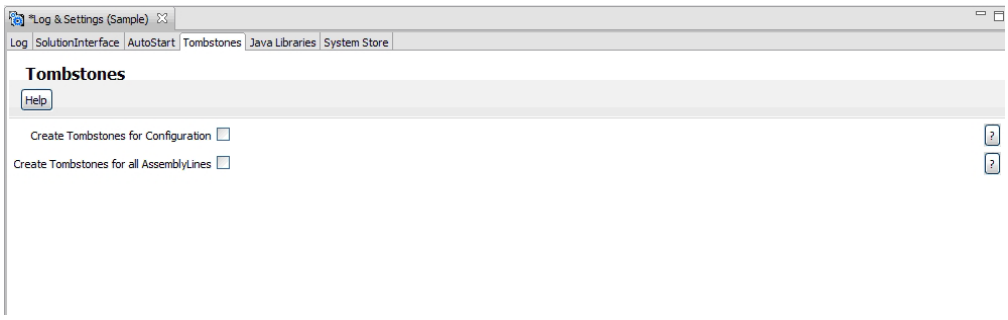
ロガーを追加するには、タイトル・バーで「**挿入**」をクリックし、除去するには「**削除**」をクリックします。



詳しくは、「*インストールと管理*」の『ロギングおよびデバッグ』セクションを参照してください。

トゥームストーン

プロジェクトのトゥームストーン構成は「**トゥームストーン**」タブに表示されます。



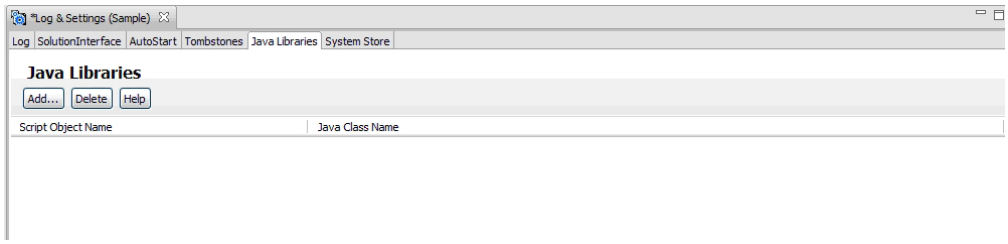
トゥームストーンとは、AssemblyLine の実行レコードであり、何らかの統計 (イテレーターにより読み取られた項目の数、スキップされたレコードの数、更新されたレコードの数など) が含まれています。

「**構成用のトゥームストーンの作成**」を選択すると、このプロジェクトから構成ファイルが派生され、IBM Security Directory Integrator Server にロードされ、終了するたびに、トゥームストーンが生成されます。

「**すべての AssemblyLine 用のトゥームストーンの作成**」を選択すると、このプロジェクトの AssemblyLine がサーバーで実行され、終了するたびに、トゥームストーンが生成されます。

Java ライブラリー

「**Java ライブラリー**」タブには、自動的にロードされ、スクリプト・エンジンの各インスタンスで定義されている Java クラスが表示されます。



自動開始

「**自動開始**」タブのリストに、構成インスタンスの開始時に自動的に開始される AssemblyLine の名前を入力できます。

図 101. 自動開始設定

項目をリストに追加するには、「**挿入**」をクリックします。これにより、ワークスペースの既存の AssemblyLine から選択できます。

「開始項目」リストから AssemblyLine を除去するには、除去する AssemblyLine を選択して、「**削除**」をクリックします。

ソリューション・インターフェースの設定

構成に関する追加情報を入力するには、ソリューション・インターフェース設定を使用可能にします。この情報は通常 Webadmin ツール (AMC) により使用されますが、ソリューション・インターフェース構成にアクセスできるその他のクライアントが使用することもできます。

このパネルの最初の 2 つのフィールドは、ソリューション名と使用可能状況です。ソリューション名のデフォルト値は、プロジェクト名自体です。ソリューション名をブランクのままにすると、エクスポートされる構成ファイルにはソリューション ID は含まれません。「使用可能」チェック・ボックスにより、構成が使用中であるかどうかが判別されます。

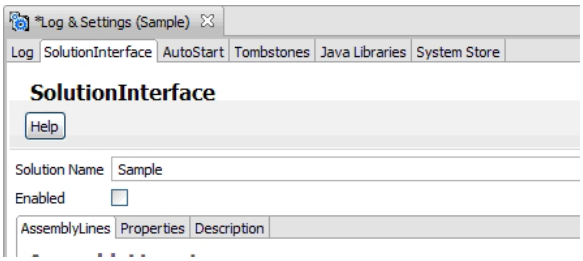


図 102. ソリューション・インターフェースの設定

ソリューション・インターフェース設定は、3 つのセクションからなります。それぞれのセクションは次のタブに対応します。

- 『AssemblyLine』
- 192 ページの『プロパティー』
- 192 ページの『説明』

AssemblyLine

このタブには、プロジェクト内のすべての AssemblyLine が表示されます。開始/停止のためにユーザーに対して表示する必要がある AssemblyLine を確認できます。

ヘルス AssemblyLine は、実行中の構成の正常性を報告するために使用される特殊な AssemblyLine です。この AssemblyLine は、構成の状態に関するカスタム・フィードバックをユーザーに提供するためものです。ヘルス AssemblyLine が呼び出されると、状況を報告する作業項目の 2 つのフィールド (`healthAL.result`および `healthAL.status`) が戻されます。この状況でのこれらのフィールドの使用法について詳しくは、Action Manager (AM) の資料を参照してください。(AMC でオンラインで参照するか、または「*インストールと管理*」を参照)。ポーリング間隔により、クライアント (AMC や AM など) がヘルス AssemblyLine を呼び出す頻度が指定されます。

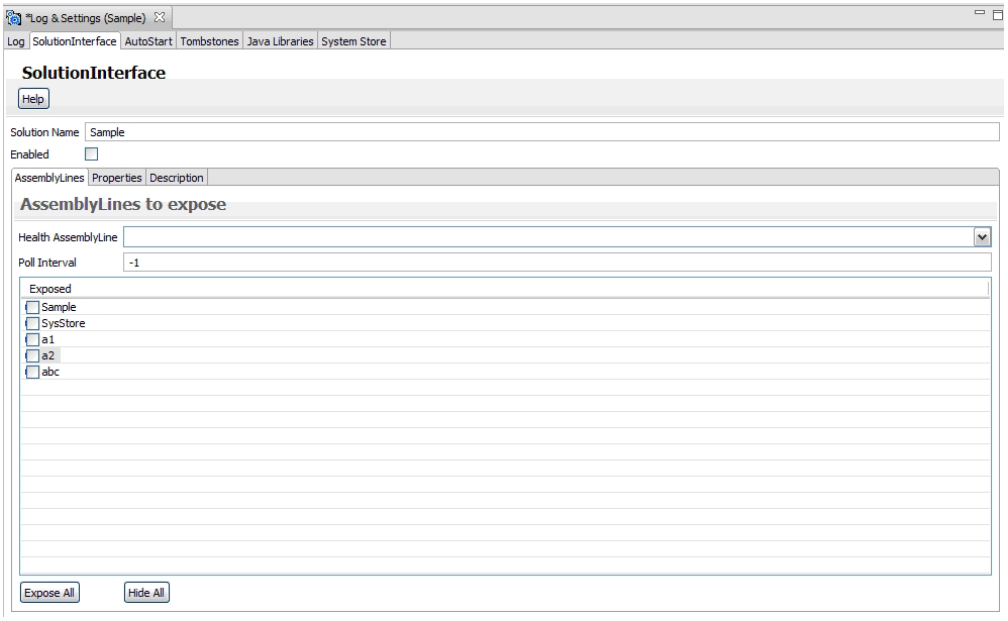


図 103. ソリューション・インターフェースの設定: AssemblyLine

プロパティー

このタブでは、ユーザーに対して表示されるプロパティーと、その表示方法を定義します。

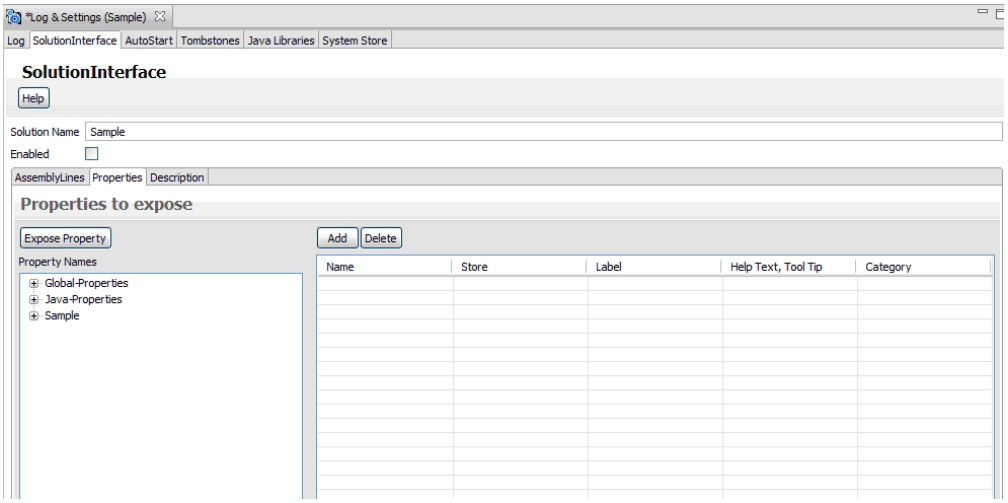


図 104. ソリューション・インターフェースの設定: プロパティー

説明

このタブでは、ソリューションを説明するテキストを入力します。これは、文書化のみを目的としており、IBM Security Directory Integrator でその他の方法で使用されることはありません。

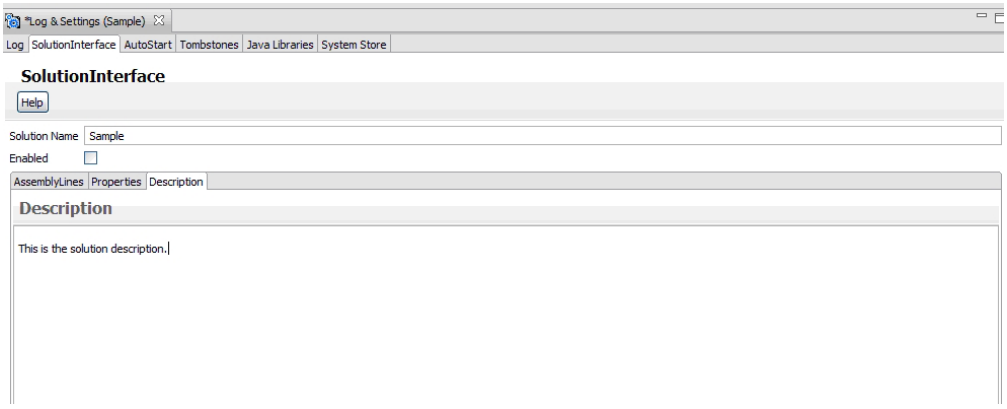


図 105. ソリューション・インターフェースの設定: 記述

サーバー・プロパティー

プロパティー・エディターを使用してプロジェクトのプロパティーを編集します。

「**ファイル**」 > 「**新規**」 > 「**プロパティー**」ウィザードを使用して新規プロパティー・ファイルを作成し、プロパティー・ストアの名前を入力します。

プロパティー・エディターの「**ダウンロード**」および「**アップロード**」コマンドを使用して、プロパティーの内容を交換できます。

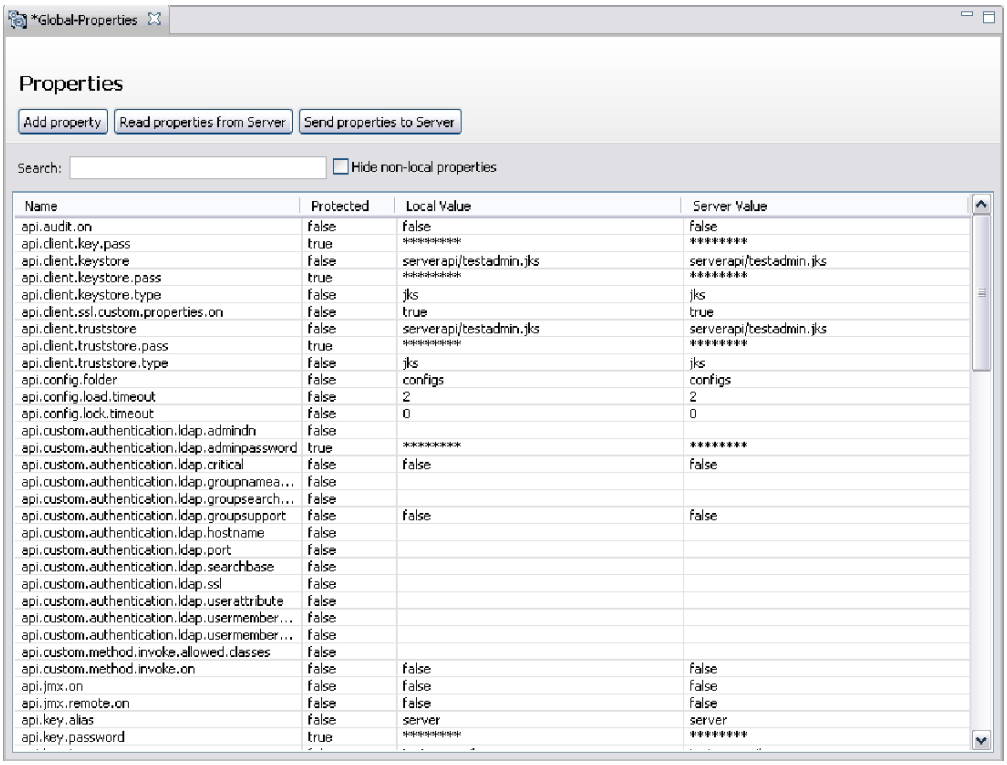


図 106. プロパティー・エディター・ウィンドウ

プロパティー・ストアからすべてのプロパティーを取得する (例えば、このストアに割り当てられたプロパティー・ファイル) には、「**ダウンロード**」を使用します。これらの値は、プロジェクトで現在選択されている IBM Security Directory Integrator サーバーから読み取られて構成エディターに渡されます。反対に、「**アップロード**」ボタンを押すと、プロパティー・ストアが (現在のサーバー経由で) エディター内の値で更新されます。ローカル値を持つプロパティーのみが更新されます。

「**検索**」テキスト・フィールドを使用して、このフィールドのテキストに一致するプロパティーを表示します。「**ローカル以外のプロパティーを非表示にします**」にチェック・マークを付けると、エディターにはローカル値を持つプロパティーのみが表示されます。

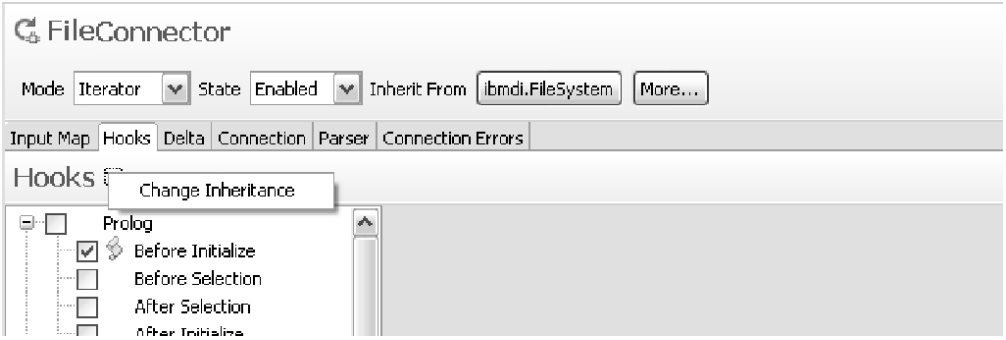
プロジェクト・ビルダーにより、実行可能構成ファイルにプロパティー・ストア構成が組み込まれます。ただしこの文書のプロパティー値は、必要な場合にのみ転送されます。

注: IBM Security Directory Integrator Server でのプロパティー・ストアの初期化とアクセスの順序は定義されていません。したがって、他のプロパティー・ストアのアクセス・パラメーター (filenames など) を定義するプロパティーを、プロパティー・ストアに信頼性のある方法で格納することはできません。

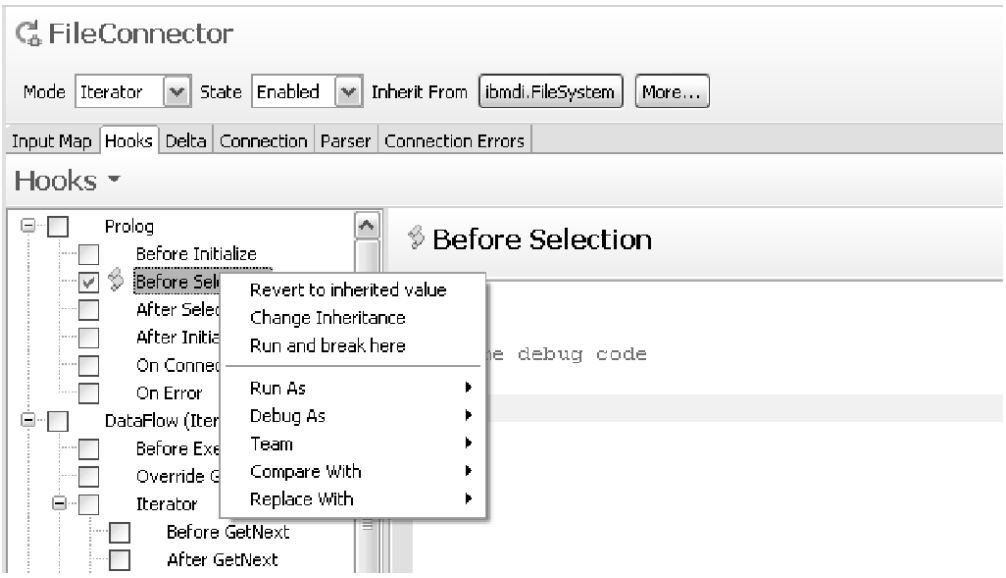
継承

コンポーネントは、コンポーネントのタイトル・バーまたはコンポーネントのサブセクションにオブジェクトをドラッグ・アンド・ドロップすることによって、ほかのコンポーネントからエレメントを継承することができます。

サブセクションの場合、タイトル・バーで使用可能なメニュー選択 (「**継承の変更**」) もあります。



フックおよび属性マップの場合、個々の項目に別々の継承選択があり、ワークスペース内のスクリプトまたは関数コンポーネントからの継承を選択できます。

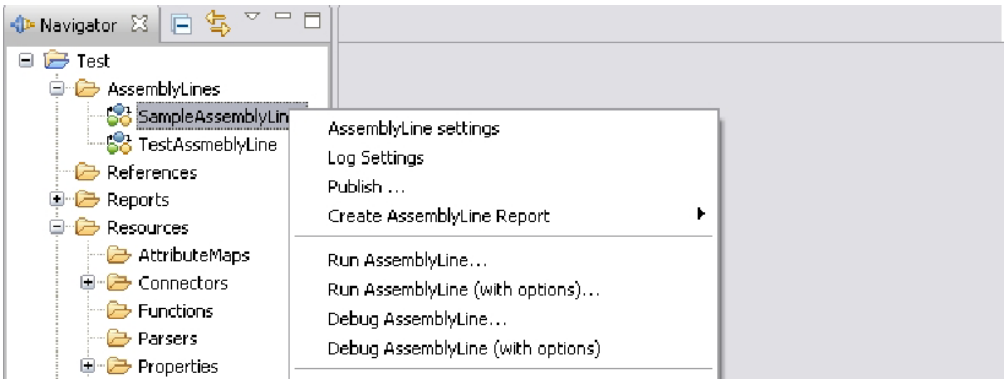


構成アイテムが継承された項目を指定変更する場合、ユーザー・インターフェースによって、ドロップダウン・メニューのアクション項目「**継承値に戻る**」を通して継承値に戻る方法が提供されます。

アクションおよびキー・バインディング

CE は、ワークベンチ内のオブジェクトだけでなく、CE 自体にもさまざまなアクションを提供します。これらのアクションは、特定のオブジェクトに対して特定の操作を実行します。

例えば、「**AssemblyLine レポートの作成**」は、`.assemblyline` という拡張子を持つファイルすべてに対して提供されるアクションです。「ナビゲーター」内で AssemblyLine を右クリックすると、ドロップダウン・メニューにこのコマンドと、このタイプのオブジェクトに対して提供されるその他のすべてのものが表示されます。



これらのアクションには、それに関連するコマンド定義もあります。コマンド定義では、ユーザーがコマンドにキーボード・ショートカットを定義することができます。コマンド定義は、次に示す 「**ウィンドウ**」 > 「**設定**」 > 「**キー**」 パネルで行います。

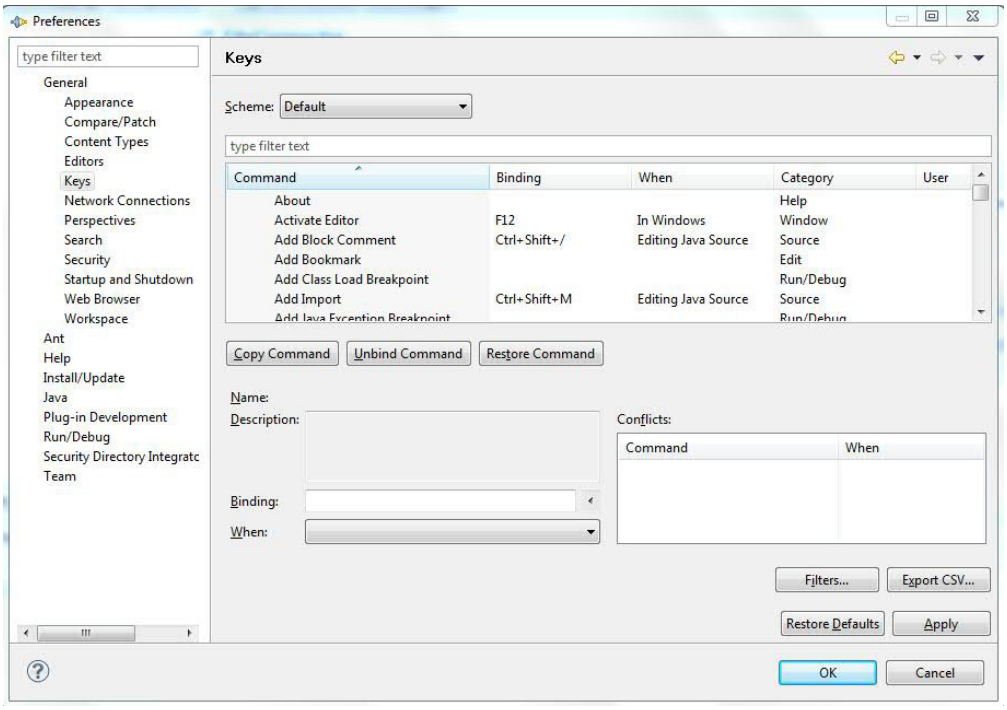
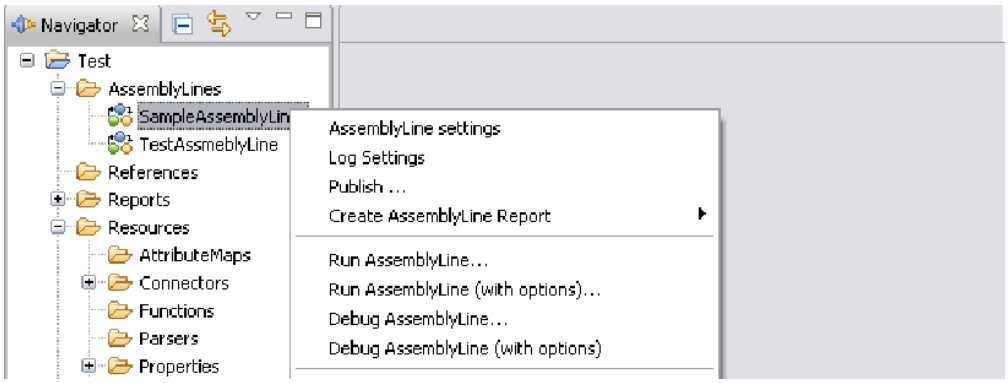


図 107. キーの割り当てウィンドウ

上の図は、ユーザー・インターフェースで行うキーボードの割り当て方を示しています。この例では、レポートの作成コマンドに、ショートカットとして `Alt+Shift+I` が割り当てられています。AssemblyLine でメニューをもう一度開くと、メニューにこの割り当てが反映されていることを確認できます。



「**スキーム**」セレクターの下にある検索フィールドに、「`security directory integrator`」というテキストを入力すると、IBM Security Directory Integrator 固有の全コマンドのリストを表示することができます。

第 4 章 IBM Security Directory Integrator のデバッグ機能

IBM Security Directory Integrator ソリューションを作成したら、通常は構成エディター (CE) を使用して、さまざまな方法でその機能を試し、テストすることができます。 IBM Security Directory Integrator が提供するデバッグ機能には、CE 内から使用できるものもあれば、IBM Security Directory Integrator インストール・ディレクトリーで使用できるスクリプトによるものもあります。

IBM Security Directory Integrator のデバッグ機能は、Sandbox、AssemblyLine のシミュレーション・モード、およびステッパーとデバッガーです。

ステッパーとデバッガーは構成エディターの一部です。 166 ページの『ステッパーとデバッガー』 を参照してください。

Sandbox

IBM Security Directory Integrator の Sandbox 機能を使用すると、AssemblyLine 内の 1 つ以上のコネクターの操作を記録できるため、後で必要なデータ・ソースが使用できない場合にも操作を再生できます。

Sandbox 機能ではシステム・ストアが使用されます。詳しくは、 215 ページの『第 6 章 システム・ストア』を参照してください。

コンポーネントを記録する場合、AssemblyLine が、コンポーネントにより使用されるコネクター・インスタンスの呼び出しをすべてインターセプトします。例えば、コネクター・コンポーネントを記録すると、コネクター・インスタンス・メソッド (`selectEntries`、`getNextEntry` など) の呼び出しがすべて記録されます。各呼び出しの結果は、構成されている Sandbox データベースに記録されます。記録される情報は、戻り値またはコネクターからスローされた例外です。

テスト・セッションを実行するには、AssemblyLine を作成し、ファイルからデータを読み取るファイル・システム・コネクターを追加します。作業項目をダンプするスクリプト・コンポーネントを追加します。次に、このパネルでファイル・システム・コネクターにチェック・マークを付け、「実行」ボタンのメニューから「**実行/記録**」を選択します。この操作が終わったら、読み取ったファイルを移動して、AssemblyLine をプレイバック・モードで実行できます。この場合ファイルがアクセス可能でないために、通常ファイル・コネクターは異常終了しますが、同じログ出力を確認できます。

この機能は、サポート資料を作成する場合に役立ちます。 AssemblyLine の環境とデータ・ソースの状態を再現して条件を再現するまでにかかる時間は、非常に広範囲にわたることがあります。記録されたセッションが保管されている Sandbox データベースを使用することで、サポート担当者は、AssemblyLine で必要なすべてのデータ・ストアに対するアクセス権がなくても AssemblyLine を実行できます。また、AssemblyLine 構成を変更して、必要に応じて詳細な情報を出力することもできます。 AssemblyLine 構成に加えることができない変更は、追加呼び出しの実行と、記録されたコンポーネントの呼び出しの順序の変更のみです。このような変更

を行うと、コネクターの呼び出しが、次に予期されているコネクター呼び出しと一致しないために、再生中にエラーが発生します。

AssemblyLine を記録または再生する前に、IBM Security Directory Integrator に対して AssemblyLine 記録データの格納場所を指示する必要があります。これは「Sandbox」ウィンドウで実行します。このウィンドウを表示するには、AssemblyLine エディターの「**AssemblyLine 設定...**」 > 「**Sandbox 設定**」を選択します。このウィンドウの上部には「**データベース**」というラベルの付いたフィールドがあり、ここに使用するシステムのディレクトリー・パスを入力できます。

サーバー・モードのコネクター、またはデルタが有効なイテレーター・コネクターを含む AssemblyLine では、Sandbox 機能はサポートされません。この機能が検出されると、サーバーは AssemblyLine の実行を中止します。

AssemblyLine 入力の記録

コンポーネントを記録する場合、AssemblyLine が、コンポーネントにより使用されるコネクター・インスタンスの呼び出しをすべてインターセプトします。例えば、コネクター・コンポーネントを記録すると、コネクター・インスタンス・メソッド (`selectEntries`、`getNextEntry` など) の呼び出しがすべて記録されます。各呼び出しの結果は、構成されている Sandbox データベースに記録されます。記録される情報は、戻り値またはコネクターからスローされた例外です。

テスト・セッションを実行するには、AssemblyLine を作成し、ファイルからデータを読み取るファイル・システム・コネクターを追加します。作業項目をダンプするスクリプト・コンポーネントを追加します。次に、このパネルでファイル・システム・コネクターにチェック・マークを付け、「**実行**」ボタンのメニューから「**実行/記録**」を選択します。この操作が終わったら、読み取ったファイルを移動して、AssemblyLine をプレイバック・モードで実行できます。この場合ファイルがアクセス可能でないために、通常ファイル・コネクターは異常終了しますが、同じログ出力を確認できます。

AssemblyLine 記録の Sandbox 再生

AssemblyLine が Sandbox モードの場合は、プレイバック用に設定されたすべてのコネクターは*仮想モード* になります。これは、コネクター・インターフェースの操作 (`getNext()` や `findEntry()` など) が実際には呼び出されないことを意味します。実際に呼び出す代わりに、これらの操作はプレイバック中にシミュレートされます。

AssemblyLine を Sandbox プレイバック・モードで実行するには、AssemblyLine の「**Sandbox**」設定ウィンドウにある該当する「**プレイバック可能**」チェック・ボックスを使用して、仮想モードで実行するコネクターを選択する必要があります。

注: 記録されたすべてのコネクターをプレイバック可能にする必要はありません。コネクターがライブ・データ・ソースにアクセスできるように設定することができますが、これはプレイバック操作の結果に影響することがあります。

コマンド行から AssemblyLine を実行するには、**-q2** スイッチを指定してサーバーを始動します。 Sandbox モードの AssemblyLine は、記録済みのデータから得られる入力 (初期作業項目を含む) で実行されます。例えば、Sandbox モードの

AssemblyLine に Java Messaging Service (JMS) コネクターが含まれている場合は、JMS コネクターが以前に記録されたデータから入力を取得し、実際には決して初期化されることはありません。

AssemblyLine の記録時に、サーバーにより、指定された「**データベース**」ディレクトリーに Derby データベースが作成され、データベース名として AssemblyLine 名が使用されます。このデータベースには、AssemblyLine の各コネクターの表があります。Sandbox モードの AssemblyLine の 1 つ以上の仮想コネクターを置換するには、記録されたコネクターの名前を変更してから、新規コネクターを元の名前で追加します。

AssemblyLine シミュレーション・モード

AssemblyLine は、AssemblyLine シミュレーション・モードを使用すれば、接続システムとの間で実際にデータを交換せずにデバッグできます。AssemblyLine がこのモードで開始されている場合、ターゲット・システムを変更する可能性があるすべての AssemblyLine コンポーネントがスキップされます。

つまり、AL は通常どおりに実行されますが、更新、デルタ、削除、または AddOnly モードのコネクターは実際の操作を実行しません。

注: このような AssemblyLine 実行では、通常モードで行われる動作に似た動作が実行されるだけです。多くの場合、シミュレーション・モードでは接続システムが更新を受け取らないので、ビジネス・ロジックが異なる動作を実行し、シミュレーションの有用性が打ち消される可能性があります。

シミュレーションの状態と、コンポーネントの状態を混同しないでください。コンポーネントの状態は、コンポーネントのシミュレーションの状態よりも優先度が高くなります。

コンポーネントの状態の値は、「使用可能」と「使用不可」です。「使用可能」状態のコンポーネントの場合、適切な操作が呼び出されると初期化が実行され、シミュレーションの状態が検査されます。状態が「使用不可」に設定されている場合、操作は呼び出されないためシミュレーション状態は検査されず、コンポーネントの初期化も実行されません。

コネクターと FC には、「受動」というもう 1 つの状態があります。状態が「受動」に設定されている場合、初期化のみが実行されます。操作はユーザー・スクリプトによってのみ実行されます。特定の操作が呼び出されるときに、シミュレーションの状態が検査されます。

AssemblyLine をシミュレーション・モードで開始する方法は複数あります。

構成エディターから開始する:

「実行モード」ドロップダウン・メニューに、AL のシミュレーションを使用可能または使用不可にするチェック・ボックスがあります。実行中に AL をシミュレートするため、ドロップダウン・メニューから使用する実行モード「オプションを指定して実行」を選択し、「オプション」ポップアップ・ダイアログ・ボックスで「**シミュレーション・モード**」チェック・ボックスにチェック・マークを付けます。デフォルトではこのチェック・ボックスにはチェック・マークが付いていません。

サーバー始動コマンド ibmdisrv を使用して開始する:

このコマンドではスイッチ **-M** が認識されます。このスイッチが指定されている場合、シミュレーションがオンの状態で AL が開始されます。

API を使用して開始する:

AL をシミュレーション・モードで実行するには、TCB オブジェクトでプロパティー `AssemblyLine.TCB_SIMULATE_MODE` を true に設定する必要があります。次に、このオブジェクトを `startAssemblyLine(String, TaskCallBlock)` メソッドに指定する必要があります。プロパティーが設定されていない場合は、デフォルトで false が設定されているものと解釈されます。

AssemblyLine をシミュレーション・モードで実行するため、新規構成が作成されます。これは AssemblyLine 構成の子です。この構成オブジェクトには、プロキシー AssemblyLine (ProxyAL) として使用される AL への接続を構成するパラメーターが含まれています。SimulationConfig オブジェクトには、シミュレーション対象 AssemblyLine のコンポーネントの状態に基づいて ProxyAL のテンプレートを作成または更新するメソッドが含まれています。SimulateConfig オブジェクトには、シミュレーション状態が「スクリプト化されている」のコンポーネントに対して定義されているすべてのフックも含まれています。各フックの名前は、シミュレーション状態が「スクリプト化されている」のコンポーネントの名前と同一です。各コンポーネントのシミュレーション状態も含まれています。

シミュレーション・モードで実行する AssemblyLine を詳細に構成できます。関連する設定を構成するには、「AssemblyLine エディター」ウィンドウで「**AssemblyLine 設定**」 > 「**シミュレーション設定**」を選択します。

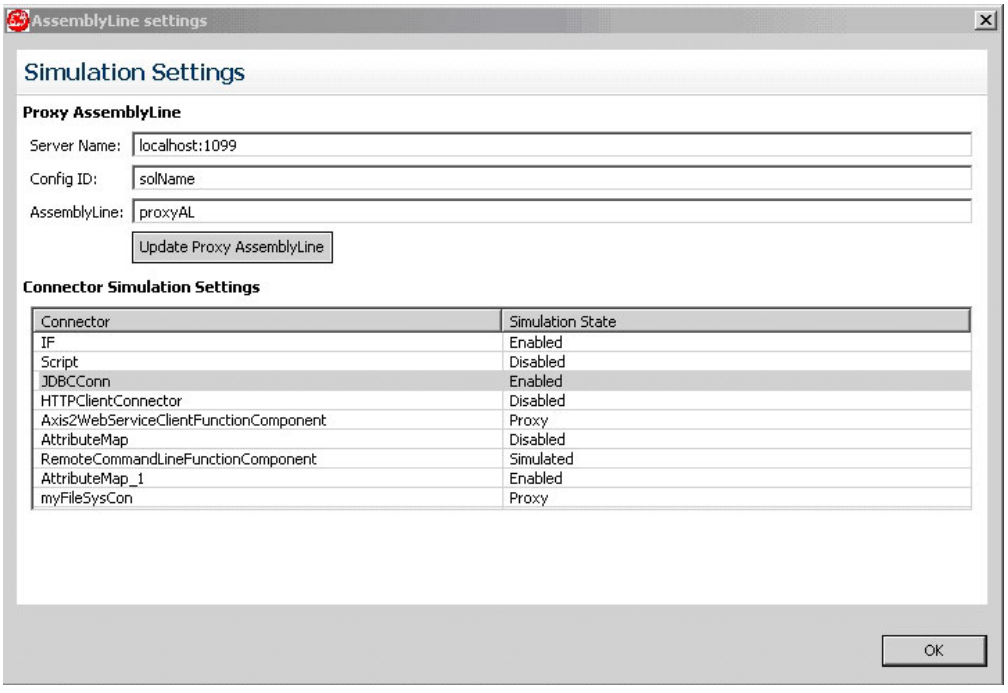


図 108. 「シミュレーション設定」ウィンドウ

「シミュレーション設定」ダイアログ・ボックスで、各コンポーネントのシミュレーション状態を構成します。このダイアログ・ボックスでは、シミュレーションの状態が「*ProxyAL*」に設定されているコンポーネントが使用する ProxyAL を構成します。「**プロキシー AssemblyLine の更新**」をクリックすると、構成エディターにより現行プロジェクトに新規 ProxyAL が作成されるか、または既存のプロキシー AssemblyLine が更新されます。作成または更新された ProxyAL はテンプレートとして提供されます。この ProxyAL の構造は、「シミュレーション設定」ダイアログ・ボックスで設定した構成に基づいています。この処理では、プロキシー AssemblyLine の名前のみが考慮されます。サーバー名と構成 ID は、シミュレートする AssemblyLine の実行中に考慮されます。ダイアログ・ボックスに指定した名前の AssemblyLine が既に存在する場合は、新規ブランチのみが追加され、古いブランチの更新や削除は行われません。これは、古いブランチの一部にユーザー固有の構成が含まれている可能性があるためです。AssemblyLine の個々のコンポーネントの状態は、以下のいずれかに設定されます。

使用可能

これは、AL で通常モードでこのコンポーネントを実行する (コンポーネントを通常どおりに実行する) 操作に相当します。

使用不可

これは、コンポーネントを使用不可にする操作に相当します (つまり、コンポーネントの初期化以外の操作やフックなどが一切実行されません)。

シミュレートされている

一般に、以降で説明するすべてのコンポーネントの統計には、AL がシミュレーションされていない場合に完了する可能性がある操作に関する情報が含まれます。シミュレーション中には重要な操作は実行されないため、重要な操作の実行結果 (正常終了またはエラー) を予測することはできません。このため、統計には操作が正常に完了したと示されます。

- AddOnly モードのコネクター: 通常どおりに実行されますが、安全でない可能性がある `connector.putEntry()` メソッドの呼び出しと `override_add` フックの呼び出しがスキップされます。
- 更新モードのコネクター: 通常どおりに実行されますが、安全でない可能性がある `connector.modEntry()` メソッドの呼び出しと `override_modify` フックの呼び出しがスキップされます。
- 削除モードのコネクター: 通常どおりに実行されますが、安全でない可能性がある `connector.deleteEntry()` メソッドの呼び出しと `override_delete` フックの呼び出しがスキップされます。
- デルタ・モードのコネクター: 通常どおりに実行されますが、このコネクターは上記のメソッドの呼び出しに依存しているため、上記のメソッドとフックがスキップされる場合にシミュレーションされます。
- デルタ・タグ付けを使用するイテレーター・モードのコネクター: 通常どおりに実行されます。これらのコンポーネントの場合、変更は前述のコネクターと似ています。つまり、安全でない可能性がある BTree クラスのメソッド (`putEntry`、`modEntry`、`deleteEntry`) の呼び出しはスキップされます。CDDeltaTaskComponent では、コミット状態がオーバーライドされ、コミットが使用不可にされます (「自動コミットなし」)。ただし `CSDeltaTaskCoponent#commitDeltaState()` メソッドを明示的に呼び出すユーザーの場合、コミットを引き続き実行できます。

- 関数コンポーネント (FC): 通常どおり実行されますが、`before_functioncall`、`after_functioncall`、および `no_reply` フックの呼び出しと、`function.perform()` メソッドの呼び出しは使用不可になります。これらのフックは、使用不可の perform メソッドから戻されるオブジェクトに関連付けられているために、呼び出しが使用不可になります。実行される唯一の操作は、FC がこの操作を正常に実行したことを示すように統計を変更する操作のみです。

  入力マップと出力マップも実行されますが、InputMap の前の実際の項目は空です (これにより、例外がスローされます。このため、関数コンポーネントから戻される属性を使用する InputMap からの各単純属性マッピングを使用不可にするか、または拡張属性マップの取得された属性の妥当性検査を追加する方法の方が適切です。あるいは、NullBehaviour メカニズムを使用して、発生する可能性があるエラーをオーバーライドすることもできます)。
- その他のモードのコネクターは通常どおり実行されます。

プロキシー

このシミュレーション状態のコネクターは、(「シミュレーション」タブに指定されている) 別の AL を開始します。これにより、安全でない可能性がある操作の実行がオーバーライドされます。この外部 AL は、このシミュレーション・モードのすべてのコンポーネントにより共用されます。これは、シミュレーション対象コンポーネント名と一致する名前の別の操作で実行されます。 203 ページの『「プロキシー AssemblyLine」ワークフロー』を参照してください。

スクリプト化されている

ユーザー定義スクリプトにより、安全でない可能性のある操作がオーバーライドされます。 AL がコンポーネント間で共用されるプロキシー状態とは異なり、このシミュレーション・モードの各コンポーネントには、固有のフックがあります。 205 ページの『シミュレーション・スクリプト・ワークフロー』を参照してください。

AL がシミュレート中であるかどうかを確認し、以下のメソッドを使用してシミュレーションのオンとオフを切り替えることができます。

- `boolean AssemblyLine#isSimulating()`: フックから task.isSimulating() のように使用されます。
- `void AssemblyLine#setSimulating(boolean)`: フックから task.setSimulating(true) のように使用されます。

各コンポーネントの状態を確認し、以下のメソッドを使用して状態を動的に設定することができます。

- `String AssemblyLineComponent.getSimulatingState()`: フックから ConnectorName.getSimulatingState() のように使用されます。
- `void AssemblyLineComponent.getSimulatingState(String)`: フックから ConnectorName.setSimulatingState("Proxy") のように使用されます。

注: 先に説明したように、すべてのシミュレーション状態が使用されるのはコネクターと FC のみです。その他のコンポーネント (およびサーバー・モードのコネクター) で使用される状態は「使用可能」と「使用不可」のみです。

IBM Security Directory Integrator では、一部の FC が安全であると認識され、そのデフォルトのシミュレーション状態が「使用可能」であると想定されます。つまり、デフォルトではこのような FC はシミュレーションされず、通常どおりに実行されます。これらの FC は単にどのシステムにも接続しないため、ターゲット・システムを変更することはできません。安全な FC を次に示します。

- CBE FC
- JavaToXML FC
- XMLToJava FC
- SDOToXML FC
- XMLToSDO FC
- パーサー関数コンポーネント
- MemQueue FC
- JavaToSOAP FC
- SOAPToJava FC
- WrapSOAP FC

重要: 明示的なコーディングにより、基盤となるデータ・システムを変更することができますが、これは、シミュレーション・モードの適用範囲外です。

上記のリストにないその他の FC は安全でない可能性があるものとして見なされており、デフォルトのシミュレーション状態は「シミュレートされている」に設定されます。

「プロキシー AssemblyLine」ワークフロー

シミュレーション・モードでのプロキシー AL の呼び出しの仕組みは、AssemblyLine コネクターを使用して必要な AL を実行する方法に似ていますが、大きな違いがいくつかあります。

コンポーネントのシミュレーションの状態が「プロキシー」の場合、特定の操作の指定変更フックの呼び出しは使用不可になっています。

コネクターが以下のいずれかのモードの場合、およびコンポーネントが関数コンポーネントの場合に呼び出されるメソッドを次の表に示します。

表 10. モード別呼び出しメソッド

モード	メソッド
コネクター: AddOnly	putEntry
コネクター: 更新	findEntry、modEntry、putEntry
コネクター: 削除	findEntry、deleteEntry
コネクター: デルタ	findEntry、modEntry、putEntry、deleteEntry
コネクター: イテレーター	selectEntries、getNextEntry
コネクター: CallReply	queryReply

表 10. モード別呼び出しメソッド (続き)

モード	メソッド
コネクター: ルックアップ	findEntry
関数コンポーネント	perform

特定のメソッドの呼び出しが着信し、コンポーネントのシミュレーションの状態が「プロキシー」の場合、メソッドの呼び出しはプロキシー AL に委任されます。プロキシー AL の開始時に、操作項目 (op-entry) と次の属性がプロキシー AL に渡されます。

- $operation - この属性には、実行する属性の名前が含まれています。コンポーネントの特定のメソッドではなくプロキシー AL が呼び出されると、この属性の値はコンポーネント名と同一になります。
- $method - この属性には、通常呼び出されるメソッドの名前が含まれています。ただし、コンポーネントのシミュレーション状態は「プロキシー」であり、一部のモード (更新など) のコンポーネントは変更 (findEntry) を実際に実行する前に複数のメソッドを実行するため、プロキシー AL はインプリメントするメソッドを認識している必要があります。この $method 属性によって、実行するメソッドがプロキシー AL に通知されるため、プロキシー AL は操作を適切に処理できます。
- search - $method 属性が findEntry の場合にこの属性が使用可能になります。この属性の値は SearchCriteria タイプのオブジェクトであり、ユーザーによって定義された検索基準を表します。例えばコンポーネントのシミュレーションの状態が「プロキシー」であり、コンポーネントのモードが「更新」の場合、ターゲット・システムの正しい項目を更新できるようにするため、検索基準が定義されている必要があります。シミュレーションの状態が「プロキシー」であるため、変更操作の前に実行される実際のルックアップ操作は、その実行をプロキシー AL に委任します。プロキシー AL はこの検索基準を使用して適切なルックアップ・シミュレーションを実行します。
- current - $method 属性が modEntry の場合にこの属性が使用可能になります。この属性の値は、実際の変更の前にターゲット・システムで検出された項目を表す項目オブジェクトです。これは、modEntry メソッドの前に実行される findEntry メソッドから戻される項目です。

初期作業項目 (IWE) は、プロキシー AL の呼び出し時にこの AL に渡されます。$method が findEntry、selectEntry、または getNextEntry の場合、IWE は呼び出し側 AL の作業項目のコピーです。その他の場合は、IWE は OutputMap プロシージャーから取得される項目 (*conn* 項目) です。 deleteEntry 操作の場合は特に、IWE は直前の findEntry 操作から取得される項目です。

プロキシー AL が実行され、$method が findEntry の場合、結果の項目に属性 *conn* があるかどうかが確認されます。この項目が使用可能な場合、この属性には findEntry 操作で検出されたすべての項目が含まれており、その値に基づいて適切なフック (no_match および multiple_match) が呼び出されます。conn という名前の属性が見つからない場合、プロキシー AL 実行結果の項目は、findEntry $method により検出された項目として扱われます。selectEntries $method をオーバーライドするプロキシー AL から取得された項目は、呼び出し側 AL の作業項目と自動的にマージされます。項目が必要なメソッド (findEntry、 getNextEntry、queryReply、および

perform) をシミュレートするプロキシー AL から取得された項目は、定義されている InputMap に送信されます。結果を戻さないその他のすべてのメソッドでは、プロキシー AL の項目は無視されます。

シミュレーション・スクリプト・ワークフロー

シミュレーションの状態が「スクリプト化されている」に設定されている各コンポーネントには、「シミュレート」タブでシミュレーション・スクリプト (SS) が定義されています。

SS から直接使用できるように公開されているオブジェクトのリストを次に示します。

- *work* - 作業項目。
- *conn* - ルックアップ操作から取得された項目、OutputMap から直接取得された項目、または NULL。
- *resEntry* - シミュレートする操作で結果が戻されている必要がある場合に、結果として使用される項目。それ以外の場合、つまり結果が使用されない場合は NULL。
- *current* - ターゲット・システムで検出された変更対象の項目、または NULL。
- *search* - ユーザー定義の SearchCriteria オブジェクトまたは NULL。
- *method* - SS により指定変更されるメソッドの名前が含まれているストリング・オブジェクト。

シミュレートされるメソッドと、SS が呼び出されるモードを次の表に示します。

表 11. モード別呼び出しメソッド

モード	メソッド
コネクター: AddOnly	putEntry
コネクター: 更新	modEntry、項目が検出されなかった場合は putEntry
コネクター: 削除	deleteEntry
コネクター: デルタ	modEntry、putEntry、または deleteEntry (内部ロジックに基づく)
コネクター: イテレーター	getNextEntry
コネクター: CallReply	queryReply
コネクター: ルックアップ	findEntry
関数コンポーネント	perform

シミュレーションの状態が「スクリプト化されている」のサーバー・モード・コネクターは、クライアントから要求を受信する必要があります。応答が送信される時点で SS が呼び出されます。

更新、削除、およびデルタ・モードのコネクターに対して実行される内部ルックアップ操作は内部で実行されます。プロキシー AL の場合のように SS からの指定変更は許可されていません。

このシミュレーション状態のコネクターは、操作の指定変更フックがある場合、指定変更フックを実行しません。

第 5 章 Easy ETL

構成エディターの EasyETL パースペクティブは、IBM Security Directory Integrator の構成を分かりやすく表示する特別なパースペクティブです。データをデータベースの形で抽出、変換、およびロード (ETL: Extract, Transform and Load) する単純なタスクを中心とするプロジェクトでお客様が素早く稼働できるようにするための専用パースペクティブです。

EasyETL パースペクティブには EasyETL プロジェクトが表示されており、新規 ETL プロジェクトの実行、オープン、および作成がきます。ETL パースペクティブは、「**ウィンドウ**」 > 「**パースペクティブのオープン**」を選択して、Easy toETL パースペクティブを選択すると表示されます。

このパースペクティブで CE を開始するには、CE のコマンド行 (`ibmditk`) に `-perspective com.ibm.tdi.rcp.perspective.etl` を追加します。

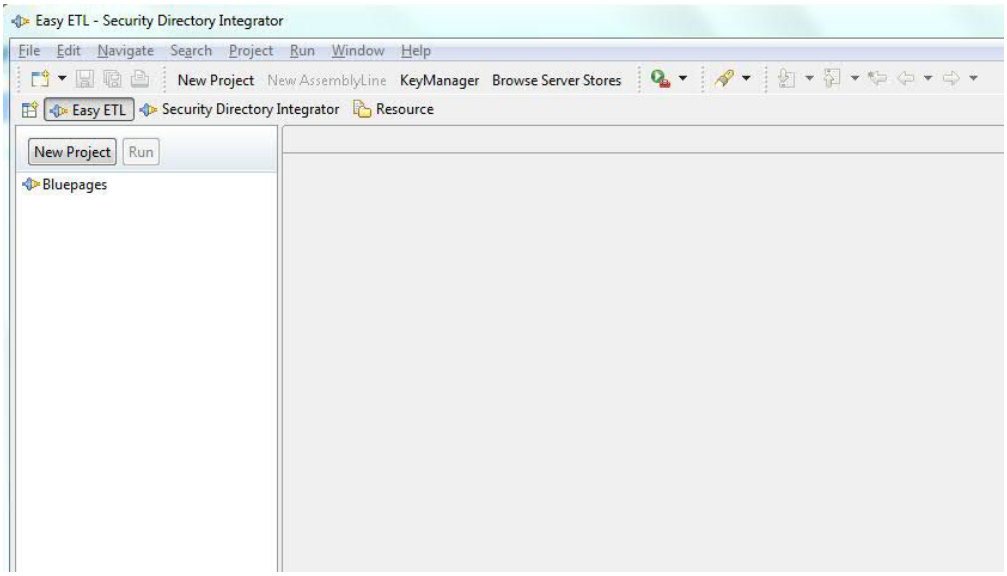


図 109. 「EasyETL」メインウィンドウ

「**新規プロジェクト**」ボタンを使用して、新しい EasyETL プロジェクトを作成します。EasyETL プロジェクトは、単一の AssemblyLine と 2 つのコネクターを使用する通常の IBM Security Directory Integrator プロジェクトです。プロジェクトをダブルクリックするか、選択して Enter キーを押すと、ETL エディターが開きます。

ETL プロジェクトのコンテキスト・メニューには次の項目があります。

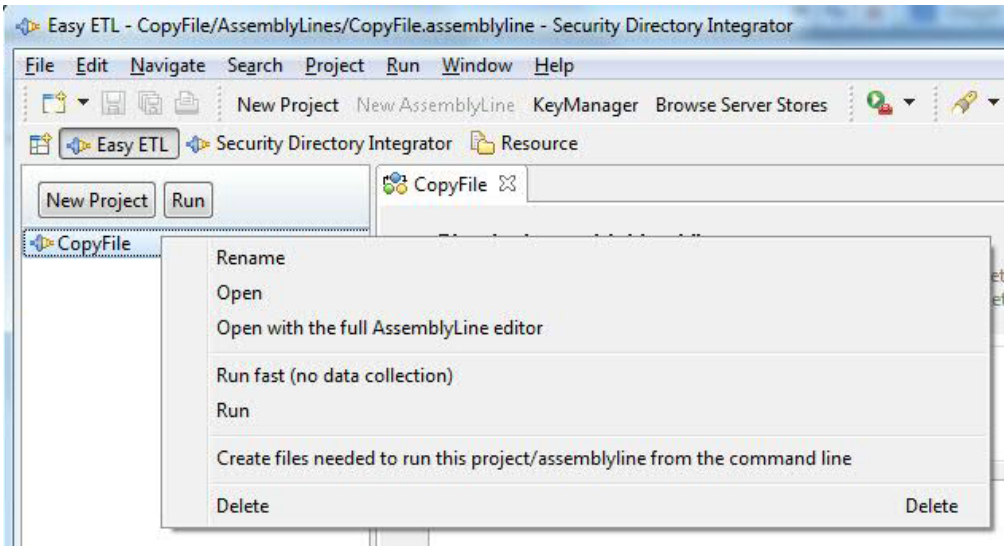


図 110. EasyETL プロジェクトのコンテキスト・メニュー

- 「**オープン**」 - エディターでプロジェクトを開きます。
- 「**フル機能の AssemblyLine エディターで開く**」 - 高機能な AssemblyLine エディターでプロジェクトを開きます。
- 「**高速実行**」 - AssemblyLine からのデータ収集なしでプロジェクトを実行します。
- 「**実行**」 - プロジェクトを実行して、収集したデータをデータ・コレクター・ビューで表示します。
- 「**ファイルの作成**」 – プロジェクトをコマンド行から実行するために必要なファイルを生成します。
- 「**名前変更**」 - プロジェクトの名前を変更します。
- 「**削除**」 - プロジェクトを削除します。

EasyETL エディターには、コネクター間のマッピングを示すテーブルとともに 2 つのコネクターが表示されます。初期画面では、次の図に示すように、両方のコネクターに何も選択されていない状態で表示されます。

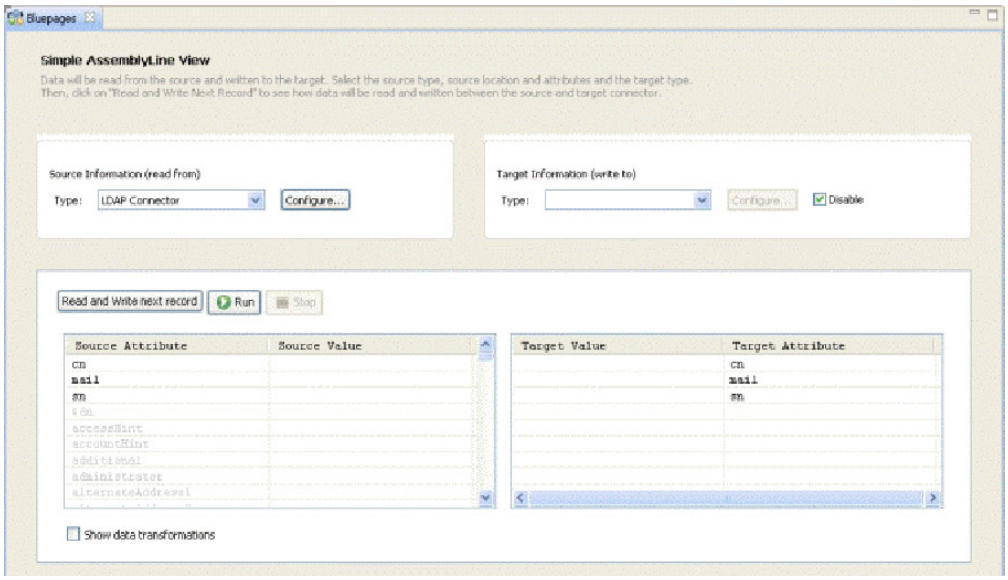


図 111. EasyETL プロジェクトの初期ウィンドウ

次に、通常はソース・コネクターを先に選択します。「タイプ」ドロップダウンには、4 つのオプションがあります。

- ファイル・システム・コネクター
- LDAP コネクター
- データベース・コネクター (JDBC)
- コネクターの選択...

上部 3 つは一般的に使用されるコネクターであるため、素早く選択できるようになっています。最後のオプション、「**コネクターの選択**」により標準コネクターの選択ダイアログが表示され、必要なコネクターを選択できます。ただし、コネクターのリストは、ソース (イテレーター) およびターゲット (AddOnly) コネクターのモードを実装しているコネクターに限定されます。

コネクターを選択したら、コネクターを構成できます。構成ダイアログでは、コネクターの構成に使用するフォームと、必要に応じてパーサーが利用できます。また、ソース・コネクターでは「デルタ」画面が利用できます。

LDAP コネクターを選択すると、次のようなウィンドウが表示されます。

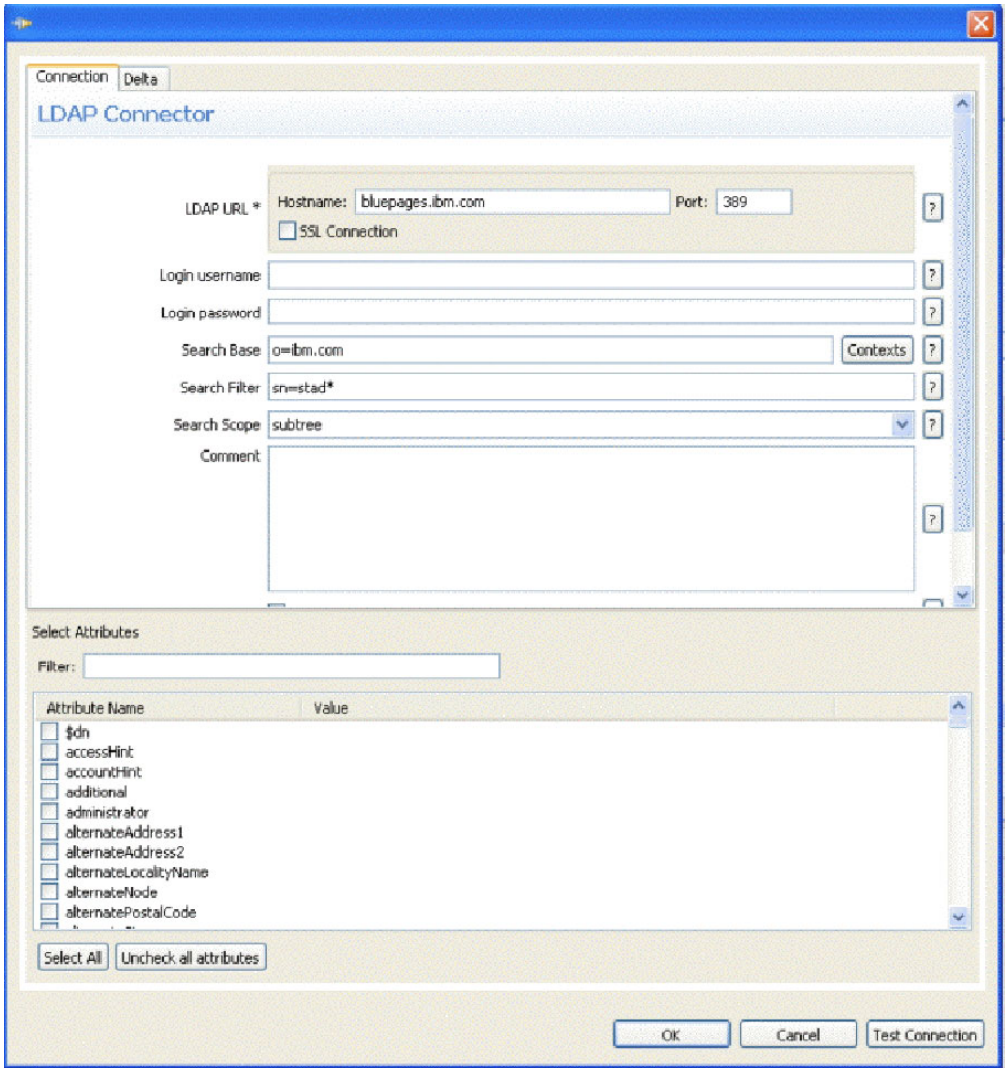


図 112. Easy ETL での LDAP コネクター

コネクターの属性の検出後に、コネクターから読み取る属性にチェック・マークを付けます。ターゲット・コネクターでは、入力コネクターの属性を基にマッピングが行われるため、スキーマ項目のリストしか表示されません。

スキーマおよび属性が使用可能になると、ソース属性列にそれらのスキーマおよび属性が表示されます。

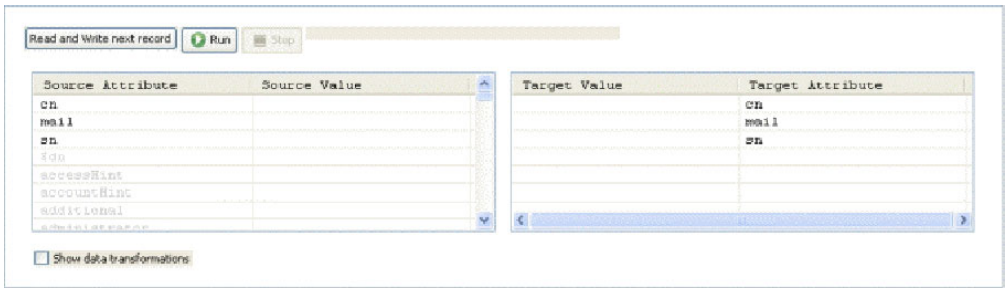


図 113. 入出力マッピング

ぼかし表示された項目は、マップされていない属性です。これらの属性をマップするには、右クリックして「**属性のマップ**」を選択します。また、現在の選択項目をダブルクリックするか、または Enter キーを押してもマップできます。ターゲット属性列では、別の出力属性名をクリックして選択できます。反対に、マップ済みの属性に同じ動作を行って、マップされていない属性のリストに戻すことができます。

2 つの属性間のマッピングをカスタマイズするには、「**データ変換の表示**」チェック・ボックスにチェック・マークを付けます。これにより、ソース・テーブルとターゲット・テーブルの間に新しいテーブルが追加されます。

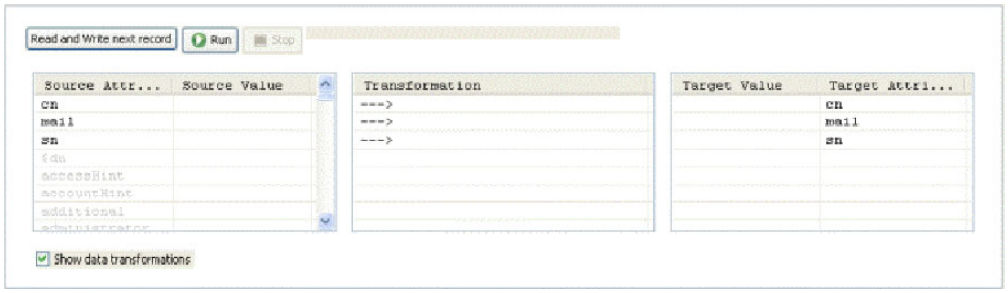


図 114. 入出力マッピング、データ変換の表示あり

表示されている変換テーブルには、2 つの属性間の逐語的なコピーを示す矢印が表示されています。変換項目をダブルクリックすると、そのマップ用の JavaScript エディターが起動します。

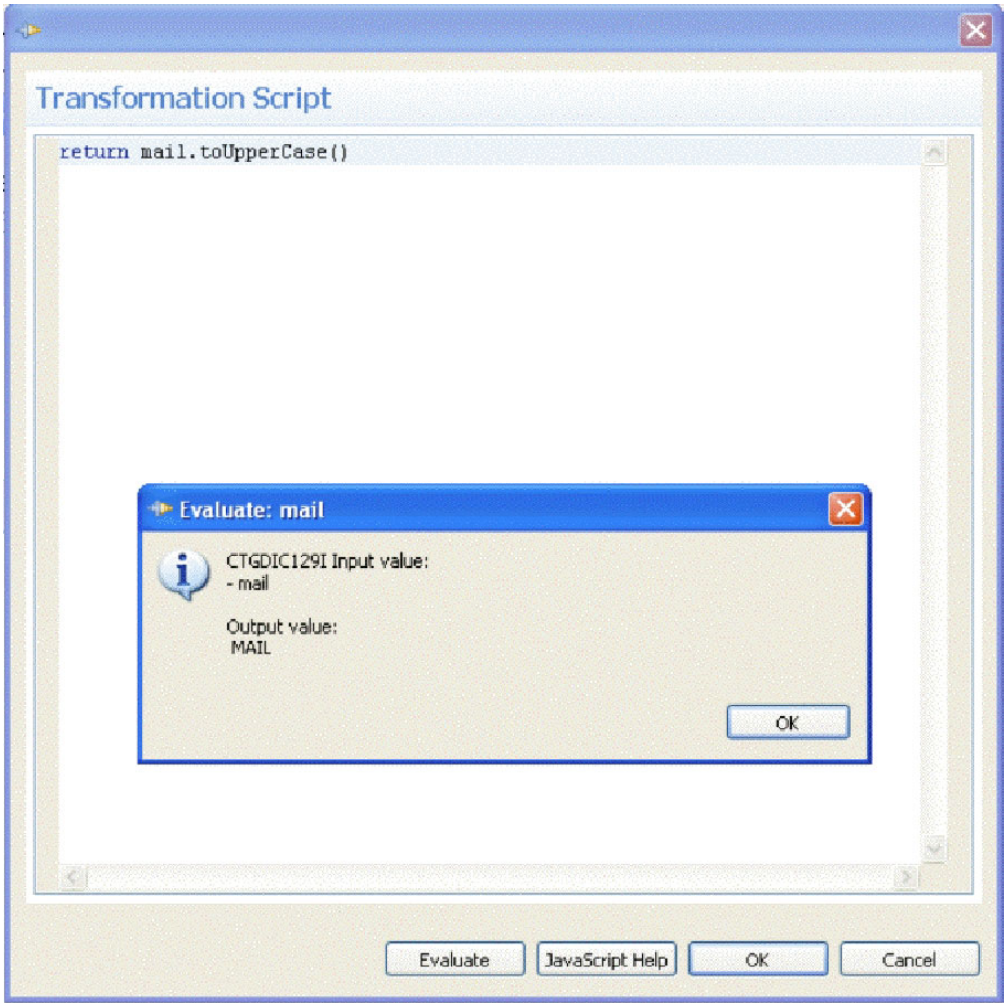


図 115. 変換スクリプト

値のカスタム変換を行うスクリプトを入力します。ソース・コネクターのマップ済み属性はすべて、最上位の Bean として利用できます。つまり、`work.cn` の表記を使用する代わりに、直接 `cn` を参照できます。また、JavaScript エディターは、「*次のレコードの読み取りおよび書き込み*」操作により読み取られた内容を基に、Java クラスを認識します。最後に読み取られた項目は、「**評価します**」ボタンでスクリプトをテストする場合にも使用されます。これにより、上の図に示すように、スクリプトの評価が、実際のライブ・データに対してテストされます。メッセージには、入力値と変換スクリプトの結果 (出力) が示されます。「**JavaScript ヘルプ**」ボタンを使用すると、JavaScript のヘルプ・ページに素早くアクセスできます。

ターゲット・コネクターには、「**使用不可**」というラベルが付いたチェック・ボックスがあります。このチェック・ボックスにチェック・マークを付けると、出力コネクターが使用不可になり、代わりに項目を (カスタム変換後に) コンソール・ログにダンプします。

コネクターを構成したら、AssemblyLine を終わりまで実行するか、または一度に 1 レコードずつステップ実行することができます。AssemblyLine をステップ実行すると、読み取られてターゲット・コネクターに書き込まれた最新の項目がテーブルに反映されます。「**実行します**」ボタンをクリックすると、AssemblyLine が終了する

まで、または「**停止**」ボタンを押すまで、連続して実行されます。実行中に「停止」ボタンを押すと、AssemblyLine が中断されて制御が戻ります。制御が戻っている状態で「停止」ボタンを押すと、AssemblyLine が終了します。AssemblyLine が終了すると、実行に関するいくつかの統計と共に完了ダイアログが表示されます。

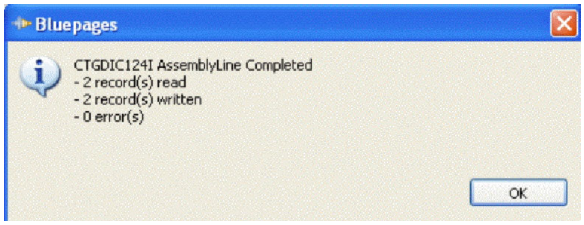


図 116. 完了ダイアログ

第 6 章 システム・ストア

システム・ストアは、基礎となるストレージ技術として Apache Derby RDBMS (以前は IBM Cloudscape) と呼ばれていた技術をデフォルトで使用し、IBM Security Directory Integrator におけるさまざまな永続ストレージのニーズに対処します。

IBM DB2 などのリレーショナル・データベースを使用してシステム・ストアを保持できます。Derby データベースをネットワーク・モードで実行する場合、またはマルチユーザー・リレーショナル・データベース・システムを使用する場合は、複数の IBM Security Directory Integrator サーバー・インスタンスがシステム・ストアを共用できます。Derby を IBM Security Directory Integrator サーバーに組み込んで実行する場合は、他のサーバーと同時に共用できません。

システム・ストアは、IBM Security Directory Integrator コンポーネントの永続ストアとして、次の 3 種類の永続ストアをインプリメントします。

- 216 ページの『ユーザー・プロパティー・ストア』
- 217 ページの『デルタ・ストア』
- Sandbox テーブル

各ストアは、ストア独自の機能と組み込み動作のセットや、ユーザー・スクリプトからアクセスできる呼び出し可能インターフェースを提供します。各ストアは、ユーザーが独自のデータや状態情報を永続化するためなどに使用できます。

プロジェクトの 187 ページの『システム・ストア設定』を構成するには、IBM Security Directory Integrator ナビゲーターでプロジェクトを選択し、「**ソリューションのロギングおよび設定**」を選択します。次に「**システム・ストア**」タブを選択します。

システム・ストア内のこれらのテーブルのいずれかに直接アクセスするように JDBC コネクターをセットアップすることもできます。ただし、ソリューションが誤動作する可能性があるため、これらのテーブル内のデータは変更しないでください。

重要: Derby をサーバーとしてネットワーク・モードで実行するのではなく、IBM Security Directory Integrator に埋め込んで実行する場合は、構成のテストまたは実行を試行する前にデータベースを必ず**クローズ**してください。構成エディターは、個別のサーバー・インスタンスを始動し、独自の JVM 内で実行されます。そのため、このサーバーではシステム・ストアを使用できません。「システム・ストア」詳細ウィンドウをクローズすると、データベースへの接続も終了します。

注: Sandbox 機能でもシステム・ストア技術を使用しますが、この場合は AssemblyLine ごとに新しいデータベース・ディレクトリーを指定します。

ユーザー・プロパティー・ストア

ユーザー・プロパティー・ストアは、キー値に関連付けられたシリアライズ Java オブジェクトを管理するために使用されるシステム・ストア・テーブルです。このストアでは、ユーザーが格納したデータと同様に、永続的なコンポーネント・パラメーターとプロパティー (「**イテレーター状態ストア**」など) も維持されます。

例えば、Active Directory 変更検出コネクターの「**イテレーター状態ストア**」パラメーターを設定するときには、コネクターがイテレーター状態を保管および復元するために使用するキー値を指定します。最初の (または最後の) 変更項目から反復処理を開始する場合は、ユーザー・プロパティー・ストア内の「イテレーター状態ストア」項目を削除します。つまり、このパラメーターの隣にある「**削除**」をクリックします。

次のシステム呼び出しを使用して、所有オブジェクトを永続化できます。

system.setPersistentObject(*keyValue***,obj)**
: 指定された *keyValue* を使用して、ユーザー・プロパティー・ストア内にオブジェクト **obj** を保管します。オブジェクトが正常に保管された場合は、オブジェクトが戻されます。それ以外の場合、関数は **null** を戻します。

system.getPersistentObject(*keyValue***)**
: 指定された *keyValue* を持つオブジェクトをユーザー・プロパティー・ストアから戻します。*keyValue* が検出されない場合、関数は NULL を戻します。

system.deletePersistentObject(*keyValue***)**
: 指定された *keyValue* を持つオブジェクトをユーザー・プロパティー・ストアから削除します。この関数は削除されたオブジェクトを戻します。*keyValue* が検出されなかった場合は NULL を戻します。

これらのメソッドは、デフォルトのユーザー・プロパティー・ストアにアクセスします。

ストア・ファクトリー を使用して独自のストアを作成および使用することもできます。

「システム・ストア」ウィンドウでユーザー・プロパティー・ストアを表示する場合は、次のテーブル定義があることに注意してください。

キー　固有キー (512 文字)

項目　キーに関連付けられたオブジェクト

注: ユーザー・プロパティー・ストア内で永続化されるすべてのオブジェクトは、シリアライズ可能である必要があります。

デルタ・ストア

デルタ・ストアは、システム・ストア・ブラウザーの「**デルタ・テーブル**」フォルダーの下にあります。各テーブルは、(イテレーターの「**デルタ**」タブ内の) 1 つの「**デルタ・ストア**」パラメーター設定を表します。さまざまなクラスやメソッドを使用してデルタ・ストアを直接操作できますが、この方法はお勧めできません。デルタ機能について詳しくは、221 ページの『第 7 章 デルタ』というタイトルのセクションを参照してください。

ストア・ファクトリー・メソッド

次の例は、ストア・ファクトリーで使用できるメソッドの例です。

public static PropertyStore getDefaultPropertyStore () throws Exception;
デフォルトのデフォルト・プロパティー・ストアを戻します。

public static PropertyStore getPropertyStore (String table) throws Exception;
名前によって識別されたプロパティー・ストアを戻します。指定された名前のインスタンスは、一度に 1 つのみ存在できます。

@param name
プロパティー・ストア名。

@return
名前に関連付けられているプロパティー・ストア・オブジェクト。

public static String getSystemDatabaseURL ();
システム・ストア JDBC URL を戻します。

public static Connection getConnection () throws Exception;
デフォルト・データベースへの接続オブジェクトを戻します。

public static Connection getConnection (String database) throws Exception;
AutoCommit が TRUE に設定されている指定されたデータベースへの接続オブジェクトを戻します。

@param database
データベース名。

public static Connection getConnection (String database, boolean autoCommit) throws Exception;
指定されたデータベースへの接続オブジェクトを戻します。

@param database
データベース名。

@param autocommit
自動コミット・フラグ。

@return
指定されたデータベースへの接続オブジェクト。

public static boolean dropTable (Connection connection, String table);
接続に関連付けられたデータベース内のテーブルをドロップします。

@param connection
getConnection() が取得した接続オブジェクト。

@param table
ドロップするテーブル。

public static void verifyTable (Connection connection, String table, Vector sql) throws Exception;
データベース内でテーブルがアクセス可能かどうかを検査します。

@param connection
getConnection() が取得した接続オブジェクト。NULL の場合はデフォルト・テーブルへの接続が取得されます。

@param table
検査するテーブル名。

@param sql
テーブルが存在しない場合にテーブルを作成するための SQL 文のベクトル。

public static Exception dropTable (String tableName);
デフォルト・データベース内のテーブルをドロップします。

@param tableName
ドロップするテーブルの名前。

public static byte[] serializeObject (Object obj) throws Exception;
オブジェクトをバイト配列にシリアライズします。

@param obj
シリアライズするオブジェクト。

@return
シリアライズド・オブジェクトを格納しているバイト配列。

public static Object deserializeObject (byte[] array) throws Exception;
バイト配列を Java オブジェクトに非シリアライズします。

@param array
シリアライズされた Java オブジェクトを格納しているバイト配列。

@return
回復された Java オブジェクト。

プロパティー・ストア・メソッド

次の例は、プロパティー・ストアで使用できるメソッドの例です。

public Object setProperty (String key, Object obj) throws Exception;
プロパティー・ストア内の値を追加または更新します。更新が実行された場合は、元の値が戻されます。

@param key
固有 ID。

@param obj
値。

@return
更新の場合は元の値。

public Object getProperty (String key) throws Exception;
プロパティー・ストア内の値を戻します。

@param key
固有 ID。

@return
ストア内の値。検出されない場合は NULL。

public Object removeProperty (String key) throws Exception;
プロパティー・ストア内の値を除去します。

@param key
除去する固有 ID。

@return
元の値。キーがテーブル内にない場合は **NULL**。

UserFunctions (システム・オブジェクト) メソッド

UserFunctions クラス (システム・オブジェクトなど) には、システム・プロパティー・ストア内でオブジェクトを取得または設定するために定義されている追加メソッドがあります。

public Object getPersistentObject (String key) throws Exception;
このメソッドは、指定されたオブジェクトをデフォルトのシステム・プロパティー・ストアから取得します。

@param key
固有キー。

public Object setPersistentObject (String key, Object value) throws Exception;
このメソッドは、指定されたオブジェクトをデフォルトのシステム・プロパティー・ストアに格納します。

@param key
固有キー。

@param value
格納するオブジェクト (Javaシリアライズ可能であることが必須)。

@return
元のオブジェクト (存在する場合)。

public Object removePersistentObject (String key) throws Exception;
このメソッドは、指定されたオブジェクトをデフォルトのシステム・プロパティー・ストアから除去します。

@param key
固有キー。

@return

元のオブジェクト (存在する場合)。

第 7 章 デルタ

デルタ・エンジン機能は、イテレーター・モードのコネクターで使用できます。イテレーターの「デルタ」タブで使用可能にすると、デルタ・エンジン機能がシステム・ストアを使用して、繰り返されているデータの「スナップショット」を取ります。読み取りに成功した各項目が、デルタ・ストアと呼ばれるスナップショット・データベースと比較され、変更されているかどうかが確認されます。読み取られた項目と、デルタ・ストアに保管された項目の間の差異に基づいて、デルタ項目と呼ばれる新しい項目がデルタ・エンジンによって作成されます。この項目には、特殊なデルタ命令コードのタグが付けられて、変更の内容と方法が示されます。

デルタ・モードは、コネクターがデルタ命令コードを「理解」して使用できるようになるコネクター・モードです。このモードのコネクターは、受け取ったデルタ項目のデルタ命令コードを使用して、接続システムに適用する必要のある変更タイプを判断します。デルタ・モードは、すべてのタイプの変更 (追加、変更、および削除) をサポートします。このモードは、異なるシステム間の同期化を円滑に行うために使用されます (例えば、異なるマシン上にある 2 つの LDAP サーバーの同期化など)。

重要: デルタ・エンジンは、同期プロセス中に変更を計算するために、データのスナップショットを保管する基本的なローカル・リポジトリーを導入しています。変更を走査する対象となるデータ・ソースが、マスター・スレーブ関係のマスターになります。この場合、スレーブ (デルタ・ストアなど) に対するすべての変更は、基本となるデータベース表を直接操作して行うのではなく、デルタ機構を使用して行う必要があります。そのようにしないと、IBM Security Directory Integrator が維持するデルタ・スナップショット情報は不整合になり、デルタ・エンジンは失敗してしまいます。

デルタ機能

ここでは、IBM Security Directory Integrator のデルタ機能について説明します。

デルタ項目

デルタ項目は、特殊なデルタ命令コードのタグが付けられた通常の項目オブジェクトです。これらのコードは変更タイプ (*追加*、*変更*、*削除*、または*未変更*) を表し、異なるレベル (Entry、Attribute、または AttributeValue レベル) で割り当てられます。

デルタ項目を生成するコンポーネント

- デルタ・エンジン: データ・ソース内の変更を検出します。これは、データ・ソース自身に変更内容に簡単にアクセスできる機能がない場合 (例えば、変更ログまたは変更通知機構) に役立ちます。変更は、データ・ソースの現在の状態と履歴スナップショットを比較することで検出されます。スナップショットは「デルタ・ストア」というリポジトリーに保存されます。物理的には、デルタ・ストアはシステム・ストア内にある多数のデルタ・テーブルからなります。

データは、次回読み取られるときに、既にデルタ・ストアに保存されているデータと比較されます。変更情報が計算された後に、コネクターによってデルタ項目が作成されて戻されます。このデルタ項目は、*更新*、*AddOnly*、*削除*、またはデルタ・モードのコネクターを使用して、検出された変更を別の接続システムに転送するために使用できます。

- 変更検出コネクター: 変更を検出するコネクターは、LDAP コネクター、RDBMS 変更検出コネクター、および Domino 変更検出コネクターです。
- LDIF および DSMLv2 パーサー: 読み取りまたは書き込み時に、LDIF および DSMLv2 パーサーは Entry、Attribute、および AttributeValue レベルでのデルタ・タグ付けをサポートします。

デルタ項目を取り込むコンポーネント

- デルタ・モード: このコネクター・モードは、システム間の同期化ソリューションを円滑化します。このモードを使用して、すべてのタイプの変更を接続システムに適用します。デルタ・モードは、デルタ項目を必要とする (および使用する) 唯一のモードです。このモードは、項目のデルタ命令コードを使用して変更タイプを検出します。
- 更新コネクター (「変更の計算」を指定): 「変更の計算」パラメーターを有効にした更新モードのコネクターは、受け取った項目にデルタ・タグが付いている場合、「変更の計算」ロジックを起動しません。

デルタ項目

デルタ項目は、通常の項目のすべての機能と関数を持つ項目です。さらに、デルタ項目にはデルタ命令コードも含まれています。デルタ命令コードは、項目に適用された変更タイプ (追加、削除、変更、または未変更) を示します。デルタ命令コードは、項目、属性、および値に追加され、これらの特定の変更を反映します。

概説

デルタ命令コードの割り当てプロセスはデルタ・タグ付けと呼ばれ、デルタ・コードは命令コードおよびデルタ・タグと呼ばれます。簡単に言えば、デルタ項目は実際にはデルタ・タグの付いた通常の項目です。次は、通常の項目とデルタ項目の例です。

通常の項目:

```
{
 "#type": "generic",
 "#count": 3,
 "UserName":
  "#type": "replace",
  "#count": 1,"tanders",
 "FullName":
  "#type": "replace",
  "#count": 1,"Teddy Anderson",
 "id":
  "#type": "replace",
  "#count": 1,"66"
}
```

デルタ項目:

```
{
 "#type": "modify",
 "#count": 3,
 "@delta.old": "{
 "UserName": "manders",
 "FullName": "Mary Anderson",
 "id": "66"
 }",
 "UserName": [
  "#type": "modify",
  "#count": 2,
  "tanders",
  "manders"
 ],
 "FullName": [
  "#type": "replace",
  "#count": 1,
  "Mary Anderson"
 ],
 "id":
  "#type": "unchanged",
  "#count": 1,"66"
}
```

デルタ・コードの評価プロセスは、上から下、または項目レベル → 属性レベル → 属性値レベルの順で進みます。高レベルでの操作が優先されます。

項目操作が*削除* の場合、その他のデルタ・タグはすべて無視されます。項目操作が*置換*、*変更*、または*追加* の場合、評価は属性デルタ・タグまで続きます。

属性に*削除*、*追加*、または*置換* のタグが付いている場合、その値のデルタ・タグは無視されます。属性に*変更* のタグが付いている場合のみ、属性値のデルタ・タグが考慮されます。これらのデルタ・タグは、属性値に異なる命令コードが存在する可能性を示します (例えば、一部は追加、一部は削除、など)。

属性値のデルタ・タグには、デルタ・タグ付けにおいて次の意味があります。

- *追加*: 特定の属性の値のリストに値が追加された。
- *削除*: 特定の属性の値のリストから値が削除された。
- *置換*: 値が置換された。これはデフォルトのデルタ・タグです。

デルタ項目は、以下のコンポーネントによって生成されます。

1. デルタが使用可能になっているイテレーター・モードのコネクター
2. 変更検出コネクター
3. LDIF および DSMLv2 パーサー (読み取り時)

デルタ項目は、以下のコンポーネントによって取り込まれます。

1. デルタ・モードのコネクター
2. LDIF および DSMLv2 パーサー (書き込み時)
3. 更新モードのコネクター

スクリプトによるデルタ命令コードの取得と設定

デルタ・タグ付けは項目、属性、および属性値レベルでサポートされています。スクリプトを使用して項目操作を取得/設定する方法を次に示します。

```
var entryOper = work.getOperation();  //get Entry operations as string (e.g. 'add')
var entryOp = work.getOp();      //get Entry operations as char (e.g. 'a')

work.setOperation("modify");      //set Entry operation
work.setOp ('m');
```

属性のデルタ・フラグを設定/取得するには、次のようなコードを使用します。

```
var attr = work.getAttribute("sn");   // get Attribute object

var attrOper = attr.getOperation();   // get delta operation as string (e.g. 'replace')
var attrOp = attr.getOper();      // get delta operation as char (e.g. 'r')

attr.setOperation("replace");     // set Attribute delta operation
attr.setOper('r');
```

AttributeValue レベルの Delta タグを次のスクリプトを使用して設定/取得できます。

```
var attr = work.getAttribute("sn");   // get Attribute object

var valOper = attr.getValueOperation(0);// get value delta operation for first value
var valOp = attr.getValueOper(0);

attr.setValueOperation(1, "add");    // set value delta operation for second value
attr.setValueOper(1, 'a');
```

デルタ項目の生成

デルタ項目は、デルタ機能、またはイテレーター・モードのコネクターの適切なパーサーを使用するか、または IBM Security Directory Integrator の変更検出コネクターによって生成されます。

特に、デルタ・タグ付けされた項目は以下のコンポーネントから戻されます。

- Active Directory 変更検出コネクター
- Domino 変更検出コネクター
- IBM Security Directory Server 変更ログ・コネクター
- RDBMS 変更検出コネクター
- Sun Directory 変更検出コネクター
- z/OS LDAP 変更ログ・コネクター

  注: IBM Security Directory Integrator バージョン 7.2 以降では z/OS オペレーティング・システムはサポートされません。
- DSMLv2 パーサー
- LDIF パーサー

イテレーター・モードのデルタ機能

イテレーター・モードのコネクターは、デルタ項目を生成できます。この機能は、デルタ・エンジンおよびデルタ・ストアを使用して変更を検出します。

デルタ・エンジン

デルタ・エンジンを使用して、データ・ソース全体を読み取って、前回の実行以降の変更 (差分) を検出することができます。この方法により、新しい項目、変更された項目、さらには削除された項目まで検出することが可能です。一部のデータ・ソース (LDIF ファイルや LDAP サーバーなど) については、IBM Security Directory

Integrator は項目内の属性および値が変更されたかどうかを検出することもできます。イテレーター・モードのコネクターでのみ、デルタ設定を構成できます。

デルタ・エンジンは、システム・ストアの一部である永続ストア内に各項目のローカル・コピーを保持することにより、どの項目または属性が追加、変更、または削除されたかを認識します。このローカル・リポジトリーは、*デルタ・ストア* と呼ばれ、*デルタ・テーブル* から構成されます。 AssemblyLine が実行されるごとに、デルタ・エンジンは反復処理されているデータと、デルタ・テーブル内のデータのコピーとを比較します。変更が検出されると、コネクターは*デルタ項目* を戻します。

注: デルタ・ストアのテーブルを手動で変更しないでください。変更すると、デルタ・スナップショットの情報が不整合になり、デルタ・エンジンが正しく機能しなくなります。

注: IBM Security Directory Integrator V6.1 より前のバージョンでは、デルタ・エンジンの処理中にデルタ・ストアに書き込まれたスナップショットが即座にコミットされていました。その結果、デルタ・エンジンは、AL フロー・セクションの処理が失敗した場合でも、変更された項目を処理されたものと見なしていました。この制限は、コネクターのデルタ・タブの Commit パラメーターで解決されます。このパラメーターの設定により、デルタ・エンジンが着信データのスナップショットをシステム・ストアにコミットするタイミングを制御できます。

固有属性名

デルタ機構が各項目を一意に識別できるようにするには、デルタ鍵として使用する固有属性を指定する必要があります。固有属性の値は、使用されるデータ・ソース内で一意である必要があります。デルタ鍵を指定するには、コネクターの「デルタ」タブで、「固有属性名」パラメーターに属性名を入力 (または選択) します。この属性は、イテレーターの入力マップに含まれている必要があります。接続されたシステムから読み取られた属性または計算された属性 (属性マッピングでスクリプトを使用) を指定できます。

複数の属性を正符号 (+) で区切って指定することもできます。

```
LastName+FirstName+BirthDate
```

固有属性名パラメーターで指定された属性のうちの少なくとも 1 つは、値を格納している必要があります。複数の属性が指定された場合は、それらのストリング値が 1 つのストリングとして連結され、固有のデルタ ID になります。値を持たない属性 (例えば、空または NULL) は、項目のデルタ鍵が生成されるときにスキップされます。

デルタ・ストア

デルタ・ストアは、物理的にはシステム・ストア内にあります。デルタ・ストアは 1 つのデルタ・システム・テーブル (DS) と 1 つ以上のデルタ・テーブルから構成されています。各デルタ・テーブルは、デルタが使用可能になっている各イテレーター・コネクターのデルタ・ストアに使用されます。

デルタ・ストア・テーブルには、JDBC コネクターおよびシステム・ストア・コネクターの両方からアクセスできますが、テーブルの構造とデルタ・エンジンによる

処理方法をよく理解していない限り、デルタ・ストア・テーブルを変更することはお勧めしません。

デルタ・テーブルの構造

各デルタ・テーブル (DT) には、特定のコネクターについてデルタ・エンジンによって処理された各項目に関する情報が格納されます。デルタ・システム・テーブル (DS) には、デルタ・ストアで現在使用されているすべてのデルタ・テーブルのリストが維持されます。

- デルタ・システム・テーブル: デルタ・システム・テーブル (DS) には、システム・ストア内の各デルタ・テーブル (DT) についての情報が含まれています。DS の目的は、各 DT のシーケンス・カウンターを管理することです。DS の構造は次のとおりです。

表 12. デルタ・システム・テーブルの構造

列	タイプ	説明
ID	Varchar	DT 識別子 (名前)
SequenceID	Int	最後の実行からのシーケンス ID
Version	Int	DS のバージョン (1)

- デルタ・テーブル: デルタ・ストアを要求する各コネクターは、コネクターと関連付けられる固有のデルタ ID を指定する必要があります。この ID は、システム・ストア内のデルタ・テーブルの名前としても使用されます。デルタ・テーブルの構造は次のとおりです。

表 13. デルタ・テーブルの構造

列	タイプ	説明
ID	Varchar	項目を識別する固有値
SequenceID	Int	項目のシーケンス番号
Entry	Long Varbinary	シリアライズされた項目オブジェクト

デルタ・プロセス

上記のデルタ・ストアの構造を前提に、シーケンス番号は、ソース・データ・セットの一部ではなくなった項目を検出するために使用されます。AssemblyLine を実行するたびに、特にコネクターによって使用されるデルタ・テーブルのシーケンス番号がデルタ・システム・テーブルから読み取られます。読み取られたシーケンス番号は増分され、増分された値が、AssemblyLine の実行の全体にわたって更新された項目のマーキングに使用されます。

デルタ・エンジンのプロセスでは 2 度の動作が行われます。

1. 読み取り → ルックアップ → 比較 → 更新 → 現行の SequenceID の設定
   a. イテレーターが入力データ・ソースから項目を読み取ります。
   b. デルタ・プロセスが、固有属性値を使用してデルタ・テーブル内でその項目に対応する項目を探します。

c. 一致が検出された場合、デルタ・プロセスは各属性 (および各属性の値) を比較して、項目が変更されているかどうかを判別します。比較の結果に基づいて、デルタ・エンジンはデルタ項目に適切な命令コード (変更 または未変更) のタグを付けて戻します。
   - 変更項目: 読み取られた項目と、デルタ・テーブル内の対応項目が異なっていると判断された。項目はデルタ・テーブルで更新される。
   - 未変更項目: 読み取られた項目と、デルタ・テーブル内の対応項目が同じと判断された。

d. デルタ・テーブル内で一致が検出されない場合、その項目は新規項目とみなされます。
   - 追加項目: 項目がデルタ・テーブルに追加される。

e. c. と d. のどちらの場合も、デルタ・テーブル内のシーケンス番号の値は、現行の AssemblyLine の実行で使用されたシーケンス番号に更新されます。

2. (SequenceID < 現行の SequenceID) となっているデータの確認 → 削除としてマーク

   イテレーターがデータの終わりに到達すると、デルタ・エンジンはデルタ・テーブル全体で 2 巡目の確認を行い、1 巡目にアクセスされなかった項目を探します。アクセスされなかった項目は、シーケンス番号が現行シーケンス番号に更新されていないため、簡単に識別できます。したがって、デルタ・テーブル内で、現行シーケンス番号よりもシーケンス番号が小さい項目はすべて削除項目と見なされて、削除済みとして戻されます。

   注: 2 巡目は、入力データの反復が正しく完了した場合にのみ行われます。何らかの理由で反復時にエラーが発生した場合、項目が AssemblyLine によって削除済みとしてタグ付けされて戻されることはありません。また、デルタ・テーブルからも削除されません。このような状態になっても、元のデータ・ソースには影響ありません。次回 AssemblyLine が正しく実行されたときに、削除項目は正しく処理されます。

行のロック

「行のロック」パラメーターは、イテレーター・コネクターの「デルタ」タブ、およびデルタ関数コンポーネントの構成で利用できます。このパラメーターを使用すると、デルタ・ストア・データベースへの接続で使用されるトランザクション分離レベルを設定できます。高い分離レベルを設定すると、行とテーブルのロックを使用することにより、「ダーティー読み取り」、「非再現読み取り」、および「ファントム読み取り」として知られるトランザクション異常が削減されます。このパラメーターには次の値があります。

READ_UNCOMMITTED

`java.sql.Connection.TRANSACTION_READ_UNCOMMITTED` に相当し、ダーティー読み取り、非再現読み取り、およびファントム読み取りが発生する可能性があります。この分離レベルでは、あるトランザクションによって変更された行が、その行の変更内容がコミットされる前に、別のトランザクションによって読み取られる可能性があります (「ダーティー読み取り」)。変更内容がロールバックされた場合、2 つ目のトランザクションは無効な行を取得したことになります。

READ_COMMITTED

java.sql.Connection.TRANSACTION_READ_COMMITTED に相当し、ダーティー読み取りは回避されますが、非再現読み取り、およびファントム読み取りが発生する可能性があります。この分離レベルでは、コミットされていない変更内容を含む行の読み取りが禁止されます。

REPEATABLE_READ

java.sql.Connection.TRANSACTION_REPEATABLE_READ に相当し、ダーティー読み取りと非再現読み取りは回避されますが、ファントム読み取りが発生する可能性があります。この分離レベルでは、コミットされていない変更内容を含む行の読み取りが禁止されるとともに、あるトランザクションが行を読み取り、別のトランザクションがその行を変更して、最初のトランザクションがその行を再読み取りすると、2 度目に取得する値が一度目と異なるという状況 (「非再現読み取り」) も禁止されます。

SERIALIZABLE

java.sql.Connection.TRANSACTION_SERIALIZABLE に相当し、ダーティー読み取り、非再現読み取り、およびファントム読み取りが回避されます。この分離レベルでは、TRANSACTION_REPEATABLE_READ での禁止操作が含まれるほか、あるトランザクションが WHERE 条件を満たす行をすべて読み取り、別のトランザクションがその WHERE 条件を満たす行を挿入して、最初のトランザクションが同じ条件で再読み取りすると、2 度目の読み取りでは追加された「ファントム」行を取得するという状況が禁止されます。一般的に、この分離レベルは最も遅く、最も安全なオプションであり、「**行のロック**」パラメーターのデフォルト値です。

トランザクション分離レベルの詳細については、java.sql.Connection インターフェースのオンライン文書 (http://docs.oracle.com/javase/1.6.0/docs/api/java/sql/Connection.html) を参照してください。

各データベース・サーバーでは、デフォルトの分離レベルが設定されています。Apache Derby、Oracle および Microsoft SQL Server のデフォルト値は TRANSACTION_READ_COMMITTED です。ただし、デルタ・コンポーネント (つまり、イテレーター・コネクターのデルタ機能、またはデルタ関数コンポーネント) の使用時には「**行のロック**」パラメーターのデフォルト値 SERIALIZABLE が、サーバーのデフォルト値より優先されます。

一部のデータベース・サーバーでは、すべてのトランザクション分離レベルがサポートされているとは限りません。このため、使用するデータベースの文書を確認して、サポートされているトランザクション分離レベルについて正しい情報を得てください。

注: トランザクション分離レベルは、データベースに確立されるすべての接続に対してデータベース・サーバー自身によって維持されます。したがって、「**トランザクション分離レベル**」を REPEATABLE_READ または SERIALIZABLE に設定し、「**コミット**」パラメーターを On Connector Close に設定したデルタ・コンポーネントがトランザクションを開始すると、同じデータを変更しようとするその他の照会はすべてブロックされます。つまり、別のコンポーネントが同じデータを変更する必要がある場合、そのコンポーネントは、最初のコンポーネントが終了時にトラ

ンザクションをコミットするまで待機する必要があります。この待機により、発行された SQL 照会がタイムアウトになり、データが未変更のままになることがあります。

また、コンポーネントの「**コミット**」パラメーターが「**自動コミットなし**」に設定されている場合は、他のコンポーネントが変更の実施を行うためにいつまでも待機し続けることがないように、トランザクションを手動でコミットする必要があります。

特定の属性内の変更のみの検出または無視

パラメーター「**属性リスト**」および「**変更検出モード**」を使用すると、デルタ・エンジンが受け取ったすべての属性内ではなく、特定の属性内の変更のみを検出する機能を構成できます。

「**属性リスト**」パラメーターは、**変更検出モード**の影響を受ける属性のコンマ区切りリストです。この「**変更検出モード**」パラメーターで、これらの属性内の変更の処理方法を指定します。このパラメーターには次の 3 つの値があります。

IGNORE_ATTRIBUTES
(「以下の属性の変更を無視」) - 変更の計算の処理中に、「**属性リスト**」パラメーターに指定したすべての属性内の変更が無視されます。

DETECT_ATTRIBUTES
(「以下の属性の変更を検出」) - このオプションでは上記と逆の処理が行われます。「**属性リスト**」パラメーターにリストされている属性内の変更のみが検出されます。

DETECT_ALL
(「すべての属性で変更を検出」) - すべての属性内の変更を検出するようにデルタ・エンジンに指示します。このオプションを選択すると、影響を受ける属性のリストは不要のため、「**属性リスト**」パラメーターが使用不可に設定されます。

使用例

デルタ・エンジンを使用する際、受け取った項目内に、重要でないと思われるため無視したい属性が含まれている場合があります。このような場合、これらの属性がデルタ計算の結果に影響を与えないようにする必要があります。これは、項目の差分がこのような属性の違いのみである場合に、デルタ・ストア・テーブルの不要な更新が行われないようにするためです。

この問題を解決するには、「**属性リスト**」および「**変更検出モード**」パラメーターを使用します。

ここでは、2 つの AssemblyLine が LDAP サーバーのレプリカ 2 つから変更ログ項目を受け取り、これらの変更が 1 つのデルタ・ストアに適用されるシナリオ例を使用します。シナリオの説明では、以下の変更ログ項目例を使用します。

Entry1:

```
Entry attributes:
 targetdn (replace): 'cn=Niki,o=IBM,c=us'
 changetime (replace): '20071015094646'
```

```
 $dn (replace): 'changenumber=78955,cn=changelog'
 ibm-changeInitiatorsName (replace): 'CN=ROOT'
 changenumber (replace): '78955'
 objectclass (replace): 'top' 'changelogentry' 'ibm-changelog'
 changetype (replace): 'modify'
 cn (replace): 'Niki' 'Niky'
 changes (replace): 'replace: cn
       cn: Niki
       cn: Niky
       -
       '
```

Entry2:

```
Entry attributes:
 targetdn (replace): 'cn=Niki,o=IBM,c=us'
 changetime (replace): '20071015094817'
 $dn (replace): 'changenumber=10076,cn=changelog'
 ibm-changeInitiatorsName (replace): 'CN=ROOT'
 changenumber (replace): '10076'
 objectclass (replace): 'top' 'changelogentry' 'ibm-changelog'
 changetype (replace): 'modify'
 cn (replace): 'Niki' 'Nikolai'
 changes (replace): 'replace: cn
       cn: Niki
       cn: Nikolai
       -
      '
```

Entry3:

```
Entry attributes:
 targetdn (replace): 'cn=Niki,o=IBM,c=us'
 changetime (replace): '20071037454817'
 $dn (replace): 'changenumber=112,cn=changelog'
 ibm-changeInitiatorsName (replace): 'CN=ADMIN'
 changenumber (replace): '112'
 objectclass (replace): 'top' 'changelogentry' 'ibm-changelog'
 changetype (replace): 'modify'
 cn (replace): 'Niki' 'Nikolai'
 changes (replace): 'replace: cn
       cn: Niki
       cn: Nikolai
       -
          '
```

変更された属性は**太字**で、無視できる属性は*イタリック* でマークされています。無視される属性 (changenumber、changetime など) は、受け取った項目と格納されている項目を比較する際に考慮されません。したがって、これらの属性は「**属性リスト**」パラメーターにリストされている必要があります。これらの属性を無視することを指定するには、「**変更検出モード**」パラメーターを「**次の属性についての変更を無視**」に設定する必要があります。

以下に、AssemblyLine がこれらの項目を受け取る際のワークフローを示します。

1. AL1 が Entry1 を受け取ると、Entry1 は変更 として戻されて、デルタ・ストア・テーブルに保存されます。
2. AL2 が Entry2 を受け取ると、changetime、$dn、ibm-changeInitiatorsName、および changenumber 属性が変更されていますが、無視されます。ただし、cn および changes 属性も変更されているため、生成されるデルタ項目は変更 としてタグ付けされて、デルタ・ストア・テーブルに保存されます。

3. AL2 が Entry3 を受け取ると、changetime、$dn、ibm-changeInitiatorsName、および changenumber 属性が変更されていますが、無視されます。残りの属性は等しいため、生成されるデルタ項目は*未変更* としてタグ付けされて、AssemblyLine に戻されるか (「**未変更項目を戻す**」パラメーターにチェック・マークが付いている場合のみ)、スキップされます。戻されるデルタ項目は、受け取った Entry3 と全く同じになります。この場合、デルタ・ストアは更新されません。「属性リスト」および「変更検出モード」パラメーターを使用していなければ、最後の Entry3 は*変更* としてタグ付けされて、デルタ・ストアに保存されます。

変更検出コネクター

変更検出コネクターは、接続されたシステムの情報を利用して変更を検出します。コネクターのタイプによって、イテレーター・モードまたはサーバー・モードのいずれかで使用します。例えば、イテレーター・モードは、多くの変更検出コネクター (LDAP、RDBMS、Active Directory、Notes/Domino 用) で使用されます。変更検出コネクターは、それぞれが同様に動作し、共通の設定に対して同じパラメーター・ラベルが使用されるように設計されています。

IBM Security Directory Integrator には、次の変更検出コネクターがあります。

- IBM Security Directory Server 変更ログ
- AD 変更検出 (Active Directory)
- Domino 変更検出
- Sun Directory 変更検出 (openLDAP、SunOne、iPlanet など)
- RDBMS 変更検出 (DB2、Oracle、SQL サーバーなど)
- z/OS LDAP 変更ログ

  注: IBM Security Directory Integrator バージョン 7.2 以降では z/OS オペレーティング・システムはサポートされません。

デルタ・エンジン機能は、特定の変更について、属性の個々の値に至るまで詳細に報告します。LDIF パーサーまたは DSMLv2 パーサーのいずれかで解析する場合は、属性値レベルでのデルタ・タグ付けも使用可能です。LDIF パーサーは、IBM Security Directory Server 変更ログ・コネクター、Sun Directory 変更検出コネクター、および z/OS LDAP 変更ログ・コネクターで内部的に使用されるため、これらのコネクターでは属性値レベルでのデルタ・タグ付けもサポートされます。その他の変更検出コネクターでは、単に項目全体が追加、変更、または削除された場合のみに報告されます。特定のコンポーネントにおけるデルタ・タグ付けサポートの詳細については、「*リファレンス*」に記載されているそのコンポーネントに関する説明を参照してください。

場合によっては、長時間実行される AssemblyLine で同一項目を複数回処理する必要があります。これらの項目は重複デルタ鍵を持つことになるため、AssemblyLine が例外をスローする原因になります。重複デルタ鍵を許可する場合は、イテレーターの「デルタ」タブで「**重複デルタ鍵を許可**」チェック・ボックスにチェック・マークを付けます。これにより、デルタが使用可能なイテレーター・コネクターまたは変更検出コネクターと、デルタ・モード・コネクターの両方を持つ AssemblyLine で重複項目を処理できます。

注: 例えば、多数の変更ログとデルタ・モード・コネクターを持つ AssemblyLine を使用することができます。この場合、デルタ・モード・コネクターが変更ログ・コネクターと同じ基盤システムをポイントすると、デルタ操作によって変更ログを再度起動することができます。新しく受信した変更とデルタ・エンジンが起動した変更を区別する方法はないので、無限ループに落ちないようにするためには、適用するシナリオを慎重に検討する必要があります。

デルタ・エンジンで計算されるデルタ情報は、作業項目オブジェクトに保管され、使用される変更検出コンポーネントまたは機能に応じて、*項目レベル*、*属性レベル* または*属性値レベル* の命令コードとして保管できます。

デルタ項目の取り込み

AssemblyLine のデルタ項目は、デルタ・モードのコネクターによって、および、変更の計算ロジックの内部で更新モードのコネクターによって動作の対象となり (「取り込まれ」) ます。

デルタ・モードは、次のコネクターで使用可能です。

- JDBC コネクター
- DSMLv2 SOAP コネクター
- JNDI コネクター
- LDAP コネクター

デルタ・モード・コネクター

デルタ・モードは、接続されたシステムへの増分変更を提供することで、データへの変更適用を簡略化できるように設計されています。これは、デルタ命令コードに基づいています。増分変更とは、変更された特定の値のみを書き込むことです。

第 1 に、デルタ・モードはすべてのデルタ・タイプ (追加、変更、削除) を処理します。デルタ・モードが動作するには、デルタ項目を受け取る必要があります。このため、デルタ・モードのコネクターを使用する場合は、デルタ項目を生成するコンポーネントと結合させる必要があります。デルタ項目を生成するコンポーネントは、デルタを使用可能にしたイテレーター・コネクター、変更検出コネクター、または LDIF または DSMLv2 パーサーを使用するコネクターです。例えば、2 つのコネクター (変更を選出する「フィード」セクションの変更検出コネクターと、ターゲット・システムに変更を適用するデルタ・モードのコネクター) のみを使用して、2 つのシステムを同期化することが可能です。

また、デルタ・モードでは、ターゲット・システム自体がサポートする最下位のレベルでデルタ情報を適用します。 これは、最初にコネクター・インターフェースをチェックして、データ・ソースがサポートする増分変更のレベルを確認することで行われます。LDAP ディレクトリーで作業中の場合、デルタ・モードが属性値の追加と削除を実行します。従来の RDBMS (JDBC) の場合、列の値を削除してから追加することは意味がないため、その属性値の置換として処理されます。

また、増分変更は、この機能をサポートするデータ・ソースに対して、デルタ・モードにより自動的に処理されます。データ・ソースが最適化された呼び出しにより増分変更を処理し、この呼び出しがコネクター・インターフェースによってサポートされる場合は、デルタ・モードがこの呼び出しを使用します。一方、接続された

システムが「インテリジェント」デルタの更新メカニズムを提供しない場合、デルタ・モードは可能な限りシミュレートし、事前更新のルックアップ (更新モードに類似)、計算の変更、および削除した変更に続くアプリケーションを実行します。

注: 増分変更をサポートするコネクターは LDAP コネクターのみです。これは、LDAP ディレクトリーがこの機能を提供するためです。

IBM Security Directory Integrator のデルタ機能は、変更検出機構を提供するデータ・ソースを使用するだけでなく、例えば、フラット・ファイルも使用して、同期ソリューションを円滑に行うよう設計されています。次のダイアグラムは、イテレーター・モードおよびデルタ・モードのコネクターを使用するそのような同期ソリューションを示しています。イテレーター・コネクターがデータ・ソースから項目を読み取ります。読み取られた項目が、前回の反復時にデルタ・ストアに保存された項目と比較されます。比較の結果は、変更の内容 (追加/削除/変更) を定義するデルタ命令コードが割り当てられたデルタ項目となります。次に、デルタ・モードのコネクターがデルタ項目を使用し、検出された変更を宛先システムに適用します。

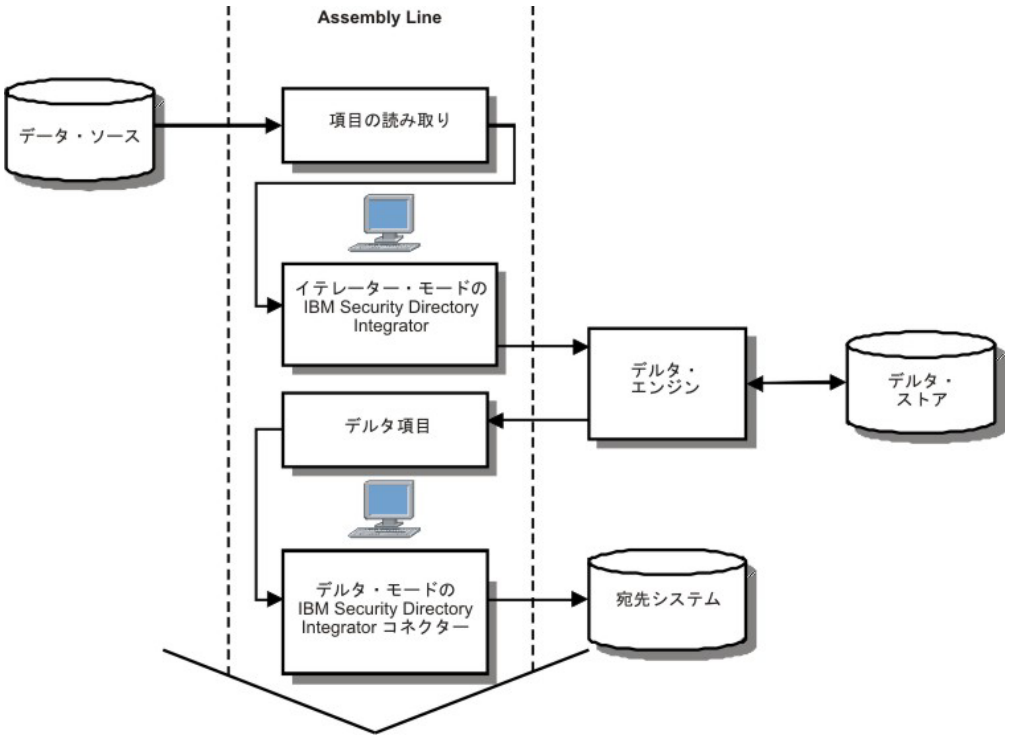


図 117. デルタ機能を使用する同期化 AssemblyLine

AssemblyLine の実行の結果、データ・ソースに含まれているデータが宛先システムのデータと同期化されます。

更新モードとデルタ項目

更新モードのコネクターでは、「変更の計算」と呼ばれる変更制御機能が追加で使用できます。この機能は、デルタ項目と組み合わせて動作させることができます。

「変更の計算」機能は、コネクターの「詳細」ペインでこの名前のチェック・ボックスを選択すると、更新モードのコネクターで使用可能になります。この機能をオンにすると、コネクターは自身の項目と、接続システム内の実際の項目を比較しま

す。2 つの項目が等しい場合、コネクターは計算を行いません。つまり、「変更の計算」オプションを使用すると、更新モードのコネクターは不要な更新をスキップすることができます。このオプションは、更新操作が比較的重いデータ・ソースで役立ちます。

ただし、「**変更の計算**」パラメーターを使用可能にした更新モードのコネクターがデルタ項目を受け取ると、「変更の計算」ロジックは起動されません。この場合は、デルタ項目に割り当てられたデルタ命令コードを使用して、必要な更新を接続システムに適用します。

例

簡単なデルタの例

ここでは、これまでに説明したデルタ機能の一部の使用例を説明します。

デルタ・エンジン機能を有効にしてファイル・システム・コネクターをセットアップします。テキスト・エディター内で簡単に変更可能な単純な XML 文書を繰り返してください。例を示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<DocRoot>
    <Entry>
        <Telephone>
            <ValueTag>111-1111</ValueTag>
            <ValueTag>222-2222</ValueTag>
            <ValueTag>333-3333</ValueTag>
        </Telephone>
        <Birthdate>1958-12-24</Birthdate>
        <Title>Full-Time SDI Specialist</Title>
        <uid>jdoe</uid>
        <FullName>John Doe</FullName>
    </Entry>
</DocRoot>
```

属性マップのすべての文字をマップする特殊文字であるアスタリスク (*) を必ず使用してください。これは、戻されるすべての属性が作業項目オブジェクトに必ずマップされるようにするために必要な唯一の属性です。

ここで次のコードを使用してスクリプト・コンポーネントを追加します。

```
// Get the names of all Attributes in work as a String array
var attName = work.getAttributeNames();

// Print the Entry-level delta op code
task.logmsg(" Entry (" +
work.getString( "FullName" ) + ") : " +
work.getOperation() );

// Loop through all the Attributes in work
for (i = 0; i < attName.length; i++) {

 // Grab an Attribute and print the Attribute-level op code
 att = work.getAttribute( attName[ i ] );
 task.logmsg("    Att ( " + attName[i] + ") : " + att.getOperation() );

 // Now loop through all the Attribute's values and print their op codes
 for (j = 0; j < att.size(); j++) {
  task.logmsg( "      Val (" +
```

```
                att.getValue( j ) + ") : " +
                att.getValueOperation( j ) );
 }
}
```

この AL を初めて実行する際にスクリプト・コンポーネント・コードによって、次のログ出力が作成されます。

```
12:46:31   Entry (John Doe) : add
12:46:31      Att ( Telephone) : replace
12:46:31        Val (111-1111) :
12:46:31        Val (222-2222) :
12:46:31        Val (333-3333) :
12:46:31      Att ( Birthdate) : replace
12:46:31        Val (1958-12-24) :
12:46:31      Att ( Title) : replace
12:46:31        Val (Full-Time SDI Specialist) :
12:46:31      Att ( uid) : replace
12:46:31        Val (jdoe) :
12:46:31      Att ( FullName) : replace
12:46:31        Val (John Doe) :
```

この項目が以前に空だったデルタ・ストアで見つからなかったため、項目レベルで new というタグが付けられます。また、その各属性には、すべての値が変更されていることを意味する replace コードが付きます (デルタによりこれが新規データであることが通知されるので意味があります)。

XML ファイルに次の変更を加えます。

1. 最後の Telephone (電話番号) の値を 333-4444 に変更します。
2. Birthdate を削除します。
3. 新規の Address 属性を追加します。

構成の結果は次のようになります。

```
<?xml version="1.0" encoding="UTF-8"?>
<DocRoot>
    <Entry>
        <Telephone>
            <ValueTag>111-1111</ValueTag>
            <ValueTag>222-2222</ValueTag>
            <ValueTag>333-4444</ValueTag>
        </Telephone>
        <Title>Full-Time SDI Specialist</Title>
        <uid>jdoe</uid>
        <FullName>John Doe</FullName>
        <Address>123 Willowby Lane</Address>
    </Entry>
</DocRoot>
```

AL を再実行します。今回のログの出力は、次のようになります。

```
13:53:22   Entry (John Doe) : modify
13:53:22      Att ( Telephone) : modify
13:53:22        Val (111-1111) : unchanged
13:53:22        Val (222-2222) : unchanged
13:53:22        Val (333-4444) : add
13:53:22        Val (333-3333) : delete
13:53:22      Att ( Birthdate) : delete
13:53:22        Val (1958-12-24) : delete
13:53:22      Att ( uid) : unchanged
13:53:22        Val (jdoe) : unchanged
13:53:22      Att ( Title) : unchanged
13:53:22        Val (Full-Time SDI Specialist) : unchanged
```

```
13:53:22      Att ( Address) : add
13:53:22        Val (123 Willowby Lane) : add
13:53:22      Att ( FullName) : unchanged
13:53:22        Val (John Doe) : unchanged
```

ここでは、項目に *modify* というタグが付けられ、変更が属性に反映されています。ご覧のように、Birthdate 属性には *delete*、Address には *add* のマークが付いています。そのため、入力マップでは、すべての文字をマップするための特殊文字を使用しました。XML 文書の最初のバージョンに存在する属性のみをマップした場合、住所が入力に表示されても取得されません。

modify としてマークされている Telephone 属性の下にある値項目のうち、最後の 2 つに特に注目してください。この属性値の 1 つを変更すると、2 つのデルタ項目 (値 *delete*、次いで *add*) が生じます。

IBM Security Directory Integrator の以前のバージョンでデータ同期 AssemblyLine を構築するには、フロー制御を処理するためにスクリプトを記述する必要がありました。変更コンポーネントまたは機能から *adds*、*modifies*、および *deletes* を受け取ることはできますが、コネクターは、2 つの要求される出力モード、更新または削除のいずれかにのみ設定可能でした。

そのため、同じターゲット・システムを指す 2 つのコネクターで、それぞれの実行前フックにスクリプトを配置し、命令コードがこのコンポーネントのモードと一致しない場合に項目を無視するようにするか、*受動*状態の 1 つのコネクター (更新モードまたは削除モードのいずれか) で、命令コードをチェックしたスクリプト・コードから、その実行を制御するようにすることが必要とされました。つまり、項目が変更されている場合に何が変更されたかがわかっていても、その変更をデータ・ソースに書き戻す前なら、更新モードのコネクターは元のデータを読み込むことがまだ可能でした。これは、数千個の値を含む、多値のグループに関連した属性で、1 つの値だけを変更する場合に、ネットワークまたはデータ・ソースに不要なトラフィックを発生させる可能性があります。

その他の例

IBM Security Directory Integrator インストール・システムの `root_directory/examples/` ディレクトリーを参照してください。

- サブディレクトリー `deltas` には、デルタ機能をデモンストレーションする簡単な IBM Security Directory Integrator 構成が格納されています。このデモは、MS Access のデータベースと JDBC:ODBC ブリッジを使用するため、MS Windows システム (Windows 2000) のみで動作します。その他のプラットフォームを使用する場合は、独自にデータベースを作成し、データベース用にコネクターの JDBC 設定を構成する必要があります。
- サブディレクトリー `delta_tagging` には、値レベルでタグ付けされた項目を通常の項目に変換する方法、および通常の項目をデルタ項目に変換する方法のデモンストレーション例が格納されています。

第 8 章 IBM Security Directory Integrator ダッシュボード

IBM Security Directory Integrator ダッシュボードを使用して、データ統合ソリューションをインストール、構成、デプロイ、およびモニターします。

ダッシュボードを使用して、単純データ統合用の EasyETL ソリューション (ETL は Extract、Transform、Load [抽出、変換、ロード] の略です) を作成および構成することもできます。EasyETL ソリューションには、単一のソース・コネクターとターゲット・コネクターを含む単一の AssemblyLine が含まれます。

IBM Security Directory Integrator ダッシュボードの概要

ダッシュボードは、Open Service Gateway Initiative (OSGI) フレームワークを使用して開発される Web アプリケーションです。ダッシュボードは、RESTful (Representational State Transfer) インターフェースを介してサーバー上のデータ統合ソリューションを構成および管理するために使用されます。ダッシュボードから、以下を実行できます。

- 既存のデータ統合ソリューションをアップロードし、それを通常のソリューションとしてインストールするか、テンプレートとして保存します。
- ローカル環境に固有の変更を反映するように、データ統合ソリューションを構成します。
- 指定した時刻に AssemblyLine を実行するように、スケジュールを追加します。
- 単一のソース・コネクターとターゲット・コネクターを含む単純 AssemblyLine を作成します。
- 選択したコネクターの内容をブラウズします。
- データ統合ソリューションのプロセスをデプロイ、モニター、および監査します。
- サーバー詳細、サーバー・ログ、およびシステム・ストア・データを表示します。
- 必要な情報のみを抽出するように、フィルター基準を設定して、ログ・ファイル・データをフィルタリングします。
- AssemblyLine 実行履歴をグラフの形式で表示します。
- AssemblyLine 実行状況に関するデータを含むレポートを作成し、それを E メールを介してユーザーに自動的に送信します。

IBM Security Directory Integrator ダッシュボードの利点

以下の利点があります。

- データ統合ソリューションをより単純に短期間でデプロイできるようにします。
- 構成エディターを使用せずに、単純な統合を作成および構成します。
- AssemblyLine 履歴の E メール・レポートをスケジュールに従って生成し、それをダッシュボード RunReport 機能を使用して送信します。このレポートを使用して、デプロイメントの高可用性/フェイルオーバー要件に対処できます。
- Eclipse 開発環境と容易に統合できます。

- AssemblyLine 実行の効果的かつ効率的な管理を実現します。

ダッシュボード・アプリケーションへのアクセス

ダッシュボード Web アプリケーションには、ブラウザーまたは構成エディターからアクセスできます。

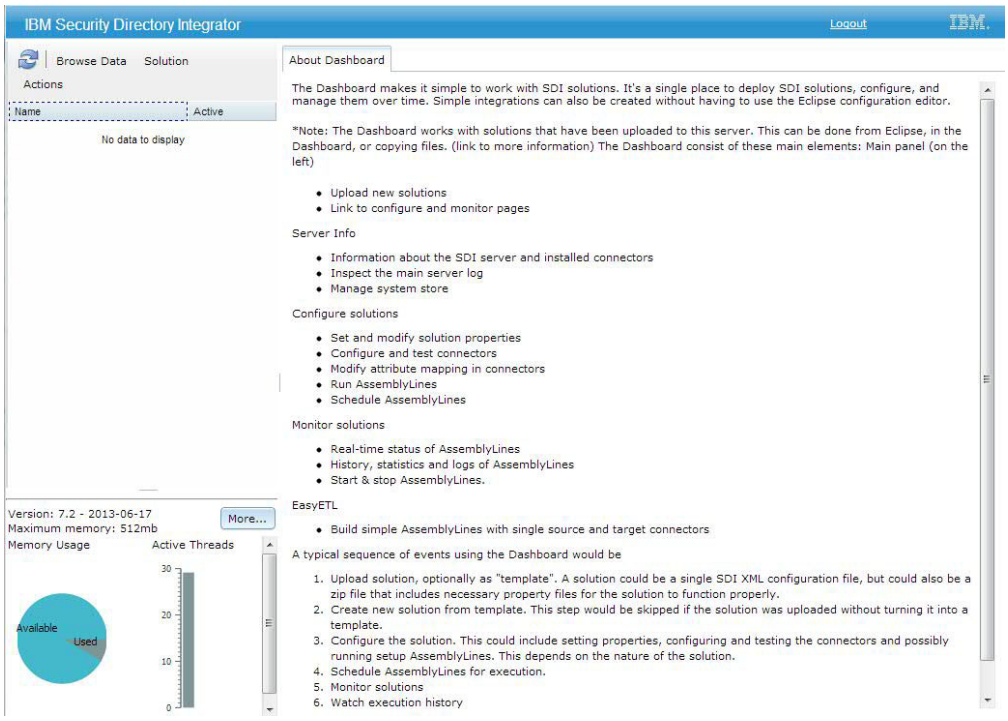
始める前に

ダッシュボードにアクセスする前に、IBM Security Directory Integrator サーバーを始動します。

ブラウザーから開く

手順

ブラウザーから、`https://<host name>:<rest-api-port>/dashboard` にアクセスします。ダッシュボードのホーム・ページが表示されます。



注: デフォルトでは、ローカル・ホスト上のダッシュボードにアクセスできます。ユーザーを認証するには、IBM Security Directory Integrator プロパティー・ファイルを設定して、リモート・アクセスとローカル・アクセスの両方の LDAP ユーザー名およびパスワードを追加します。

Windows の「スタート」メニューから開く

手順

Windows の「**スタート**」メニューから、「**スタート**」 > 「**すべてのプログラム**」 > 「**IBM Security Directory Integrator v7.2** > 「**ダッシュボードを開く**」を選択します。 IBM Security Directory Integrator ダッシュボードのホーム・ページが表示されます。

表 14. IBM Security Directory Integrator ダッシュボードのメニュー・オプション

メニュー・オプション		説明
データのブラウズ		このオプションを使用して、選択したコネクターを構成し、その内容をブラウズします。
ソリューション	**モニター**	このオプションを使用して、AssemblyLine 実行の進行状況を追跡し、ソリューションの実行履歴を表示します。
	開始	このオプションを使用して、選択したソリューションを開始します。
	停止	このオプションを使用して、実行中のソリューションを強制終了します。
	構成	このオプションを使用して、選択したソリューションを変更するために開きます。
	削除	このオプションを使用して、選択したソリューションをサーバーから削除します。
	ソリューションのロックを解除	このオプションを使用して、ソリューションのロックを解除します。
アクション	**データのブラウズ**	このオプションを使用して、選択したコネクターを構成し、その内容をブラウズします。
	システム・ログの表示	このオプションを使用して、システム・ログ (`ibmdi.log`) の内容を表示します。
	ソリューションの作成	このオプションを使用して、テンプレートを選択し、データ統合ソリューションを作成します。
	ソリューションのアップロード	このオプションを使用して、既存のソリューションを IBM Security Directory Integrator サーバーにアップロードします。
	サーバー詳細の表示	このオプションを使用して、バージョン、インストール済みコンポーネント、その他のサーバー情報など、IBM Security Directory Integrator についての情報を表示します。

リモート・アクセス用の Internet Explorer の設定

Internet Explorer ブラウザーでリモート・システムからダッシュボードにアクセスするために必要な設定を追加します。ここでは、Internet Explorer セキュリティ強化の構成 (IE ESC) を有効にします。

デフォルトでは、IE ESC は、Web ページ上で実行されるすべてのスクリプトをブロックします。ダッシュボードでは、Web ページ上に表示を行う前にいくつかのスクリプトがロードされます。したがって、IESC が有効な Internet Explorer ブラウザーでは、ダッシュボードを開くとブランク・ページが表示されます。ページにアクセスするには、ダッシュボードをホストするサイトを Internet Explorer の安全なリストに追加する必要があります。

1. **Internet Explorer** のメニューから、「**ツール**」 > 「**インターネット オプション**」をクリックします。
2. 「**セキュリティ**」タブをクリックします。
3. 「**信頼済みサイト**」をクリックします。
4. 「**サイト**」をクリックします。
5. 「**この Web サイトをゾーンに追加する**」フィールドに、ダッシュボードの URL を入力します。例: `https://mydashboard.com/dashboard/*`。
6. 「**追加**」をクリックします。
7. 「**閉じる**」の次に「**OK**」をクリックしてページを閉じ、設定を保存します。
8. Internet Explorer ブラウザーを再起動します。
9. Internet Explorer ブラウザーでダッシュボードにアクセスします。

データ統合ソリューションのアップロード

ダッシュボードの「ソリューションのアップロード」ウィンドウを使用して、既存のデータ統合ソリューションをアップロードし、そのソリューションを IBM Security Directory Integrator サーバー上にインストールします。アップロードしたソリューションをテンプレートとして保存することもできます。

このタスクについて

ダッシュボードを使用して、既存の IBM Security Directory Integrator データ統合ソリューション (XML 構成ファイル) をアップロードし、そのソリューションをサーバー上にインストールすることができます。デプロイする前に、必要なプロパティーを指定して、ソリューションを変更できます。既存のソリューションをアップロードし、そのソリューションをテンプレートとして保存することもできます。保存したテンプレートを使用して、同じ統合の多くのインスタンスを作成できます。

手順

1. 「ダッシュボード」ウィンドウのナビゲーション・ペインで、「**アクション**」 > 「**ソリューションのアップロード**」をクリックします。
2. 「ソリューションのアップロード」ウィンドウで以下の設定を定義します。

オプション	説明
ブラウズ	アップロードする既存のソリューションをブラウズします。
テンプレートとしてアップロード	アップロードしたソリューションをテンプレートとして保存します。 **注:** 保存したテンプレートは、IBM Security Directory Integrator サーバー (config ディレクトリー) 上にはインストールされません。

オプション	説明
既存のソリューションの上書き	既存のインストール済みのソリューションまたはテンプレートを上書きします。
ソリューション名	アップロードするソリューションの名前を指定します。 **注:** 「ソリューションのアップロード」ウィンドウで、現在インストール済みのソリューションおよびテンプレート・ソリューションの名前を確認することもできます。

3. 「**OK**」をクリックします。

データ統合ソリューションの作成

ダッシュボードの「ソリューションの作成」ウィンドウで、さまざまなオプションを使用して、データ統合ソリューションを作成します。

手順

1. 「ダッシュボード」ウィンドウのナビゲーション・ペインで、「**アクション**」 > 「**ソリューションの作成**」をクリックします。
2. 「ソリューションの作成」ウィンドウで以下のオプションのいずれかを選択します。

オプション	説明
インストール済みのソリューション	ソリューションは、IBM Security Directory Integrator サーバー上の既にインストール済みのソリューションに基づいて作成されます。
テンプレート	ソリューションは、テンプレートとしてアップロードされる既存のソリューションに基づいて作成されます。
デフォルト	単一の AssemblyLine と 2 つのコネクターを含む EasyETL ソリューションが作成されます。

3. ソリューションの名前を「**ソリューション名**」フィールドに入力します。
4. 「**OK**」をクリックします。

ソリューション構成

ダッシュボードは、データ統合ソリューションのコンポーネントを構成するためのさまざまなエディターから成り立っています。

ダッシュボードでは、デプロイメントの前に、ユーザー固有の必要性に応じた適切な設定を指定して、選択したソリューションを変更できます。迅速に構成できるように、ソリューションの構成中に表示される情報の量を制限できます。ソリューションに固有のプロパティーを指定するために、新規構成ページを追加することもできます。

インストール済みのデータ統合ソリューションのみが「ダッシュボード」ウィンドウのナビゲーション・パネルに表示されます。各ソリューションには、AssemblyLine や関連付けられたコネクターなどのコンポーネントが含まれます。これらのコンポーネントは、ソリューション・エディターにツリー構造として表示されます。ソリューション・エディターから、以下のエディターを使用して、ソリューションをインストール、作成、構成、デプロイ、およびモニターできます。

- AssemblyLine エディター – このエディターを使用して、AssemblyLine 内のスケジュールを作成および構成し、AssemblyLine を実行してその出力をログ・ビューアーに表示します。
- コネクター・エディター – このエディターを使用して、AssemblyLine のコネクターを構成します。
- EasyETL エディター – このエディターを使用して、単一の AssemblyLine と 2 つのコネクターを含む EasyETL ソリューションを構成します。
- モニター・エディター – このエディターを使用して、ソリューションの現在および過去の両方のパフォーマンスの情報をモニターします。

ソリューション・エディターには、次のメニュー・オプションがあります。

オプション	説明
ソリューション・インターフェースの編集	このオプションを使用して、編集用に表示されるソリューションのコンポーネントを選択します。
新規 AssemblyLine	このオプションを使用して、EasyETL ソリューションを作成し、それを選択したソリューションに追加します。
Assemblyline 名の変更	このオプションを使用して、選択した Assemblyline 名を変更します。
Assemblyline の削除	このオプションを使用して、AssemblyLine を削除します。
RunReport の作成	このオプションを使用して、AssemblyLine 実行の状況についての E メールを送信する RunReport を作成します。

ソリューションの説明の追加

ダッシュボードのソリューション・エディターを使用して、データ統合ソリューションについての重要な情報を追加します。

手順

1. 「ダッシュボード」ウィンドウのナビゲーション・ペインで、ソリューションを選択します。
2. 「**ソリューション**」 > 「**構成**」をクリックします。
3. ソリューション・エディターで、「**ソリューションの説明**」を選択し、「**説明の編集**」をクリックします。
4. 説明をテキスト域に入力します。
5. 「**クローズ**」をクリックします。
6. 「**保管**」をクリックします。

AssemblyLine スケジュールの構成

AssemblyLine エディターを使用して、ダッシュボード・スケジューラーを作成または構成することにより、指定した時刻にデータ統合ソリューションの AssemblyLine を実行します。

スケジュールの作成

手順

1. ツリー構造から AssemblyLine を選択します。
2. AssemblyLine エディターで、「**スケジュール**」タブをクリックします。
3. 「**スケジュールの作成**」をクリックします。
4. 以下の設定を定義します。

表 15. スケジュール・オプション

設定	オプション	説明
スケジュール	**ソリューションの開始時に自動的に開始する**	このオプションを使用して、選択した実行オプションに基づいて AssemblyLine を実行します。
	既に実行中の場合は開始しない	
	AssemblyLine が失敗した場合にスケジュールを強制終了する	
月	**毎月**	このオプションを使用して、スケジュールを実行する月を選択します。
	月の選択	
日	**毎日**	このオプションを使用して、スケジュールを実行する日を選択します。
	平日	
	曜日の選択	
時/分/秒	**時**	このオプションを使用して、スケジュールを実行する時刻を選択します。
	分	
	秒	
ロギングの共用		このオプションを使用して、スケジューラーと AssemblyLine の間でロギングを共有するかどうかを指定します。

5. 「**保管**」をクリックします。

スケジュールの削除

手順

1. ツリー構造から AssemblyLine を選択します。
2. AssemblyLine エディターで、「**スケジュール**」タブをクリックします。
3. 「**スケジュールの削除**」をクリックします。
4. 「**保管**」をクリックします。

ダッシュボード・スケジューラーの実行と停止

手順

1. ツリー構造から AssemblyLine を選択します。
2. AssemblyLine エディターで、「**実行**」タブをクリックします。
3. AssemblyLine を実行するには、「**実行**」をクリックします。出力はログ・ビューアーに表示されます。
4. AssemblyLine を停止するには、「**停止**」をクリックします。

コネクターの構成

コネクター・エディターを使用して、属性のマッピング情報や接続詳細などのコネクター詳細を、構成要件に合わせて構成します。

接続詳細の変更

手順

1. ソリューションのツリー構造からコネクターを選択します。
2. 「**コネクター**」タブをクリックします。
3. 重要なフィールドのみを表示するには、「**詳細の非表示**」をクリックします。
4. コネクター・タイプを変更するには、「**コンポーネントの選択**」をクリックし、「**コネクターの選択**」リストからコネクターを選択します。
5. 要件に応じて、接続設定を変更します。
6. データ・ソースへの接続をテストするには、「**テスト接続**」をクリックします。
7. 「**保管**」をクリックします。

属性マッピングの変更

手順

1. ソリューションのツリー構造からコネクターを選択します。
2. 「**属性マップ**」タブをクリックします。
3. 属性を追加するには、以下を実行します。
   a. 「**追加**」をクリックします。
   b. リストから作業属性を選択するか、テキスト・フィールドに新しい名前を入力します。
   c. 「**OK**」をクリックします。
4. コネクター・データを読み込み、それを属性マップ内に表示するには、「**次を読み込む**」をクリックします。
5. データ・ソースへの接続をクローズするには、「**接続のクローズ**」をクリックします。
6. 選択した属性を変更するには、「**アクション**」をクリックします。
   a. 選択した属性の割り当てを編集するには、「**属性の編集**」をクリックし、以下の割り当てタイプを変更します。
      - **単純な割り当て** – ソース属性リストから属性を割り当てることができます。

- **スクリプト化された割り当て** – テキスト域で属性にスクリプトを書き込むことができます。

b. 「**クローズ**」をクリックします。

c. 選択した属性をマップするには、「**属性のマップ**」をクリックします。

d. 選択した属性のマッピングを解除するには、「**属性のマップ解除**」をクリックします。

7. 「**保管**」をクリックします。

ダッシュボード EasyETL

ダッシュボード EasyETL 機能は、単純データ統合ソリューションを作成および構成するために使用できます。

ダッシュボード EasyETL は、単一のソースとターゲット・コネクターを含む単純な AssemblyLine を作成および構成するためのユーザー・インターフェースを提供します。 EasyETL ソリューションは、以下のように操作できます。

- ダッシュボード・スケジューラーでスケジュールする
- テンプレートとして保存する
- RunReport に組み込む
- ダッシュボード・ソリューション・モニターにデプロイしてモニターする

EasyETL ソリューションの構成

EasyETL エディターを使用して、スケジュールを構成し、それを EasyETL ソリューションに追加します。

手順

1. 「ダッシュボード」ウィンドウのナビゲーション・ペインで、EasyETL ソリューションを選択します。
2. 「**ソリューション**」 > 「**構成**」をクリックします。

   または

   a. 選択したソリューションを右クリックします。

   b. メニューから、「**構成**」をクリックします。
3. ソリューション・エディターで、EasyETL ソリューションを選択します。
4. 説明をソリューションに追加します。詳しくは、 242 ページの『ソリューションの説明の追加』のトピックを参照してください。
5. EasyETL エディターで、ソリューションを選択し、「**構成**」をクリックします。
6. 「**コネクター**」リストから、ソース・コネクターとターゲット・コネクターのためのコンポーネントを選択します。
7. 選択したコネクターの構成詳細をフォームに指定します。
8. 「**パーサー**」リストからパーサーを選択します。

   注: 「**パーサー**」リストは、選択したコネクターがパーサーを必要とする場合、またはパーサーを使用できる場合にのみアクティブになります。

接続を確立し、データをブラウズするには、以下を実行します。

a. 接続を確立し、コネクターの最初の 25 個のレコードを読むには、「**接続**」をクリックします。
b. 次の 25 個のレコードを表示するには、「**次へ**」をクリックします。
c. データ・レコード・ビューを切り替えて一度に 1 つのレコードを表示するには、「**ビューの切り替え**」をクリックします。
d. 切り替えたビューで次のレコードを表示するには、「**次へ**」をクリックします。
e. 選択した属性のみをテーブルに表示するには、以下を実行します。
   1) メニュー・バーの「**オプションの構成**」アイコンをクリックします。
   2) 「オプションの構成」ウィンドウで、属性を選択します。
   3) 属性の順序を変更するには、上向きまたは下向き矢印をクリックします。
   4) 「**OK**」をクリックします。

9. 「**戻る**」をクリックして、EasyETL エディターに戻ります。
10. 構成したコネクターの属性を表示するには、「**ディスカバー**」をクリックします。
11. 選択したソース属性をターゲット属性にマップするには、「**マップ**」をクリックします。 ソース属性の選択に基づいて、以下のマッピング・タイプが作成されます。
    - ソース・ビューとターゲット・ビューでそれぞれ 1 つの属性が選択されている場合は、2 つの属性の間に 1 対 1 マップが作成されます。
    - ソース・ビューで複数の属性が選択され、ターゲット・ビューで属性が選択されていない場合は、属性ごとに 1 対 1 マップが作成されます。ターゲット・ビューに属性がない場合は、それらの属性が作成されます。
    - ソース・ビューで複数の属性が選択され、ターゲット・ビューで 1 つの属性が選択されている場合は、以下のタスクのいずれかを実行します。
      – 「**コピー**」をクリックして、ソース属性をそれに相当するターゲット属性にコピーします。
      – 「**マージ**」をクリックして、ソースからのすべての値をターゲット内の 1 つの属性にマージします。
      – 「**連結**」をクリックして、ソース属性からの値を連結して、ターゲット属性に対する 1 つの値を生成します。
12. 「**保管**」をクリックします。

サーバー構成

「サーバー情報」ウィンドウを使用して、サーバー構成を表示および変更します。

「サーバー情報」ウィンドウでは、以下のサーバー・プロパティーを表示し、サーバーの動作を構成することができます。

- 作成および保守されるログ・ファイルの最大数の指定
- トゥームストーン・マネージャーの開始および停止

- ダッシュボード・アプリケーションにアクセスするユーザーを認証するための LDAP サーバー設定の定義
- IBM Security Directory Integrator バージョンとビルド日、ホスト名、IP アドレス、サーバーのブート時刻、OS 名、サーバー ID などのサーバー情報の表示
- IBM Security Directory Integrator サーバー上にインストールされたコネクターとパーサーの表示
- IBM Security Directory Integrator サーバーが使用しているさまざまなシステム・ストアの内容の表示

「ダッシュボード」ウィンドウの左下隅では、アプリケーションのバージョン情報を確認できます。以下のメトリックをモニターして、アプリケーションの実行状況のスナップショットを取得することもできます。

- メモリー使用量の円グラフ
- サーバー・プロセスでのアクティブなスレッド数

ログ設定の構成

「ログおよびトゥームストーン」タブを使用して、デフォルト・ロガーを使用可能または使用不可に設定し、保存するログ・ファイルの最大数を指定します。

手順

1. 「ダッシュボード」ウィンドウで、**「アクション」 > 「サーバー詳細の表示」**をクリックするか、**「詳細の表示」**をクリックします。
2. **「ログおよびトゥームストーン」**タブをクリックします。
3. 以下の設定を定義します。

オプション	説明
デフォルト・ロガー	デフォルト・ロガーを使用可能にします。 **注:** AssemblyLine ごとに個別のログ・ファイルが必要な場合は、実行されるすべての AssemblyLine に付加されるデフォルト・ロガーを使用します。
ログ・ファイルの最大数	ロガーが保存できるログ・ファイルの最大数を指定します。

4. **「更新」**をクリックします。

トゥームストーンの構成

「ログおよびトゥームストーン」タブを使用して、トゥームストーン・レコードを作成するダッシュボード・トゥームストーン・マネージャーを開始します。

このタスクについて

ダッシュボードのトゥームストーン・マネージャーは、各 AssemblyLine が強制終了されるときにそのトゥームストーン・レコードを作成します。トゥームストーン・レコードには、統計 (イテレーターにより読み取られた項目の数、スキップされたレコードの数、更新されたレコードの数など) が含まれています。統計は、ワークロードを時間の経過に従ってグラフィカルに表示するために使用できます。

手順

1. 「ダッシュボード」ウィンドウで、**「アクション」 > 「サーバー詳細の表示」**をクリックするか、**「詳細の表示」**をクリックします。
2. トゥームストーン・レコードを生成するには、**「トゥームストーン・マネージャーの開始」**をクリックします。

   注: トゥームストーン・マネージャーを停止するには、IBM Security Directory Integrator サーバーを再始動します。

ダッシュボード・セキュリティー設定の構成

「セキュリティーおよび接続」タブを使用して、ローカル接続とリモート接続の両方のユーザーを認証するために LDAP サーバーを構成します。

手順

1. 「ダッシュボード」ウィンドウで、**「アクション」 > 「サーバー詳細の表示」**をクリックするか、**「詳細の表示」**をクリックします。
2. **「セキュリティーおよび接続」**タブをクリックします。
3. 以下の設定を定義します。

オプション	説明
ローカル	ローカル・ホストからの接続を指定します。
リモート	非ローカル・ホストからの接続を指定します。
LDAP ホスト名	LDAP サーバーの名前を指定します。
LDAP 検索ベース	ユーザーを見つけるための LDAP 検索ベース名を指定します。
LDAP Group DN	ユーザーのメンバーシップを検査するための LDAP Group DN 名を指定します。

4. **「更新」**をクリックします。

インストール済みコンポーネントの表示

「インストール済みコンポーネント」タブを使用して、IBM Security Directory Integrator サーバー上にインストールされているコネクターやパーサーなどのコンポーネントを表示します。

手順

1. 「ダッシュボード」ウィンドウで、**「アクション」 > 「サーバー詳細の表示」**をクリックするか、**「詳細の表示」**をクリックします。
2. コンポーネントのリストを表示するには、**「インストール済みコンポーネント」**タブをクリックします。

システム・ストア・データの表示

「サーバー・ストア」タブを使用して、IBM Security Directory Integrator サーバーが使用しているさまざまなシステム・ストアの内容を表示します。

手順

1. 「ダッシュボード」ウィンドウで、「**アクション**」 > 「**サーバー詳細の表示**」をクリックするか、「**詳細の表示**」をクリックします。
2. サーバー・ストアのリストを表示するには、「**サーバー・ストア**」タブをクリックします。

ダッシュボード RunReport

ダッシュボード RunReport 機能を使用して、AssemblyLine 実行履歴の自動化された E メール・レポートを作成します。

設定したスケジュールに基づいて実行される AssemblyLine を作成して、モニター対象の AssemblyLine の自動化された E メール・レポートを生成します。RunReport AssemblyLine は、IBM Security Directory Integrator トゥームストーン・データベースを使用して、RunReport が最後に実行された後で AssemblyLine が実行されたかどうかを判別します。

E メール・レポートには、モニター対象の AssemblyLine の実行が成功したかどうかを示す実行履歴についての情報が記載されています。このレポートは、指定した受信者に定期的にメール送信されます。

RunReport の作成

「RunReport の作成」ウィンドウを使用して、RunReport AssemblyLine を作成します。この AssemblyLine は、自動化された E メール・レポートを生成するために使用されます。E メール・レポートは、モニター対象の AssemblyLine の実行履歴についての情報を提供し、指定した受信者に定期的に送信されます。

RunReport の作成およびスケジューリング

手順

1. ソリューション・エディターから、「**アクション**」 > 「**RunReport の作成**」をクリックします。
2. 「RunReport の作成」ウィンドウで、RunReport AssemblyLine の名前を「**名前**」フィールドに入力します。
3. 「**OK**」をクリックします。
4. AssemblyLine エディターで、以下の設定を定義します。

表 16. RunReport のスケジューリング・オプション

設定	オプション	説明
スケジュール	**ソリューションの開始時に自動的に開始する**	選択した実行オプションに基づいて AssemblyLine を実行します。
	既に実行中の場合は開始しない	
	AssemblyLine が失敗した場合にスケジュールを強制終了する	
月	**毎月**	スケジュールを実行する月を指定します。
	選択された月	

表 16. *RunReport* のスケジューリング・オプション *(続き)*

設定	オプション	説明
日	**毎日**	スケジュールを実行する日を指定します。
	平日	
	選択された日	
時/分/秒	**時**	スケジュールを実行する時刻を指定します。
	分	
	秒	
件名 (成功)		モニター対象の AssemblyLine の実行が成功したことを示す E メール件名行を指定します。 **注:** 生成されるレポートでは、最後に生成されたレポート以降にモニター対象の AssemblyLine の実行が成功したかどうかに基づいて、件名行が使用されます。
件名 (失敗)		モニター対象の AssemblyLine の実行が失敗したことを示す E メール件名行を指定します。
差出人アドレス		差出人の E メール・アドレスを指定します。
受信者のアドレス		受信者の E メール・アドレスを指定します。 **注:** E メール・アドレスが複数ある場合は、コンマまたはセミコロンで区切られたリストを指定します。
SMTP ホスト		SMTP 接続を受け入れ、メールを受信者に転送する SMTP ホスト・パラメーター
AssemblyLine のモニター		モニター対象の AssemblyLine 名のコンマ区切りリスト
ロギングの共用		スケジューラーと AssemblyLine の間でロギングを共有するかどうかを指定します。

5. 「**保管**」をクリックします。

スケジュールの削除

手順

1. ツリー構造から RunReport AssemblyLine を選択します。
2. AssemblyLine エディターで、「**スケジュール**」タブをクリックします。
3. 「**スケジュールの削除**」をクリックします。

4. 「**保管**」をクリックします。

RunReport スケジューラーの実行と停止

手順

1. ツリー構造から RunReport AssemblyLine を選択します。
2. AssemblyLine エディターで、「**実行**」タブをクリックします。
3. AssemblyLine を実行するには、「**実行**」をクリックします。出力はログ・ビューアーに表示されます。
4. AssemblyLine を停止するには、「**停止**」をクリックします。

コネクター・データの構成およびブラウズ

ソリューション・エディターの「データのブラウズ」ページを使用して、コネクターを構成し、コネクターのデータ・ソースをブラウズし、データ統合ソリューションを作成します。

手順

1. 「ダッシュボード」ウィンドウのナビゲーション・ペインで、「**データのブラウズ**」をクリックします。
2. 「データのブラウズ」ウィンドウで、「**コネクター**」リストからコネクターを選択して、その内容を構成およびブラウズします。
3. 「**パーサー**」リストからパーサーを選択します。

   注: 「**パーサー**」リストは、選択したコネクターがパーサーを必要とする場合、またはパーサーを使用できる場合にのみアクティブになります。
4. ウィンドウの右側にあるフォームでコネクターの詳細を構成します。
5. 選択したコネクターのソリューションを作成するには、「**統合の作成**」をクリックします。
6. 接続を確立し、コネクターの最初の 25 個のレコードを読むには、「**接続**」をクリックします。
7. 次の 25 個のレコードを表示するには、「**次へ**」をクリックします。
8. データ・レコード・ビューを切り替えて一度に 1 つのレコードを表示するには、「**ビューの切り替え**」をクリックします。
9. 切り替えたビューで次のレコードを表示するには、「**次へ**」をクリックします。
10. 選択した属性のみをテーブルに表示するには、以下を実行します。
    a. 「データのブラウズ」ウィンドウの「**オプションの構成**」アイコンをクリックします。
    b. 「オプションの構成」ウィンドウで、属性を選択します。
    c. 属性の順序を変更するには、上向きまたは下向き矢印をクリックします。
    d. 「**OK**」をクリックします。

ソリューション・モニター

ダッシュボード・モニター・エディターを使用して、AssemblyLine 実行の進行状況を追跡し、ソリューションの実行履歴を表示します。

モニター・エディターでは、選択したソリューションの AssemblyLine を確認できます。AssemblyLine の現在および過去の両方のパフォーマンスの情報が表示されます。以下のアクティビティーをモニターできます。

- AssemblyLine のリアルタイム状況モニター。以下の情報が含まれます。
  - AssemblyLine 実行の開始時刻
  - AssemblyLine 実行の終了時刻
  - AssemblyLine の次の実行がスケジュールされている時刻
  - AssemblyLine 内で処理されるデータ・オブジェクトの数
- AssemblyLine の実行履歴。ワークロードとして時間の経過に従ってグラフィカルに表示されます。
- 各 AssemblyLine 実行の結果および問題を記録するログ・ファイル。
- ソリューション内のすべての AssemblyLine のトゥームストーン・レコード。
- サーバー・ログ・ファイル (`ibmdi.log`)。

AssemblyLine の開始および停止

ダッシュボード・モニター・エディターを使用して、AssemblyLine を開始および停止します。

手順

1. ダッシュボードで、インストール済みのソリューションを選択します。
2. 「**ソリューション**」 > 「**モニター**」をクリックします。

   または

   a. 選択したソリューションを右クリックします。

   b. メニューから、「**モニター**」をクリックします。
3. モニター・エディターで、AssemblyLine を選択して、以下のアクションを実行します。
   - AssemblyLine を開始するには、「**開始**」をクリックします。
   - AssemblyLine を停止するには、「**停止**」をクリックします。

   実行の詳細がエディターに表示されます。AssemblyLine 名に対応する緑のアイコンは、その AssemblyLine がアクティブであることを示します。

AssemblyLine 実行履歴の表示

ダッシュボード・モニター・エディターを使用して、AssemblyLine 実行履歴をグラフの形式で表示します。

手順

1. モニター・エディターで、AssemblyLine を選択します。

2. 「**履歴**」をクリックするか、選択した AssemblyLine をダブルクリックします。AssemblyLine 実行の詳細が、グラフィカル・ビューにワークロードとして時間の経過に従って表示されます。
3. 以下のカウンターから任意のものを選択して、グラフ化します。
   - 取得
   - エラー
   - 追加
   - 削除
   - ルックアップ
   - 変更
4. 特定の実行のログ・ファイルを表示するには、グラフ上のノードをクリックします。その実行の実行統計をツールチップとして表示することもできます。

トゥームストーン・レコードの表示

ダッシュボード・モニター・エディターの「トゥームストーン」タブを使用して、AssemblyLine の実行が完了したときに作成されるトゥームストーン・レコードを表示します。

手順

1. ダッシュボードのモニター・エディターに移動します。
2. 記録されたトゥームストーンを表示するには、「**トゥームストーン**」タブをクリックします。

   注: トゥームストーン・レコードは、トゥームストーン・マネージャーが実行されているときにのみ表示できます。詳しくは、247 ページの『トゥームストーンの構成』のトピックを参照してください。

ログ・ファイルの表示

ダッシュボード・モニター・エディターの「ログ・ファイル」タブを使用して、各 AssemblyLine 実行で作成されるログ・ファイルを表示します。ログ・ファイルは、問題を素早く簡単に分析し、すべての問題をデバッグするために使用されます。

手順

1. モニター・エディターで、「**ログ・ファイル**」タブをクリックします。すべての AssemblyLine 実行のログ・ファイルが、ツリー構造として表示されます。
2. ログ・ファイルの内容を表示するには、ログ・ファイルをツリー構造から選択し、ダブルクリックします。
3. ログ内容に含まれる特定の情報を検索するには、検索するテキストを「**検索**」フィールドに入力します。
4. メッセージ・ソース (メッセージをログに記録したコンポーネント) を組み込むには、「**ソースの組み込み**」チェック・ボックスを選択します。
5. 「**オプション**」をクリックして、以下の詳細を設定します。

オプション	説明
ページ・サイズ	

オプション	説明
メッセージ・タイプ	以下のメッセージ・タイプから任意のものを選択して、ログ・ファイルに組み込みます。 • INFO • WARN • ERROR • DEBUG
表示オプション	ログ・ファイルに日付、時刻、メッセージ・ソース、または行番号を表示するオプションを選択します。 • 日付の表示 • 時刻の表示 • ソースの組み込み • 行番号の表示

特記事項

本書は米国 IBM が提供する製品およびサービスについて作成したものです。本書に記載の製品、サービス、または機能が日本においては提供されていない場合があります。日本で利用可能な製品、サービス、および機能については、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、またはサービスのみが使用可能であることを意味するものではありません。これらに代えて、IBM の知的所有権を侵害することのない、機能的に同等の製品、プログラム、またはサービスを使用することができます。ただし、IBM 以外の製品とプログラムの操作またはサービスの評価および検証は、お客様の責任で行っていただきます。

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む) を保有している場合があります。本書の提供は、お客様にこれらの特許権について実施権を許諾することを意味するものではありません。実施権についてのお問い合わせは、書面にて下記宛先にお送りください。

〒103-8510
東京都中央区日本橋箱崎町19番21号
日本アイ・ビー・エム株式会社
法務・知的財産
知的財産権ライセンス渉外

以下の保証は、国または地域の法律に沿わない場合は、適用されません。

IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態で提供し、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。

国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

この情報には、技術的に不適切な記述や誤植を含む場合があります。本書は定期的に見直され、必要な変更は本書の次版に組み込まれます。 IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものではありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムとその他のプログラム (本プログラムを含む) との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、下記に連絡してください。

IBM Corporation
2Z4A/101
11400 Burnet Road
Austin, TX 78758 U.S.A.

本プログラムに関する上記の情報は、適切な使用条件の下で使用することができますが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、またはそれと同等の条項に基づいて、IBM より提供されます。

この文書に含まれるいかなるパフォーマンス・データも、管理環境下で決定されたものです。そのため、他の操作環境で得られた結果は、異なる可能性があります。一部の測定が、開発レベルのシステムで行われた可能性がありますが、その測定値が、一般に利用可能なシステムのものと同じである保証はありません。さらに、一部の測定値が、推定値である可能性があります。実際の結果は、異なる可能性があります。お客様は、お客様の特定の環境に適したデータを確かめる必要があります。

IBM 以外の製品に関する情報は、その製品の供給者、出版物、もしくはその他の公に利用可能なソースから入手したものです。IBM は、それらの製品のテストは行っておりません。したがって、他社製品に関する実行性、互換性、またはその他の要求については確証できません。 IBM 以外の製品の性能に関する質問は、それらの製品の供給者にお願いします。

IBM の将来の方向性および指針に関するすべての記述は、予告なく変更または撤回される場合があります。これらは目標および目的を提示するものにすぎません。

表示されている IBM の価格は IBM が小売り価格として提示しているもので、現行価格であり、通知なしに変更されるものです。卸価格は、異なる場合があります。

本書はプランニング目的としてのみ記述されています。記述内容は製品が使用可能になる前に変更になる場合があります。

本書には、日常の業務処理で用いられるデータや報告書の例が含まれています。より具体性を与えるために、それらの例には、個人、企業、ブランド、あるいは製品などの名前が含まれている場合があります。これらの名称はすべて架空のものであり、名称や住所が類似する企業が実在しているとしても、それは偶然にすぎません。

著作権使用許諾:

本書には、様々なオペレーティング・プラットフォームでのプログラミング手法を例示するサンプル・アプリケーション・プログラムがソース言語で掲載されています。お客様は、サンプル・プログラムが書かれているオペレーティング・プラット

フォームのアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。このサンプル・プログラムは、あらゆる条件下における完全なテストを経ていません。従って IBM は、これらのサンプル・プログラムについて信頼性、利便性もしくは機能性があることをほのめかしたり、保証することはできません。お客様は、IBM のアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。

それぞれの複製物、サンプル・プログラムのいかなる部分、またはすべての派生的創作物にも、次のように、著作権表示を入れていただく必要があります。

© (お客様の会社名) (西暦年). このコードの一部は、IBM Corp. のサンプル・プログラムから取られています。© Copyright IBM Corp. _年を入れる_. All rights reserved.

この情報をソフトコピーでご覧になっている場合は、写真やカラーの図表は表示されない場合があります。

商標

IBM、IBM ロゴおよび ibm.com® は、世界の多くの国で登録された International Business Machines Corporation の商標です。 他の製品名およびサービス名等は、それぞれ IBM または各社の商標である場合があります。現時点での IBM の商標リストについては、http://www.ibm.com/legal/copytrade.shtml をご覧ください。

Adobe、PostScript は、Adobe Systems Incorporated の米国およびその他の国における登録商標または商標です。

IT Infrastructure Library は英国 Office of Government Commerce の一部である the Central Computer and Telecommunications Agency の登録商標です。

インテル、Intel、Intel ロゴ、Intel Inside、Intel Inside ロゴ、Centrino、Intel Centrino ロゴ、Celeron、Xeon、Intel SpeedStep、Itanium、Pentium は、Intel Corporation または子会社の米国およびその他の国における商標または登録商標です。

Linux は、Linus Torvalds の米国およびその他の国における登録商標です。

Microsoft、Windows、Windows NT および Windows ロゴは、Microsoft Corporation の米国およびその他の国における商標です。

ITIL は英国 The Minister for the Cabinet Office の登録商標および共同体登録商標であって、米国特許商標庁にて登録されています。

UNIX は The Open Group の米国およびその他の国における登録商標です。

Java およびすべての Java 関連の商標およびロゴは Oracle やその関連会社の米国およびその他の国における商標または登録商標です。

Cell Broadband Engine は、Sony Computer Entertainment, Inc.の米国およびその他の国における商標であり、同社の許諾を受けて使用しています。

Linear Tape-Open、LTO、LTO ロゴ、Ultrium、および Ultrium ロゴは、HP、IBM Corp. および Quantum の米国およびその他の国における商標です。

索引

日本語, 数字, 英字, 特殊文字の順に配列されています。なお, 濁音と半濁音は清音と同等に扱われています。

[ア行]

[カ行]

[サ行]

[タ行]

[ナ行]

[ハ行]

[マ行]

[ヤ行]

[ラ行]

[ワ行]

A

B

C

E

F

I

J

L

N

R

S

T

U

W

Printed in Japan

SC88-8416-03

日本アイ・ビー・エム株式会社
〒103-8510 東京都中央区日本橋箱崎町19-21