

InnoRules

Application Programming Interface Guide

InnoRules v7.2



InnoRules Version 7.2 Copyright © 2007 INNORULES Corporation. All rights reserved.

InnoRules® and InnoProduct® are registered trademarks of INNORULES Corporation.

目次

[1. Java API 11](#_Toc40799093)

[1.1 必要なライブラリ 11](#_Toc40799094)

[1.1.1 ビルドライブラリ 11](#_Toc40799095)

[1.1.2 ランタイムライブラリ 11](#_Toc40799096)

[1.2 ルールサービス呼び出しAPI 11](#_Toc40799097)

[1.2.1 ルールインタフェースの取得 11](#_Toc40799098)

[1.2.2 com.innoexpert.rulesclient.RuleInterface 14](#_Toc40799099)

[1.2.3 コード体系 16](#_Toc40799100)

[1.2.4 データ型と関連定数 17](#_Toc40799101)

[1.2.5 数値型モード 18](#_Toc40799102)

[1.2.6 I/Oオブジェクト方式 18](#_Toc40799103)

[1.2.7 I/Oアダプタ方式 33](#_Toc40799104)

[1.2.8 ルールエンジンの構成方式に基づくI/Oオブジェクト方式とI/Oアダプタ方式の性能比較 48](#_Toc40799105)

[1.3 コールバックAPI 50](#_Toc40799106)

[1.3.1 コールバックルール 50](#_Toc40799107)

[1.3.2 コールバックサービスの処理フロー 51](#_Toc40799108)

[1.3.3 com.innoexpert.rulesclient.CallbackHandler 52](#_Toc40799111)

[1.3.4 com.innoexpert.rulesclient.CallbackReq 55](#_Toc40799112)

[1.3.5 com.innoexpert.rulesclient.CallbackRes 57](#_Toc40799113)

[1.3.6 com.innoexpert.rulesclient.CallbackParam 58](#_Toc40799114)

[1.3.7 com.innoexpert.rulesclient.CallbackHandlerFactory 62](#_Toc40799115)

[1.3.8 com.innoexpert.rulesclient.CallbackException 67](#_Toc40799116)

[2. C/C++ API 68](#_Toc40799117)

[2.1 ルールサービス呼び出しAPI 68](#_Toc40799118)

[2.1.1 APIの呼び出し順序 68](#_Toc40799119)

[2.1.2 ライブラリとアプリケーションビルド 70](#_Toc40799120)

[2.1.3 ハンドルとディスクリプタ 71](#_Toc40799121)

[2.1.4 定数と列挙型(enum) 71](#_Toc40799122)

[2.1.5 ハンドル関連の関数 73](#_Toc40799123)

[2.1.6 ルール入力設定関数 74](#_Toc40799124)

[2.1.7 ルールサービス呼び出しおよびルール結果照会関数 75](#_Toc40799125)

[2.1.8 ルールインタフェースクラスタの属性文字列 79](#_Toc40799126)

[2.2 コールバックAPI 82](#_Toc40799127)

[2.2.1 コールバックサービスハンドラー 82](#_Toc40799128)

[2.2.2 コールバックパラメータの照会および値の設定 83](#_Toc40799129)

[3. C# 86](#_Toc40799130)

[3.1 必要なライブラリ 86](#_Toc40799131)

[3.2 初期化モジュール 86](#_Toc40799132)

[3.2.1 初期化モジュールの呼び出し順序 86](#_Toc40799133)

[3.2.2 InnoRules.Client.SimpleInitializer 88](#_Toc40799134)

[3.2.3 InnoRules.Resources.Msg 89](#_Toc40799135)

[3.2.4 InnoRules.RulesClient.Release 90](#_Toc40799136)

[3.3 ルールサービス呼び出しAPI 90](#_Toc40799137)

[3.3.1 ルールインタフェースの取得 90](#_Toc40799138)

[3.3.2 InnoRules.RulesClient.RuleInterface 93](#_Toc40799139)

[3.3.3 コード体系 94](#_Toc40799140)

[3.3.4 データ型と関連定数 95](#_Toc40799141)

[3.3.5 I/Oオブジェクト方式 96](#_Toc40799142)

[3.3.6 I/Oアダプタ方式 108](#_Toc40799143)

[4. REST API 121](#_Toc40799144)

[4.1 HTTP Client 121](#_Toc40799145)

[4.1.1 URL 121](#_Toc40799146)

[4.1.2 HTTPヘッダー 122](#_Toc40799147)

[4.1.3 HTTPメソッド 122](#_Toc40799148)

[4.2 JSONメッセージ 122](#_Toc40799149)

[4.2.1 リクエストメッセージ 122](#_Toc40799150)

[4.2.2 レスポンスメッセージ 123](#_Toc40799151)

[4.2.3 エラーメッセージ 124](#_Toc40799152)

[4.2.4 特殊文字 124](#_Toc40799153)

[5. ルールアプリケーション開発イシュー 125](#_Toc40799154)

[5.1 数値型イシュー 125](#_Toc40799155)

[5.1.1 hello-double.c 125](#_Toc40799156)

[5.1.2 2進法での表記 126](#_Toc40799157)

[5.1.3 IEEE754倍精度実数型 127](#_Toc40799158)

[5.1.4 演算の誤差・累積 129](#_Toc40799159)

[5.1.5 文字列への変換 129](#_Toc40799160)

[5.1.6 整数変換および整数演算 134](#_Toc40799161)

[5.1.7 誤差の補正 135](#_Toc40799162)

[5.1.8 BigDecimal関連イシュー 136](#_Toc40799163)

[5.1.9 結論 137](#_Toc40799164)

[5.2 ルールインタフェース 137](#_Toc40799165)

[5.3 ルール呼び出し回数のイシュー 139](#_Toc40799166)

[5.4 アプリケーションフレームワークとの連動に関するイシュー 140](#_Toc40799167)

[5.5 性能イシュー 140](#_Toc40799168)

[5.6 業務実装におけるルールカバレッジのイシュー 140](#_Toc40799169)

[6. 必要なライブラリ 142](#_Toc40799170)

[6.1 ビルドライブラリ 142](#_Toc40799171)

[6.2 ランタイムライブラリ 142](#_Toc40799172)

[7. 簡単な例 143](#_Toc40799173)

[7.1 関数の原型(Prototype) 143](#_Toc40799174)

[7.2 関数のコード 143](#_Toc40799175)

[7.3 ヘルプファイル 146](#_Toc40799176)

[7.4 関数のインストールと設定 147](#_Toc40799177)

[7.5 ヘルプの照会およびルールの作成 147](#_Toc40799178)

[8. ユーザ定義関数APIクラス 150](#_Toc40799179)

[8.1 com.innoexpert.innorulesj.suite.udf.UserDefinedFunction 150](#_Toc40799180)

[8.2 com.innoexpert.innorulesj.abstractcompiler.Type 152](#_Toc40799181)

[8.3 com.innoexpert.innorulesj.suite.udf.Engine 155](#_Toc40799182)

[8.4 com.innoexpert.innorulesj.runtime.SOValue 158](#_Toc40799183)

[8.5 com.innoexpert.innorulesj.runtime.MOValue 160](#_Toc40799184)

[8.6 例外クラス 163](#_Toc40799185)

[8.6.1 com.innoexpert.innorulesj.exceptions.CompileException 163](#_Toc40799186)

[8.6.2 com.innoexpert.suite.udf.UserDefinedFunctionException 163](#_Toc40799187)

[9. ユーザ定義関数の開発フレームワーク 164](#_Toc40799188)

[9.1 ビルドライブラリ 164](#_Toc40799189)

[9.2 ランタイムライブラリ 164](#_Toc40799190)

[9.3 ユーザ定義関数フレームワークAPIクラス 164](#_Toc40799191)

[9.3.1 com.innoexpert.innorulesj.suite.udfunit.UdfUnit 164](#_Toc40799192)

[9.4 簡単な例 166](#_Toc40799193)

[10. 定数ルールとユーザ定義関数 168](#_Toc40799194)

[10.1 定数ルール 168](#_Toc40799195)

[10.2 RANDOM関数 169](#_Toc40799196)

[10.3 ビルトイン関数に対するユーザ定義関数の制約事項 171](#_Toc40799197)

図目次

[[図 1 I/Oオブジェクト方式のルールサービス呼び出し] 18](#_Toc40799198)

[[図 2 I/Oアダプタ方式のルールサービスの呼び出し] 34](#_Toc40799199)

[[図 3 ItemProviderダイアグラム] 36](#_Toc40799200)

[[図 4 ResultConsumerダイアグラム] 40](#_Toc40799201)

[[図 5 I/Oオブジェクト方式とI/Oアダプタ方式の比較システム] 48](#_Toc40799202)

[[図 6 コールバックルール] 50](#_Toc40799203)

[[図 7 ルールAPIライブラリの構成] 86](#_Toc40799204)

[[図 8 I/Oオブジェクト方式のルールサービスの呼び出し] 96](#_Toc40799205)

[[図 9 I/Oアダプタ方式のルールサービスの呼び出し] 108](#_Toc40799206)

[[図 10 ItemProviderダイアグラム] 110](#_Toc40799207)

[[図 11 ResultConsumerダイアグラム] 114](#_Toc40799208)

[[図 12 9986.21 \* 100 – 998620] 131](#_Toc40799209)

[[図 13 書式 - 標準] 132](#_Toc40799210)

[[図 14 書式 - 小数点以下15桁] 133](#_Toc40799211)

[[図 15 小数点以下15桁までの結果] 133](#_Toc40799212)

[[図 16 ユーザ定義関数リスト] 147](#_Toc40799213)

[[図 17 FACT関数のヘルプ] 148](#_Toc40799214)

[[図 18 FACT関数を使用するテーブルルール] 148](#_Toc40799215)

[[図 19 {CallFACT}ルールのテストの結果] 149](#_Toc40799216)

[[図 20 正しくないFACT関数パラメータに対するエラーメッセージ] 149](#_Toc40799217)

[[図 21 非定数ルール] 168](#_Toc40799218)

[[図 22 定数ルール] 168](#_Toc40799219)

[[図 23 CallRANDOMルール] 170](#_Toc40799220)

[[図 24 ルール実行結果] 170](#_Toc40799221)

[[図 25 CallSYSTIME] 171](#_Toc40799222)

[[図 26 非定数ルールCallRANDOM] 171](#_Toc40799223)

表目次

[[表 1 com.innoexpert.rulesclient.ClusterManager] 12](#_Toc40799224)

[[表 2 com.innoexpert.rulesclient.InterfaceCluster] 12](#_Toc40799225)

[[表 3 ルールインタフェースを取得して返すサンプル] 13](#_Toc40799226)

[[表 4 com.innoexpert.rulesclient.ClusterManager.getInterface()] 14](#_Toc40799227)

[[表 5 ルールインタフェースを取得して返すサンプル] 14](#_Toc40799228)

[[表 6 com.innoexpert.rulesclient.RuleInterface] 15](#_Toc40799229)

[[表 7 コード体系を指定する定数値] 17](#_Toc40799230)

[[表 8 データ型定数] 17](#_Toc40799231)

[[表 9 数値型モード] 18](#_Toc40799232)

[[表 10 I/Oオブジェクト方式のexecuteメソッド] 19](#_Toc40799233)

[[表 11 com.innoexpert.rulesclient.RuleReq] 19](#_Toc40799234)

[[表 12 com.innoexpert.rulesclient.Item] 22](#_Toc40799235)

[[表 13 NumberItemの例] 24](#_Toc40799236)

[[表 14 簡略化されたNumberItemの例] 24](#_Toc40799237)

[[表 15 com.innoexpert.rulesclient.ResultSet] 25](#_Toc40799238)

[[表 16 com.innoexpert.rulesclient.ResultSetMetaData] 29](#_Toc40799239)

[[表 17 ルールサービス呼び出し例] 30](#_Toc40799240)

[[表 18 I/Oアダプタ方式のexecuteメソッド] 34](#_Toc40799241)

[[表 19 com.innoexpert.rulesclient.NumericModeSelector] 35](#_Toc40799242)

[[表 20 com.innoexpert.rulesclient.ItemProvider] 36](#_Toc40799243)

[[表 21 アプリケーションデータオブジェクト例 - Javaプロパティ] 37](#_Toc40799244)

[[表 22 ItemProvider の実装例 – PropertiesProvider] 37](#_Toc40799245)

[[表 23 com.innoexpert.rulesclient.ResultConsumer] 40](#_Toc40799246)

[[表 24 ResultConsumer の実装例 – PropertyConsumer] 42](#_Toc40799247)

[[表 25 I/Oアダプタ方式のexecuteメソッド] 45](#_Toc40799248)

[[表 26 ルールサービス呼び出し例] 46](#_Toc40799249)

[[表 27 テストシステム構成] 49](#_Toc40799250)

[[表 28 I/Oオブジェクト方式とI/Oアダプタ方式の比較結果] 49](#_Toc40799251)

[[表 29 com.innoexpert.rulesclient.CallbackHandler] 53](#_Toc40799252)

[[表 30 アプリのデータベースコネクションをコールバックコンテキストオブジェクトに引き渡す例] 54](#_Toc40799253)

[[表 31 アプリケーショントランザクション上でクエリをするコールバックハンドラー] 54](#_Toc40799254)

[[表 32 com.innoexpert.rulesclient.CallbackReq] 55](#_Toc40799255)

[[表 33 lookupParamメソッドを利用した必須コールバック入力パラメータの照会] 57](#_Toc40799256)

[[表 34 lookupRequiredParamメソッドを利用したコードの簡略化] 57](#_Toc40799257)

[[表 35 com.innoexpert.rulesclient.CallbackRes] 57](#_Toc40799258)

[[表 36 com.innoexpert.rulesclient.CallbackParam] 58](#_Toc40799259)

[[表 37 com.innoexpert.rulesclient.CallbackHandlerFactory] 62](#_Toc40799260)

[[表 38 シングルトンハンドラー方式の例] 63](#_Toc40799261)

[[表 39 シングルトンハンドラー方式の別の例] 64](#_Toc40799262)

[[表 40 コールバックサービスごとに新しいコールバックハンドラーオブジェクトを生成する方式の例] 65](#_Toc40799263)

[[表 41 ハンドラープール方式の例] 66](#_Toc40799264)

[[表 42 com.innoexpert.rulesclient.CallbackException] 67](#_Toc40799265)

[[表 43 ルールサービスを呼び出す簡単なCアプリケーション] 68](#_Toc40799266)

[[表 44 データ型の再定義] 71](#_Toc40799267)

[[表 45 データ型定数] 71](#_Toc40799268)

[[表 46 ハンドル類型定数] 72](#_Toc40799269)

[[表 47 便利な定数] 72](#_Toc40799270)

[[表 48 コード体系の列挙型] 72](#_Toc40799271)

[[表 49 ハンドル関連の関数] 73](#_Toc40799272)

[[表 50 項目値設定関数] 74](#_Toc40799273)

[[表 51 ルールサービス呼び出しおよびルール結果照会関数] 75](#_Toc40799274)

[[表 52 属性文字列の形式] 79](#_Toc40799275)

[[表 53 コールバックサービスハンドラー] 82](#_Toc40799276)

[[表 54 コールバックサービスハンドラー登録関数] 82](#_Toc40799277)

[[表 55 コールバック入力パラメータの照会] 83](#_Toc40799278)

[[表 56 ルールAPIライブラリの構成] 86](#_Toc40799279)

[[表 57 設定ファイルによるルールサービス初期化アプリケーション] 87](#_Toc40799280)

[[表 58 Propertiesオブジェクトによるルールサービス初期化アプリケーション] 87](#_Toc40799281)

[[表 59 InnoRules.Client.SimpleInitializer] 88](#_Toc40799282)

[[表 60 クラスタ設定情報] 89](#_Toc40799283)

[[表 61 InnoRules.Resources.Msg] 89](#_Toc40799284)

[[表 62 InnoRules.Client.Release] 90](#_Toc40799285)

[[表 63 InnoRules.RulesClient.ClusterManager] 90](#_Toc40799286)

[[表 64 InnoRules.RulesClient.InterfaceCluster] 91](#_Toc40799287)

[[表 65 ルールインタフェースを取得して返すサンプル] 92](#_Toc40799288)

[[表 66 InnoRules.RulesClient.ClusterManager.GetInterface()] 92](#_Toc40799289)

[[表 67 ルールインタフェースを取得して返すサンプル] 93](#_Toc40799290)

[[表 68 InnoRules.RulesClient.RuleInterface] 93](#_Toc40799291)

[[表 69 コード体系を指定する定数の値] 95](#_Toc40799292)

[[表 70 データ型定数] 95](#_Toc40799293)

[[表 71 I/Oオブジェクト方式のexecuteメソッド] 96](#_Toc40799294)

[[表 72 InnoRules.Tcp.RuleReq] 97](#_Toc40799295)

[[表 73 InnoRules.RulesClient.Item] 99](#_Toc40799296)

[[表 74 NumberItemの例] 101](#_Toc40799297)

[[表 75 簡略化されたNumberItemの例] 101](#_Toc40799298)

[[表 76 InnoRules.Client.ResultSet] 101](#_Toc40799299)

[[表 77 InnoRules.Client.ResultSetMetaData] 105](#_Toc40799300)

[[表 78 ルールサービス呼び出し例] 106](#_Toc40799301)

[[表 79 I/Oアダプタ方式のexecuteメソッド] 109](#_Toc40799302)

[[表 80 InnoRules.RulesClient.ItemProvider] 110](#_Toc40799303)

[[表 81 アプリケーションデータオブジェクトの事例] 111](#_Toc40799304)

[[表 82 ItemProviderの実装事例 – HashtableProvider] 112](#_Toc40799305)

[[表 83 InnoRules.RulesClient.ResultConsumer] 114](#_Toc40799306)

[[表 84 ResultConsumerの実装事例 – HashtableConsumer] 116](#_Toc40799307)

[[表 85 I/Oアダプタ方式のExecuteメソッド] 117](#_Toc40799308)

[[表 86 ルールサービス呼び出し例] 118](#_Toc40799309)

[[表 87 リクエストメッセージ例] 122](#_Toc40799310)

[[表 88 単一結果値のレスポンスメッセージ例] 123](#_Toc40799311)

[[表 89 多重結果値のレスポンスメッセージ例] 123](#_Toc40799312)

[[表 90 エラーメッセージ例] 124](#_Toc40799313)

[[表 91 hello-double.c] 125](#_Toc40799314)

[[表 92 小数点以下20桁まで出力] 126](#_Toc40799315)

[[表 93 10進数11を2進数に変換] 126](#_Toc40799316)

[[表 94 10進数0.125の2進法での表記] 127](#_Toc40799317)

[[表 95 10進数0.1の2進法での表記] 127](#_Toc40799318)

[[表 96 IEEE754倍精度実数型] 127](#_Toc40799319)

[[表 97 浮動小数点数の代数計算と誤差の累積] 129](#_Toc40799320)

[[表 98 double値を文字列で出力するCソースコード] 129](#_Toc40799321)

[[表 99 double値を文字列で出力するJavaソースコード] 130](#_Toc40799322)

[[表 100 暗黙的な整数型変換の問題] 134](#_Toc40799323)

[[表 101 有効数字の桁数を利用した補正例] 135](#_Toc40799324)

[[表 102 有効数字15桁で補正] 135](#_Toc40799325)

[[表 103 5.1.6の例を有効数字15桁で補正したJavaソースコードと結果] 136](#_Toc40799326)

[[表 104 ルールインタフェースを借りるtry-finallyパターン] 138](#_Toc40799327)

[[表 105 メソッド間のルールインタフェース共有パターン] 138](#_Toc40799328)

[[表 106 FACT 関数の原型] 143](#_Toc40799329)

[[表 107 com.innorules.apiguide.udf.FuncFACT] 143](#_Toc40799330)

[[表 108 ヘルプ XML] 146](#_Toc40799331)

[[表 109 com.innoexpert.innorulesj.suite.udf.UserDefinedFunction] 150](#_Toc40799332)

[[表 110 com.innoexpert.innorulesj.abstractcompiler.Type] 152](#_Toc40799333)

[[表 111 データ型定数 - com.innoexpert.innorulesj.runtime.Types] 153](#_Toc40799334)

[[表 112 com.innoexpert.innorulesj.suite.udf.Engine] 155](#_Toc40799335)

[[表 113 SO型データ] 158](#_Toc40799336)

[[表 114 com.innoexpert.innorulesj.runtime.SOValue] 158](#_Toc40799337)

[[表 115 MO型データ] 160](#_Toc40799338)

[[表 116 com.innoexpert.innorulesj.runtime.MOValue] 160](#_Toc40799339)

[[表 117 com.innoexpert.suite.udf.CompileException] 163](#_Toc40799340)

[[表 118 com.innoexpert.suite.udf.UserDefinedFunctionException] 163](#_Toc40799341)

[[表 119 com.innoexpert.innorulesj.suite.udfunit.UdfUnit] 164](#_Toc40799342)

[[表 120 com.innoexpert.innorulesj.suite.udfunit.UdfUnit] 166](#_Toc40799343)

[[表 121 RANDOM関数] 169](#_Toc40799344)

1. ルールサービスAPI

ルールサービスAPI(Rule Service API)は、ルールサービスを呼び出すためのAPIである。Java、C、C++、C#、RESTfulサービスのためのAPIライブラリを提供している。

# Java API

アプリケーション開発者は、JavaルールサービスAPIを利用してプロトコルに依存しない方法でルールサービスを呼び出すアプリケーションを作成できる。この章ではAPIの主要なクラスについての詳細と使用方法を説明する。

APIは一連のJavaライブラリで構成されており、APIが正常に動作するためにはルールシステムが構成されている必要がある。

ルールサービスAPIはルールを呼び出すためのAPIとコールバックサービスのためのAPIに区分される。

## 必要なライブラリ

### ビルドライブラリ

ルールサービスAPIすべてのクラスとインタフェースはinnorules-api.jarに含まれている。したがって、ルールサービスAPIを使用するアプリケーションをビルドするためにはinnorules-api.jarをビルドパスに設定する必要がある。

### ランタイムライブラリ

ルールサービスAPIのインタフェースはinnorules-api.jarに含まれているが、これをプロトコル別に実装したクラスはinnorulesj.jarまたは別のライブラリに位置する。したがって、ルールアプリケーションを実行するためにはinnorules-api.jar以外にもinnorulesj.jarおよび関連サードパーティライブラリなどがクラスパスに追加設定されている必要がある

Webアプリケーション形式のアプリケーションの場合、Webアプリケーションサーバに既にルールランタイムシステムと必要なライブラリが構成されているためアプリケーション開発者はランタイムライブラリ設定に対して考える必要がない。Webアプリケーションサーバにルールランタイムシステムを構成することの詳細内容は「InnoRules Installation and Operation Guide 2.5 ルールアプリケーション構成ウィザード」を参照のこと。

バッチアプリケーション形式のルールアプリケーション開発者はバッチアプリケーションのクラスパスにランタイムライブラリを追加し、ルールランタイムシステム初期化ルーチンを直接呼び出す必要がある。バッチアプリケーションにルールランタイムシステムを構成する方法については「InnoRules Installation and Operation Guide 2.5 ルールアプリケーション構成ウィザード」を参照のこと。

## ルールサービス呼び出しAPI

### ルールインタフェースの取得

ルールサービスはルールインタフェース(Rule Interface)というルールエンジンプロキシオブジェクトを利用して呼び出す。アプリケーションは、実行環境内に構成されているルールインタフェースクラスタからルールインタフェースを取得してルールサービスを呼び出す。ルールインタフェースクラスタはルールインタフェースクラスタマネージャから取得できる。ルールインタフェースクラスタマネージャは、多数のルールインタフェースクラスタを管理する。各クラスタには名前があり、該当名を使用してクラスタマネージャからクラスタを検索できる。

com.innoexpert.rulesclient.ClusterManagerクラスは、ルールインタフェースクラスタマネージャを表現するクラスである。次の表は、ClusterManagerクラスの一部のメソッドを示す。

[表 1 com.innoexpert.rulesclient.ClusterManager]

|  |
| --- |
| 1 package com.innoexpert.rulesclient;  2  3 public class ClusterManager  4 {  5 public static InterfaceCluster get( String clusterName ) throws RulesException;  6 public static InterfaceCluster get() throws RulesException;  7 …  8 } |

ClusterManagerクラスには、ルールインタフェースクラスタを提供する2つのgetメソッドがある。ルールインタフェースクラスタはcom.innoexpert.rulesclient.InterfaceClusterインタフェースで表現される。

最初のメソッドはクラスタの名前をパラメータとして受け取り、この名前を持つクラスタを返す。指定された名前のクラスタがない場合は、RulesExceptionを投げる。2番目のメソッドはパラメータを受け取らないメソッドで、“rulesclient”という名前のルールインタフェースクラスタを返す。“rulesclient”は、基本ルールインタフェースクラスタの名前である。このメソッドは名前が“rulesclient”であるルールインタフェースクラスタがマネージャに登録されてない場合、RulesExceptionを投げる。

次の表は、InterfaceClusterクラスの一部のメソッドを示す。

[表 2 com.innoexpert.rulesclient.InterfaceCluster]

|  |
| --- |
| 1 package com.innoexpert.rulesclient;  2  3 public interface InterfaceCluster  4 {  5 RuleInterface getInterface() throws RulesException;  6 String getName();  7 …  8 } |

getInterfaceは、ルールインタフェースクラスタからルールインタフェースを取得するメソッドである。

com.innoexpert.rulesclient.RuleInterfaceは、ルールインタフェースを表現するインタフェースである。取得したRuleInterfaceオブジェクトは、使用を終えた後、RuleInterface.close()メソッドを呼び出してルールインタフェースクラスタに返す必要がある。ルールインタフェースクラスタから取得したルールインタフェースを返さない場合、リソースリークの発生や全体的なアプリケーションの性能が低下することがある。ルールインタフェースクラスタを構成するすべてのルールインタフェースプールから有効なルールインタフェースをアプリケーションに返すことができない際に、RulesExceptionが投げられる。次のような場合、ルールインタフェースクラスタは有効なルールインタフェースを返すことができない。

* TCP/IPを利用してリモートルールエンジンを呼び出すルールインタフェースを提供するルールインタフェースクラスタで、リモートルールエンジンに接続できない場合
* ローカルルールエンジンを呼び出すルールインタフェースを提供するルールインタフェースクラスタで、ルールエンジンが使用するルールリポジトリキャッシュが初期化されてない場合

getNameメソッドは、このルールインタフェースクラスタの名前を返す。

次の表は、“dev”というルールインタフェースクラスタからルールインタフェースを取得し、ルールインタフェースを使用後に返すサンプルソースである。サンプルで、finallyブロックでRuleInterface.close()を呼び出すことがポイントである。RuleInterfaceのリークを防ぐ方法として、ルールインタフェースクラスタからルールインタフェースを取得した後、try-finallyブロックでRuleInterface.close()を呼び出すことを推奨する。

[表 3 ルールインタフェースを取得して返すサンプル]

|  |
| --- |
| 1 import com.innoexpert.rulesclient.\*;  2  3 …  4  5 InterfaceCluster cluster = ClusterManager.get( "dev" );  6 RuleInterface intf = cluster.getInterface();  7  8 try  9 {  10 // use RuleInterface object and invoke rule services  11 }  12 finally  13 {  14 intf.close();  15 } |

[表 3 ルールインタフェースを取得して返すサンプル]で、ルールインタフェースを取得するステップは2段階で行われる。最初の段階はルールインタフェースクラスタマネージャから“dev”という名前のルールインタフェースクラスタを取得するもので、2番目の段階はルールインタフェースクラスタからルールインタフェースを取得するものである。ルールインタフェースクラスタマネージャは、この2段階を一度に実行するメソッドを提供する。次の表では二つのgetInterfaceメソッドを提供する。

[表 4 com.innoexpert.rulesclient.ClusterManager.getInterface()]

|  |
| --- |
| 1 package com.innoexpert.rulesclient;  2  3 public class ClusterManager  4 {  5 // equivalent to get( clusterName ).getInterface()  6 public static RuleInterface getInterface( String clusterName )  7 throws RulesException;  8 // equivalent to get().getInterface()  9 public static RuleInterface getInterface() throws RulesException;  10 …  11 } |

このメソッドは、同一パラメータを持つgetメソッドを呼び出した後に返されるInterfaceClusterのgetInterfaceを呼び出すことと同様である。つまり、ClusterManager.get( clusterName ).getInterface()はClusterManager.getInterface( clusterName )と同一である。次の表のソースコードは[表 3 ルールインタフェースを取得して返すサンプル]のコードをこのメソッドを使用して変換したサンプルであり、[表 3 ルールインタフェースを取得して返すサンプル]と同様に動作する。

[表 5 ルールインタフェースを取得して返すサンプル]

|  |
| --- |
| 1 import com.innoexpert.rulesclient.\*;  2  3 …  4  5 RuleInterface intf = ClusterManager.getInterface( "dev" );  6  7 try  8 {  9 // use RuleInterface object and invoke rule services  10 }  11 finally  12 {  13 intf.close();  14 } |

### com.innoexpert.rulesclient.RuleInterface

RuleInterfaceインタフェースにはルールサービスを呼び出せる多様なメソッドが定義されている。このメソッドはルールエンジンとの通信プロトコルに依存しないように定義されている。したがって、プロトコルに依存しないアプリケーションが作成でき、アプリケーションの運用中にプロトコルが変更されてもアプリケーションの修正や再コンパイルが不要である。次の表は、RuleInterfaceに定義されたルールサービス呼び出しメソッドを示す。次に記述されてないメソッドについては技術サポートを受けて使用すること。

[表 6 com.innoexpert.rulesclient.RuleInterface]

|  |
| --- |
| 1 package com.innoexpert.rulesclient;  2  3 public interface RuleInterface  4 {  5 ResultSet execute( RuleReq req, int codetype )  6 throws RulesException, IllegalArgumentException;  7 ResultSet execute( RuleReq req, int codetype, Object ctx )  8 throws RulesException, IllegalArgumentException;  9 void execute( String rule, ItemProvider itemProvider, ResultConsumer consumer,  10 int codetype, String date, Object ctx )  11 throws RulesException, IllegalArgumentException;  12 void execute( String rule, ItemProvider itemProvider, ResultConsumer consumer,  13 int codetype, String date, Object ctx, Trace trace, Map callctx )  14 throws RulesException, IllegalArgumentException;  15 void close();  16 boolean isClosed();  17 boolean available();  18 …  19 } |

executeメソッドは、入出力値の引き渡し方式によって2種類に分類できる。最初と2番目のメソッドは、RuleReqオブジェクトをパラメータとして受け取りResultSetオブジェクトを返す\*1。3番目と4番目のメソッドは、ItemProviderとResultConsumerオブジェクトをパラメータとして受け取る。前者をI/Oオブジェクト(I/O Object)方式といい、後者をI/Oアダプタ(I/O Adapter)方式という。各方式については「1.2.6 I/Oオブジェクト方式」と「1.2.7 I/Oアダプタ方式」で説明し、「1.2.8 ルールエンジンの構成方式に基づくI/Oオブジェクト方式とI/Oアダプタ方式の性能比較」で各方式のメリットとデメリットを説明する。

1. ResultSetの完全修飾名(Fully Qualified Name)は、com.innoexpert.rulesclient.ResultSetである。java.sql.ResultSetとは異なるインタフェースである。しかし、多くの部分がjava.sql.ResultSetメソッドを再定義したため、同様のメソッドが多く定義されており、使用方法も似ている。ただし、java.sql.ResultSetは使用後にcloseを呼び出す必要があるが、com.innoexpert.rulesclient.ResultSetはcloseメソッドがない。

次は、executeを除いた各メソッドに関する説明である。

void close()

RuleInterfaceをルールインタフェースクラスタに返す。

boolean isClosed()

RuleInterfaceがルールインタフェースクラスタに返された場合、TRUEを返す。その他の場合は、FALSEを返す。

boolean available()

RuleInterfaceが現在使用可能な状態なのか判断する。使用可能の場合TRUE、使用不可の場合FALSEを返す。

### コード体系

I/Oオブジェクト方式では、呼び出すルールコードと入力項目のコードおよび値がRuleReq入力オブジェクトに保存されて引き渡される。I/Oアダプタ方式では、呼び出すルールのコードがexecuteメソッドのパラメータとして引き渡される。コードとして使用できる値は、次のとおりである。

* ルールまたは項目のID
* ルールまたは項目の名前
* ルールまたは項目のエイリアス

IDは、ルールシステムが管理目的ですべてのルールと項目に付与するアルファベットと数字の10桁のコードである。ルールシステムで自動的に付与されたIDは、‘#S’で始まる。IDは固有のものが付与されるため、ルール同士または項目同士で重複することはない。IDは、一度付与されると変更できない。

名前は、ルール管理者がすべてのルールと項目に付与する127文字以内のコードである。いくつかの特殊文字を除き、ほとんどの文字が使用できる。ルールビルダーで、すべてのルールの内容は名前で表現されるため、大概は業務に合わせた名前が付与される。名前もIDと同様にルールシステム内では固有のものであるため、ルール同士または項目同士で重複しないが、変更は可能である。

IDはルールシステムから識別および管理のために付与され、名前はルール管理者が業務的な意味を込めて付与する。一般的にITシステムでは様々な制約条件によって業務的な意味を持つ名前をそのまま使用せず、制限された文字を使う。例えば、アルファベットの大文字と数字、‘\_’で構成された別のコード体系を運用する。エイリアスは、ルールまたは項目に、このコード体系に基づくコードを付与できるようにする。ルールエイリアスは、IDまたは名前とは異なり、すべてのルールと項目に付与する必要はない。エイリアスは必要なルールと項目に任意で付与でき、付与した後でも変更できる。ただし、付与された場合はルール同士または項目同士で重複できない。

ルールサービスを呼び出す時にどのルールを呼び出すか、どの項目の値を設定するかを指定するために、ID、名前、エイリアスの中から1つのコード体系を選択できる。ただし、1つのルールの呼び出しには1つのコード体系のみ使用できる。例えば、ルールを指定するためにIDを使用する場合は、項目もIDを利用して指定する必要がある。この文書の以降の部分においてのコードとは、ID、名前またはエイリアスのうちの1つを意味する。

ルールおよび項目のコード指定にどのコード体系が使用されたかは、[表 6 com.innoexpert.rulesclient.RuleInterface]のRuleInterface.executeメソッドのcodetypeパラメータに指定する。com.innoexpert.rulesclient.Constantsインタフェースには、次の表のようにcodetypeの値として指定できる定数が定義されている。

[表 7 コード体系を指定する定数値]

|  |
| --- |
| 1 package com.innoexpert.rulesclient;  2  3 public interface Constants  4 {  5 static final int CODETYPE\_ID = 0x00;  6 static final int CODETYPE\_NAME = 0x01;  7 static final int CODETYPE\_ALIAS = 0x02;  8 …  9 } |

CODETYPE\_IDは使用されたコード体系がIDであることを意味し、CODETYPE\_NAMEは名前を、CODETYPE

\_ALIASはエイリアスを意味する。

### データ型と関連定数

ルールエンジンは、3つの基本データ型をサポートする。これを表現するJavaデータ型は、次のとおりである。

* 数値型:double、java.lang.Double、double[]、java.math.BigDecimalまたはjava.math.BigDecimal[]
* 文字型:java.lang.Stringまたはjava.lang.String[]
* 論理型:boolean、java.lang.Booleanまたはboolean[]

項目の形式やルール実行結果カラムの形式は、上記のデータ型のうちの1つに該当する。項目またはルール結果カラムがどの形式なのか識別するのに使用される定数が、次の表のようにcom.innoexpert.rulesclient.Constantsインタフェースに定義されている。

[表 8 データ型定数]

|  |
| --- |
| 1 package com.innoexpert.rulesclient;  2  3 public interface Constants  4 {  5 static final int TYPE\_NONE = 0x00;  6 static final int TYPE\_NUMBER = 0x01;  7 static final int TYPE\_STRING = 0x02;  8 static final int TYPE\_BOOLEAN = 0x03;  9 …  10 } |

TYPE\_NONEは、データ型が指定されていないことを意味する。TYPE\_NUMBERは数値型を、TYPE\_STRINGとTYPE\_BOOLEANはそれぞれ文字型と論理型を意味する。

### 数値型モード

ユーザはルールエンジンで使用する数値型をJavaのdouble型、またはBigDecimal型から選んで利用することができる。ユーザが使用できる数値型のモードは次の表のようにcom.innoexpert.rulesclient.NumericMode列挙型(enum)に定義している。

[表 9 数値型モード]

|  |
| --- |
| 1 package com.innoexpert.rulesclient;  2  3 public enum NumericMode  4 {  5 IEEE754,  6 DECIMAL  7 } |

数値型モードの意味は次のとおりである。

* IEEE754：倍精度浮動小数点数、10進数で15桁の精度で表現可能
* DECIMAL：BigDecimalモード、10進数で16桁の精度で表現可能

I/Oオブジェクト方式を使用する場合RuleReqオブジェクトを生成する時点で数値型モードを指定して使用できる。ルールエンジンはユーザが数値型モードを選択しなかった場合、IEEE754モードで動作する。I/Oアダプタ方式を使用する場合は構成方式により数値型モードが決まる。

### I/Oオブジェクト方式

I/Oオブジェクト方式は、呼び出すルールのコードや入力項目の値を1つの入力オブジェクトにすべて格納してルールを呼び出し、その結果を1つのオブジェクトに格納して受け取るルール呼び出し方式である。次の図は、I/Oオブジェクト方式のルールサービスの呼び出しを示す。



[図 1 I/Oオブジェクト方式のルールサービス呼び出し]

ルールコードとルール実行に必要な項目値は、RuleReqという入力オブジェクト(Input Object)に格納してルールエンジンに引き渡される。ルールエンジンはそのオブジェクトに入力された情報のみを利用してルールを実行し、その結果をResultSetという出力オブジェクト(Output Object)に格納して返す。ルールエンジンがルール実行中にRuleReqに設定された項目以外の項目値が必要な場合は、アプリケーションにエラーを返す。

次の表は、RuleInterfaceに定義されたI/Oオブジェクト方式のexecuteメソッドである。これらのメソッドは、すべてパラメータとしてRuleReqオブジェクトを受け取り、ResultSetを返す。executeメソッドを呼び出す際、コード体系を指定するcodetypeに「1.2.3 コード体系」以外の値が指定されるとIllegalArgumentExceptionを発生する。3番目のctxオブジェクトはコールバックコンテキストオブジェクトとしてコールバックハンドラーのサービスメソッドにそのまま渡される。コールバックについての詳細内容は「1.3 コールバックAPI」を参照する。

[表 10 I/Oオブジェクト方式のexecuteメソッド]

|  |
| --- |
| 1 package com.innoexpert.rulesclient;  2  3 public interface RuleInterface  4 {  5 ResultSet execute( RuleReq req, int codetype )  6 throws RulesException, IllegalArgumentException;  7 ResultSet execute( RuleReq req, int codetype, Object ctx )  8 throws RulesException, IllegalArgumentException;  9 …  10 } |

#### com.innoexpert.rulesclient.RuleReq

com.innoexpert.rulesclient.RuleReqクラスは、I/Oオブジェクト方式で入力オブジェクトとして使用されるクラスである。このクラスには呼び出すルールのコードおよび入力項目値を設定し、設定された値を照会するメソッドの他にも、ルール実行に使用される付加情報を設定するメソッドが定義されている。RuleReqクラスには、次の表のメソッドが定義されている。

[表 11 com.innoexpert.rulesclient.RuleReq]

|  |
| --- |
| 1 package com.innoexpert.rulesclient;  2  3 import java.util.\*;  4  5 public class RuleReq  6 {  7 public RuleReq();  8 public RuleReq( NumericMode nm );  9  10 public void setRuleCode( String ruleCode );  11 public String getRuleCode();  12  13 public void setDate( String date );  14 public String getDate();  15  16 public NumericMode getNumericMode();  17  18 public Item addNumberItem( String code );  19 public Item addStringItem( String code );  20 public List<Item> getItems();  21 public void resetItems();  22  23 public void setWantTrace( boolean wantTrace );  24 public boolean getWantTrace();  25  26 public void setContext( Map context );  27 public Map getContext();  28  29 public ItemProvider createItemProvider();  30  31 } |

次は、各メソッドに関する説明である。

public RuleReq()

数値型モードでIEEE754を指定し、RuleReqオブジェクトを生成する。

public RuleReq( NumericMode nm )

数値型モードでnmパラメータを指定し、RuleReqオブジェクトを生成する。nmパラメータはNumericModeの値である。(詳細は「1.2.5 数値型モード」を参照)

void setRuleCode( String ruleCode )

呼び出すルールのコードを設定する。ruleCodeパラメータはルールのコードであり、このコードは

RuleInterface.executeに引き渡されるコード体系のコードとなる。例えば、executeメソッドにコード体系としてConstants.CODETYPE\_IDが指定された場合、ruleCodeパラメータは呼び出すルールのIDとなる。

String getRuleCode()

RuleReqオブジェクトに設定されたルールコードを返す。

void setDate( String date )

ルール実行の基準日付を指定する。1つのルールには、日付別に異なる規則が定義できる。これをルールのバージョンという。ルールを呼び出す時に基準日を指定すると、ルールエンジンはその日付のバージョンを検索して実行する。基準日付が指定されない場合、ルールエンジンのシステム日付を使用する。dateパラメータは‘yyyy-MM-dd’または‘yyyyMMdd’形式の基準日である。nullに設定される場合、基準日付を指定しない。

String getDate()

RuleReqオブジェクトに設定された基準日を返す。設定されていない場合は、nullを返す。

public NumericMode getNumericMode()

RuleReqオブジェクトに設定された数値型モードを返却する。

public Item addNumberItem( String code )

RuleReqオブジェクトに数値型Itemオブジェクトを追加する。codeパラメータは項目のコードであり、このコードはRuleInterface.executeに渡されるコード体系のコードである必要がある。例えば、executeメソッドにコード体系としてConstants.CODETYPE\_IDが指定された場合、codeパラメータは追加する項目のIDである必要がある。このメソッドはItemオブジェクト自身を返す。

public Item addStringItem( String code )

RuleReqオブジェクトに文字型Itemオブジェクトを追加する。codeパラメータは項目のコードであり、このコードはRuleInterface.executeに渡されるコード体系のコードである必要がある。例えば、executeメソッドにコード体系としてConstants.CODETYPE\_IDが指定された場合、codeパラメータは追加する項目のIDである必要がある。このメソッドはItemオブジェクト自身を返す。

List<Item> getItems()

RuleReqオブジェクトに設定されている項目、つまりItemのjava.util.Listオブジェクトを返す。

この時、返されたjava.util.ListオブジェクトはCollections.unmodifiableList()を通じて返され、Itemの追加または削除が不可である。

void resetItems()

RuleReqオブジェクトに設定されている項目をリセットする。設定された項目情報は、すべて削除される。ただし、ルールコードや基準日のような項目以外の情報はそのまま保持される。

void setWantTrace( boolean wantTrace )

ルールを実行する際、実行経路トレースの作成可否について設定する。TRUEに設定すると、ルールエンジンは実行経路トレースを作成し、これを出力オブジェクトに格納して返す。FALSEに設定すると、実行経路トレースを作成しない。実行経路トレースを作成する場合、ルール実行性能がわずかに低下し、出力オブジェクトの大きさがわずかに増加することがある。そのため、実行経路トレースは問題解決の目的のみで使用を推奨する。

boolean getWantTrace()

実行経路トレース作成オプションの設定状況について返す。設定されていればTRUEを、そうでなければFALSEを返す。

void setContext( Map context )

ルールを実行する時、必要な実行情報を格納したcontextを設定する。このオブジェクトは、ルール実行時、使用されるルール実行結果キャッシュまたはルール実行経路トレースの可否などルールインタフェースを実装したクラスから使用可能な情報を格納して使用できる。

Map getContext()

ルールを実行する時、使用可能なcontextオブジェクトを返す。

ItemProvider createItemProvider()

I/Oオブジェクト方式のオブジェクトを、I/Oアダプタ方式のオブジェクトに変換して返す。

#### com.innoexpert.rulesclient.Item

com.innoexpert.rulesclient.Itemインタフェースは、ルール呼び出しの入力で使用される項目のコードと値を保存するためのインタフェースである。アプリケーションには項目値のタイプによって次のクラスを使用する。

* 数値型:com.innoexpert.rulesclient.NumberItem
* 文字型:com.innoexpert.rulesclient.StringItem

Itemインタフェースには、次の表のメソッドが定義されている。

[表 12 com.innoexpert.rulesclient.Item]

|  |
| --- |
| 1 package com.innoexpert.rulesclient;  2  3 import java.math.BigDecimal;  4  5 public interface Item  6 {  7 String getCode();  8 byte getType();  9 int length();  10  11 Item add( double value ) throws UnsupportedOperationException;  12 Item add( String value ) throws UnsupportedOperationException;  13 Item add( BigDecimal value ) throws UnsupportedOperationException;  14  15 double getNumber( int index ) throws UnsupportedOperationException, IndexOutOfBoundsException;  16 String getString( int index ) throws UnsupportedOperationException, IndexOutOfBoundsException;  17 BigDecimal getDecimal( int index ) throws UnsupportedOperationException, IndexOutOfBoundsException;  18  19 NumericMode getNumericMode() throws UnsupportedOperationException;  20 } |

次は、各メソッドに関する説明である。

String getCode()

Itemオブジェクトの項目コードを返す。

byte getType()

Itemオブジェクトのデータ型を返す。返されるデータ型コードについては、「1.2.4 データ型と関連定数」を参照のこと。

int length()

Itemオブジェクトの項目値の配列の長さを返す。

Item add( double value ) throws UnsupportedOperationException

Itemの配列に倍精度浮動小数点数値を追加する。倍精度浮動小数点数値を使用できないStringItemのオブジェクトの場合、UnsupportedOperationExceptionが投げられる。このメソッドはオブジェクト自身を返す。

Item add( String value ) throws UnsupportedOperationException

Itemの配列に文字値を追加する。文字値が使用できないNumberItemオブジェクトの場合、UnsupportedOperationExceptionが投げられる。このメソッドはオブジェクト自身を返す。

Item add( BigDecimal value ) throws UnsupportedOperationException

Itemの配列にBigDecimal値を追加する。BigDecimal値を使用できないオブジェクトの場合、UnsupportedOperationExceptionが投げられる。このメソッドはオブジェクト自身を返す。

double getNumber( int index ) throws UnsupportedOperationException, IndexOutOfBoundsException

Item値の配列のindex番目の倍精度浮動小数点数値を返す。indexは、0ベース(0-base)のインデックスである。配列の長さがindexより小さい場合、java.lang.IndexOutOfBoundsExceptionが投げられる。数値を使用できないオブジェクトの場合、UnsupportedOperationExceptionが投げられる。

String getString( int index ) throws UnsupportedOperationException, IndexOutOfBoundsException

Item値の配列のindex番目の文字値を返す。indexは、0ベース(0-base)のインデックスである。配列の長さがindexより小さい場合、java.lang.IndexOutOfBoundsExceptionが投げられる。文字値は使用できないオブジェクトの場合は、UnsupportedOperationExceptionが投げられる。

BigDecimal getDecimal( int index ) throws UnsupportedOperationException, IndexOutOfBoundsException

Item値の配列のindex番目のBigDecimal値を返す。indexは、0ベース(0-base)のインデックスである。配列の長さがindexより小さい場合、java.lang.IndexOutOfBoundsExceptionが投げられる。BigDecimal値は使用できないオブジェクトの場合は、UnsupportedOperationExceptionが投げられる。

NumericMode getNumericMode() throws UnsupportedOperationException

Itemオブジェクトの数値型モードを返す。数値型を使用できないオブジェクトの場合は、UnsupportedOperationExceptionが投げられる。

Item.addメソッドは自身の参照値を返すため、addメソッドを連結するコーディングパターンが使用できる。[表 14 簡略化されたNumberItemの例]は、そのようなパターンを使用して[表 13 NumberItemの例]のコードを修正したものであり、両コードは同様に動作する。

[表 13 NumberItemの例]

|  |
| --- |
| 1 NumberItem item = new NumberItem( "ITEMIDHERE" );  2 item.add( 1.0 );  3 item.add( 2.0 );  4 item.add( 3.0 ); |

[表 14 簡略化されたNumberItemの例]

|  |
| --- |
| 1 NumberItem item = new NumberItem( "ITEMIDHERE" );  2 item.add( 1.0 ).add( 2.0 ).add( 3.0 ); |

#### com.innoexpert.rulesclient.ResultSet

ルール呼び出し結果データをテーブル形式で表現するインタフェースである。インタフェースの名前のとおり、java.sql.ResultSetとメソッドおよび使用方法が類似している。ResultSetは、データの中で現在の行を示すカーソルを持っている。RuleInterface.executeメソッドから返された直後のこのカーソルは、最初の行のすぐ前を示す。nextメソッドはカーソルを1行ずつ進め、それ以上進めない時、つまりデータの最後ではFALSEを返す。したがって、java.sql.ResultSetのようにwhile文の条件式にnextメソッドを使用できる。

このインタフェースは、java.sql.ResultSetと異なりcloseメソッドが存在しない。データベースクエリは結果の一部のみアプリケーションに引き渡され、引き渡された結果をすべて照会すると残りの結果をサーバからインポートする方式だが、ルールサービスの実行終了時、すべての結果が一回でアプリケーションに引き渡されるため、サーバ側に占有されているリソースがなく、closeを呼び出す必要がない。

ResultSetの各カラムには、インデックスと名前が付与される。特定カラム値を取得するために、カラムインデックスまたはカラム名が使用できる。カラムインデックスは1ベース(1-base)で、最初のカラムのインデックスは1である。カラム名は、RuleInterface.executeを呼び出した時に指定されたコード体系によって異なる。コード体系をIDまたは名前で指定した場合、ResultSetのカラム名は呼び出したルールのリターン項目名に対応する。最初のカラム名は呼び出したルールの最初のリターン項目名で設定され、2番目以降のカラムも同一方式が適用される。コード体系をエイリアスで指定した場合、カラム名は呼び出したルールのリターン項目のエイリアスに対応される。ルールリターン項目のうち、一部のみにエイリアスが付与された場合は、エイリアスが付与されていないカラム値は名前を利用して取得できない。もちろん、カラムインデックスを利用して取得することはできる。

ResultSet インタフェースには、次の表のメソッドが定義されている。

[表 15 com.innoexpert.rulesclient.ResultSet]

|  |
| --- |
| 1 package com.innoexpert.rulesclient;  2  3 import java.math.BigDecimal;  4  5 public interface ResultSet  6 {  7 boolean next();  8 boolean previous();  9 String getString( int columnIndex ) throws IndexOutOfBoundsException;  10 boolean getBoolean( int columnIndex ) throws IndexOutOfBoundsException;  11 double getDouble( int columnIndex )  12 throws IndexOutOfBoundsException, NumberFormatException;  13 BigDecimal getBigDecimal( int columnIndex ) 14 throws IndexOutOfBoundsException, NumberFormatException;  15 short getShort( int columnIndex )  16 throws IndexOutOfBoundsException, NumberFormatException;  17 int getInteger( int columnIndex )  18 throws IndexOutOfBoundsException, NumberFormatException;  19 String getString( String columnName ) throws IndexOutOfBoundsException;  20 boolean getBoolean( String columnName ) throws IndexOutOfBoundsException;  21 double getDouble( String columnName )  22 throws IndexOutOfBoundsException, NumberFormatException;  23 BigDecimal getBigDecimal( String columnName ) 24 throws IndexOutOfBoundsException, NumberFormatException;  25 short getShort( String columnName )  26 throws IndexOutOfBoundsException, NumberFormatException;  27 int getInteger( String columnName )  28 throws IndexOutOfBoundsException, NumberFormatException;  29 ResultSetMetaData getMetaData();  30 int findColumn( String columnName );  31 void beforeFirst();  32 void afterLast();  33 boolean isFirst();  34 boolean isLast();  35 boolean isBeforeFirst();  36 boolean isAfterLast();  37 boolean first();  38 boolean last();  39 int getRow();  40 int getRowCount();  41 boolean absolute( int row );  42 boolean relative( int rows );  43 } |

次は、各メソッドに関する説明である。

boolean next()

カーソルを次の行に移動させる。次の行に移動できない時、つまり現在のカーソルの位置が最後の行の次にある場合は、FALSEを返す。そうでない場合は、TRUEを返す。

boolean previous()

カーソルを前の行に移動させる。前の行に移動できない時、つまり現在のカーソルの位置が最初の行の前にある場合は、FALSEを返す。そうでない場合は、TRUEを返す。

String getString( int columnIndex ) throws IndexOutOfBoundsException

現在の行のcolumnIndex番目のカラム値をjava.lang.String形式で返す。このカラムのデータ型が文字型でない場合は、Stringに変換された値を返す。Stringに変換する際に特定の形式に変換する場合は、カラムのデータ型に合致するようにgetDoubleまたはgetBooleanを呼び出してフォーマットする必要がある。columnIndexは1ベースインデックスで、最初のカラムのインデックスは1である。columnIndexが0以下またはカラムの大きさを超える場合は、IndexOutOfBoundsExceptionが投げられる。

boolean getBoolean( int columnIndex ) throws IndexOutOfBoundsException

現在の行のcolumnIndex番目のカラム値をboolean形式で返す。このカラムのデータ型が論理型でない場合は、適切な変換を行う。カラムのデータ型が文字型である場合は、Boolean.valueOf()メソッドを利用してbooleanに変換し、数値型である場合、0ならFALSE、そうでない場合TRUEを返す。columnIndexは1ベースインデックスで、最初のカラムのインデックスは1である。columnIndexが0以下またはカラムの大きさを超える場合は、IndexOutOfBoundsExceptionが投げられる。

double getDouble( int columnIndex ) throws IndexOutOfBoundsException, NumberFormatException

BigDecimal getBigDecimal( int columnIndex ) throws IndexOutOfBoundsException, NumberFormatException

short getShort( int columnIndex ) throws IndexOutOfBoundsException, NumberFormatException

int getInteger( int columnIndex ) throws IndexOutOfBoundsException, NumberFormatException

現在の行のcolumnIndex番目のカラム値をそれぞれdouble、BigDecimal、short、int形式で返す。このカラムのデータ型が数値型の場合は、デフォルト値はdouble形式の値であり、getDouble、getBigDecimal、getShort、getIntegerはそれぞれDouble、BigDecimal、short、intに型変換して返す。カラムのデータ型が文字型の場合は、Double.parseDouble()、new BigDecimal()、Short.parseShort()、Integer.parseInt()を呼び出して型変換をするが、文字値が正しい数字表現でない場合はNumberFormatExceptionが投げられる。カラムのデータ型が論理型の場合、その値がTRUEである場合は1、そうでない場合は0に変換される。columnIndexは1ベースインデックスで、最初のカラムのインデックスは1である。columnIndexが0以下またはカラムの大きさを超える場合は、IndexOutOfBoundsExceptionが投げられる。

String getString( String columnName ) throws IndexOutOfBoundsException

カラム名がcolumnNameであるカラムの値をjava.lang.String形式で返す。columnNameは、ルールサービスの呼び出しに使用されたコード体系がIDまたは名前である場合はルールのリターン項目の名前であり、コード体系がエイリアスである場合はルールのリターン項目のエイリアスである。columnNameに該当するカラムが存在しない場合は、IndexOutOfBoundsExceptionが投げられる。カラムのデータ型が文字型でない場合は、適切な変換を行う。これについては、getString(int)を参照のこと。

boolean getBoolean( String columnName ) throws IndexOutOfBoundsException

カラム名がcolumnNameであるカラムの値をboolean形式で返す。columnNameは、ルールサービスの呼び出しに使用されたコード体系がIDまたは名前である場合はルールのリターン項目の名前であり、コード体系がエイリアスである場合はルールのリターン項目のエイリアスである。columnNameに該当するカラムが存在しない場合は、IndexOutOfBoundsExceptionが投げられる。カラムのデータ型が論理型でない場合は、適切な変換をする。これについては、getBoolean(int)を参照のこと。

double getDouble( String columnName ) throws IndexOutOfBoundsException, NumberFormatException

BigDecimal getBigDecimal( String columnName ) throws IndexOutOfBoundsException, NumberFormatException

short getShort( String columnName ) throws IndexOutOfBoundsException, NumberFormatException

int getInteger( String columnName ) throws IndexOutOfBoundsException, NumberFormatException

カラム名がcolumnNameであるカラムの値をそれぞれdouble、BigDecimal、short、int形式で返す。columnNameは、ルールサービスの呼び出しに使用されたコード体系がIDまたは名前の場合はルールのリターン項目の名前であり、コード体系がエイリアスの場合はルールのリターン項目のエイリアスである。columnNameに該当するカラムが存在しない場合は、IndexOutOfBoundsExceptionが投げられる。カラムのデータ型が数値型でない場合は、適切な変換を行う。これについては、getDouble(int)、getBigDecimal()、getShort(int)、getInteger(int)をそれぞれ参照のこと。

ResultSetMetaData getMetaData()

ResulSetのカラム情報を持っているResultSetMetaDataオブジェクトを返す。ResultSetMetaDataは、ルール実行結果のMetaDataを持っている。このResultSetMetaDataについては、「1.2.6.4 com.innoexpert.rulesclient.ResultSetMetaData」を参照のこと。

int findColumn( String columnName )

カラム名がColumnNameであるカラムのindexを返す。

void beforeFirst()

ResultSetのカーソルを最初の行の前に移動させる。このメソッドを呼び出してgetString、getBoolean、getDouble、getShort、getIntegerのようなメソッドを呼び出す場合は、nullまたはエラーを返す。このメソッドを呼び出した後は、next()メソッドを呼び出して結果値を参照すること。

void afterLast()

ResultSetのカーソルを最後の行の次に移動させる。このメソッドを呼び出してgetString、getBoolean、getDouble、getShort、getIntegerのようなメソッドを呼び出す場合は、nullまたはエラーを返す。このメソッドを呼び出した後は、previous()メソッドを呼び出して結果値を参照すること。

boolean isFirst()

ResultSetのカーソルが最初の行の場合は、TRUEを返す。そうでない場合は、FALSEを返す。

boolean isLast()

ResultSetのカーソルが最後の行の場合は、TRUEを返す。そうでない場合は、FALSEを返す。

boolean isBeforeFirst()

ResultSetのカーソルが最初の行の前にある場合は、TRUEを返す。そうでない場合は、FALSEを返す。

boolean isAfterLast()

ResultSetのカーソルが最後の行の次にある場合は、TRUEを返す。そうでない場合は、FALSEを返す。

boolean first()

ResultSet のカーソルを最初の行に移動させる。正常に移動した場合は、TRUEを返す。ResultSetの結果、行が存在しない場合はFALSEを返す。

boolean last()

ResultSetのカーソルを最後の行に移動させる。正常に移動した場合は、TRUEを返す。ResultSetの結果、行が存在しない場合はFALSEを返す。

int getRow()

ResultSetの現在の行のindexを返す。

int getRowCount()

結果データの全体の行数を返す。

boolean absolute( int row )

ResultSetのカーソルを指定されたRowに移動させる。入力された値が正の数の場合は、最初のRowの前から指定された値の分のみ移動させる。つまり、1が入力された場合はカーソルを最初のRowに移動させる。入力された値が負の数の場合は、最後のRowの次から指定された値の分のみカーソルを戻す。つまり、-1が入力された場合はカーソルを最後のRowに移動させる。0が入力された場合は、IllegalArgumentExceptionが投げられる。入力された値を最初のRowから最後のRowの範囲内で移動させる場合はTRUEを返す。入力された値が最初のRowから最後のRowの範囲外にある場合はFALSEを返し、入力された値が正の数である場合は、カーソルを最後のRowの次に移動させ、負の数である場合は最初のRowの前に移動させる。ResultSetの結果、行が存在しない場合はFALSEを返す。

boolean relative( int rows )

ResultSetのカーソルを現在の位置から指定されたrowsの分のみ移動させる。入力された値が正の数の場合は、現在の位置から指定された値の分のみカーソルを先に進める。つまり、1が入力された場合はnext()と同一である。入力された値が負の数の場合は、指定された値の分のみカーソルを戻す。つまり、-1が入力された場合はprevious()と同一の動作となる。0が入力された場合は、現在の位置から移動しない。移動させる位置が最初のRowから最後のRowの範囲内である場合はTRUEを返し、移動させる位置が最初のRowから最後のRowの範囲外である場合はFALSEを返す。カーソルを移動させる位置が正の数の場合は、最後のRowの次に移動させ、負の数の場合は最初のRowの前に移動させる。ResultSetの結果、行が存在しない場合はFALSEを返す。

#### com.innoexpert.rulesclient.ResultSetMetaData

ResultSetMetaDataインタフェースはResultSetのカラム情報を提供するインタフェースで、カラム数、名前、データ型などを提供する。各カラムはルールのリターン項目に対応するが、カラム名はルールサービスの呼び出しに使用されたコード体系によってはルールのリターン項目名またはエイリアスになることがある。カラム名とルールのリターン項目との関係については、「1.2.6.3 com.innoexpert.rulesclient.ResultSet」を参照のこと。

ResultSetMetaDataには、次の表のメソッドが定義されている。

[表 16 com.innoexpert.rulesclient.ResultSetMetaData]

|  |
| --- |
| 1 package com.innoexpert.rulesclient;  2  3 public interface ResultSetMetaData  4 {  5 int getColumnCount();  6 String getColumnName( int columnIndex ) throws IndexOutOfBoundsException;  7 short getColumnType( int columnIndex ) throws IndexOutOfBoundsException;  8 List<String> getTraceList();  9 } |

次は、各メソッドに関する説明である。

int getColumnCount()

カラム数を返す。

String getColumnName( int columnIndex ) throws IndexOutOfBoundsException

columnIndex番目のカラム名を返す。columnIndexは1ベースインデックスで、最初のカラムのインデックスは1である。columnIndexが0以下またはカラム数より大きい場合は、IndexOutOfBoundsExceptionが投げられる。

short getColumnType( int columnIndex ) throws IndexOutOfBoundsException

カラムのデータ型コードを返す。データ型コードについては、「1.2.4 データ型と関連定数」を参照する。columnIndexは1ベースのインデックスであり、columnIndexが0以下またはカラム数より大きい場合IndexOutOfBoundsExceptionが投げられる。

List<String> getTraceList()

List形式のルール実行経路トレース結果を返す。

#### ルールサービス呼び出し例

次の表は、手数料を計算するルールサービスを呼び出すWebページのソースコードのサンプルである。

[表 17 ルールサービス呼び出し例]

|  |
| --- |
| 1 <%@page contentType="text/html;charset=utf8"%>  2 <%@page import="com.innoexpert.rulesclient.\*"%>  3 <%  4 String tradeType = request.getParameter( "TRDTYP" );  5 String amount = request.getParameter( "AMT" );  6 String message = null;  7  8 if( tradeType != null && tradeType.length() > 0 && amount != null && amount.length() > 0 )  9 {  10 RuleReq req = new RuleReq( NumericMode.IEEE754 );  11 req.setRuleCode( "COMMISSION" );  12 req.setDate( "2013-07-01" );  13  14 req.addStringItem( "TRDTYP" ).add( tradeType );  15 req.addNumberItem( "AMT" ).add( Double.parseDouble( amount ) );  16  17 ResultSet rs = null;  18 RuleInterface intf = ClusterManager.getInterface();  19  20 try  21 {  22 rs = intf.execute( req, Constants.CODETYPE\_ALIAS );  23 }  24 catch( RulesException e )  25 {  26 message = "ERROR: " + e.getMessage();  27 }  28 finally  29 {  30 intf.close();  31 }  32  33 if( rs != null && rs.next() )  34 {  35 String rate = rs.getString( "CMSRATE" );  36 String commission = rs.getString( "CMSAMT" );  37 message = "Commission is " + commission + ". Rate is " + rate + ".";  38 }  39 }  40 else  41 tradeType = amount = "";  42 %>  43  44 <html>  45 <body>  46 <form id="commission-form" method="post">  47 <li>  48 <label for="TRDTYP">Trade Type:</label>  49 <input type="text" name="TRDTYP" id="TRDTYP" value="<%=tradeType%>"/>  50 </li>  51 <li>  52 <label for="AMT">Amount:</label>  53 <input type="text" name="AMT" id="AMT" value="<%=amount%>"/>  54 </li>  55 <li>  56 <input type="submit" value="submit"/>  57 </li>  58 </form>  59 <%=message != null ? message : "&nbsp"%>  60 </body>  61 </html> |

[44~61行目]は、ユーザから入力を受け取るためのHTMLコードである。HTMLは、取引の類型と取引金額が入力され再びこのページを呼び出すフォームを構成する。[4~8行目]は、フォームからの入力値を読み込んで有効性を検証するコードである。[10~38行目]はルールサービスを呼び出すコードで、フォームから入力された値を利用して手数料計算ルールを呼び出して結果メッセージを作成する。

[10~12行目]ではRuleReqオブジェクトを生成して呼び出すルールのエイリアスである“COMMISSION”を設定し、ルール実行の基準日を“2013-07-01”に設定している。基準日にシステム日付を使用する場合は、[12行目]の引数にnullを設定するか、12行自体を削除する。[14~15行目]では、入力項目の値を設定している。このルールは2つの項目を必要とし、そのエイリアスはそれぞれTRDTYPとAMTである。[18行目]ではルールインタフェースクラスタでマネージャからルールインタフェースを取得しており、その結果を利用して[22行目]でルールサービスを呼び出し、[30行目]でクラスタに返している。try-finallyブロックを利用することで、ルール呼び出し中に例外が投げられた場合でもルールインタフェースがクラスタに返されるよう保障する。[24~27行目]では、ルール呼び出し中に投げられるエラーを処理している。呼び出すルールと項目の値を指定する際、コード体系としてエイリアスを使用しているため、[22行目]のexecuteメソッドのパラメータとしてConstants.CODETYPE\_ALIASが引き渡されている。

[33~38行目]では、ルール呼び出しの結果を抽出してメッセージを作成している。手数料計算ルールは単一ルールであるため、結果の行数は常に1行で、rs.next()がif文の条件式に使用されているが、呼び出したルールが多重ルールである場合はrs.next()をwhile文の条件式に使用することもある。[35行目]と[36行目]では、ルールの結果から手数料率と手数料金額を抽出している。ルール呼び出しに使用されたコード体系がエイリアスであるため、ここでもルールリターンのエイリアスが使用されている。

例では、次の順序でルールサービスが呼び出されている。

1. RuleReqオブジェクト設定
2. ルールインタフェースの取得
3. ルールの呼び出し
4. ルールインタフェースの返却
5. ルール結果の使用

ルールインタフェースは、ルールインタフェースクラスタのプールで管理される。あるアプリケーションがインタフェースを使用中の場合、別のアプリケーションはそのインタフェースを使用できない。そのため、ルールインタフェースの占有時間を短く設定することを推奨する。アプリケーションが必要以上にルールインタフェースを参照している場合、別のアプリケーションがインタフェースの取得のため待機状況となる可能性もある。これはアプリケーションの性能を低下させる要因となるため、ルールインタフェースを使用しないコード、つまりRuleReqオブジェクトの設定やルール結果の使用などは、ルールインタフェースを取得する前やルールインタフェースを返した後に実行することを推奨する。

ただし、アプリケーションの特性によっては、異なることもあり、バッチアプリケーションのように他のアプリケーションとルールインタフェースを共有して使用しない環境では、毎回ルールを呼び出すたびにルールインタフェースを取得し、返す必要はない。バッチアプリケーションのような場合は、アプリケーションが初期化される時にルールインタフェースを取得し、終了する時に返す方がより良い結果となることもある。

### I/Oアダプタ方式

ルールサービスの入力として使用されるデータは、様々な場所に位置している場合がある。あるデータは値オブジェクト(Value Object)やマップのようなビジネスアプリケーションのデータオブジェクトに存在している場合や、あるデータがデータベーステーブル上に存在している場合もある。さらに、あるデータが値オブジェクトまたはデータベーステーブルに存在するデータを加工して作成されている場合もある。

ルールには、そのルールを実行するために必要な項目が定義される。これらの項目内のある項目は、ルールが実行されるたびに使用されることもあれば、ある項目は特定条件を満たす時のみ使用されることもある。ルールアプリケーションはルールの構成を認識できないため、ルールの入力として定義された項目のうち、どの項目が実際に使用される項目かを把握することができない。そのため、I/Oオブジェクト方式ではルールを呼び出す時にルールに定義されたすべての入力項目の値をRuleReqオブジェクトに格納する必要がある。もし、この入力項目の値がデータベースに存在する場合、その値の使用有無に問わず、アプリケーションはクエリを実行して値を読み込む必要がある。

ルールの入力項目が変更される場合にも、アプリケーションのメンテナンスに問題がある可能性がある。“age”および “gender”というフィールドを持つアプリケーションデータオブジェクトを想定し、あるルールで“age”のみを使用した判定を行う場合、I/Oオブジェクト方式を使用するアプリケーションは、データオブジェクトのフィールドのうち“age”のみをRuleReqに設定してルールを呼び出す。しかし、ルールの判定規則が変更され“gender”を追加で使用する場合、アプリケーションコードの修正が必要である。アプリケーションデータオブジェクトにはすでに“gender”というフィールドはあるが、RuleReqに設定しておらず、このフィールドの値を設定するようにアプリケーションコードを修正する必要がある。ルールの入力項目が変更される時にアプリケーションを修正しないためには、最初からRuleReqに“age”のみでなく“gender”も設定することであるが、実際の環境ではアプリケーションデータオブジェクトには数多くのフィールドがあるため、現実的には不可能である。もしアプリケーションがルールにどの項目値を設定すべきか考慮せずに、ルールエンジンが必要とする項目をアプリケーションデータオブジェクトから取得することで、こうした問題解決に効果がある。

I/Oアダプタ方式は、ルールの実行に必要な項目の値をルール呼び出し時に引き渡すのではなく、ルールエンジンがルールを実行している最中に必要な項目値をアプリケーションに問い合わせする方式である。この方式では、ルールを呼び出す際は呼び出すルールのコードと基準日のみを引き渡す。



[図 2 I/Oアダプタ方式のルールサービスの呼び出し]

[図 2 I/Oアダプタ方式のルールサービスの呼び出し]は、I/Oアダプタ方式でルールサービスを呼び出すプロセスを示す。1のプロセスで、アプリケーションはルールコードと基準日を引き渡してルールサービスを呼び出す。ルールエンジンはルールを実行し、実行中に必要な項目値がある場合、アプリケーション側にその項目の値を問い合わせする(2のプロセス)。アプリケーション側では、その項目の値を探してルールエンジンが送る(3のプロセス)。ルールエンジンはルールの残りの部分を実行し、追加の項目が必要な場合2と3のプロセスを繰り返す。ルールエンジンはルールの実行を終え、結果をアプリケーション側に設定する(4のプロセス)。

次の表は、[表 6 com.innoexpert.rulesclient.RuleInterface]のRuleInterfaceのメソッドのうちI/Oアダプタ方式のexecuteメソッドのリストである。executeメソッドを呼び出す場合、コード体系を指定するcodetypeに「1.2.3 コード体系」以外の値が指定されるとIllegalArgumentExceptionが発生する。

[表 18 I/Oアダプタ方式のexecuteメソッド]

|  |
| --- |
| 1 package com.innoexpert.rulesclient;  2  3 public interface RuleInterface  4 {  5 void execute( String rule, ItemProvider itemProvider, ResultConsumer consumer,  6 int codetype, String date, Object ctx )  7 throws RulesException, IllegalArgumentException;  8 void execute( String rule, ItemProvider itemProvider, ResultConsumer consumer,  9 int codetype, String date, Object ctx, Trace trace, Map callctx )  10 throws RulesException, IllegalArgumentException;  11 …  12 } |

#### com.innoexpert.rulesclient.NumericModeSelector

com.innoexpert.rulesclient.NumericModeSelectorインタフェースはI/Oアダプタ方式で使用されるcom.innoexpert.rulesclient.ItemProviderを実装したクラスに数値型モードを定義するためのインタフェースである。ユーザがItemProviderを実装したクラスにNumericModeSelectorインタフェースを実装しない場合、ルールエンジンは倍精度浮動小数点数モードで動作する。

次の表はNumericModeSelectorインタフェースに定義されたメソッドである。

[表 19 com.innoexpert.rulesclient.NumericModeSelector]

|  |
| --- |
| 1 package com.innoexpert.rulesclient;  2  3 public interface NumericModeSelector  4 {  5 NumericMode getNumericMode();  6 } |

次は、各メソッドに関する説明である。

NumericMode getNumericMode()

ルールエンジンがルールを実行する途中、数値型項目の値を処理する前にこのメソッドを呼び出す。たとえ数値型項目の値がNumericModeSelectorインタフェースを実装しないとルールエンジンは数値型項目の値を倍精度浮動小数点数モードで処理する。

#### com.innoexpert.rulesclient.ItemProvider

com.innoexpert.rulesclient.ItemProviderインタフェースは、ルールエンジンに項目値を提供するために定義されたインタフェースである。ItemProviderには、項目のデータ型により、その値を提供するメソッドが定義されている。アプリケーションは、次の図のように、アプリケーションデータオブジェクトへアクセス、またはデータベースにアクセスして項目値を提供するItemProviderオブジェクトを作成し、executeメソッドにパラメータとして引き渡す。ItemProviderは、業務データが保存されている場所または使用されるオブジェクトの形式などによって様々な形式で実装される。しかし、アプリケーションが使用するデータオブジェクトの形式が同一の場合、1つのItemProvider実装クラスを作成してすべてのアプリケーションが共有することもできる。ルールエンジンは、ルールの実行中に項目の値が必要な場合にItemProviderインタフェースを利用して値を問い合わせする。ローカルルールエンジン構成での問い合わせは、単に実装されたItemProviderオブジェクトのメソッドを呼び出す場合もあるが、TCP/IPリモートルールエンジン構成のようにTCP/IP通信を伴う複雑な作業になる場合もある。ローカルルールエンジン構成とリモートルールエンジン構成における性能比較は、「1.2.8 ルールエンジンの構成方式に基づくI/Oオブジェクト方式とI/Oアダプタ方式の性能比較」で詳しく扱う。単にJavaメソッドを呼び出す場合や、TCP/IP通信の場合においても、ItemProvider実装クラスのメソッドが呼び出される。



[図 3 ItemProviderダイアグラム]

次の表は、ItemProviderに定義されたメソッドである。

[表 20 com.innoexpert.rulesclient.ItemProvider]

|  |
| --- |
| 1 package com.innoexpert.rulesclient;  2  3 public interface ItemProvider  4 {  5 Object getAsNumber( String itemCode ) throws RulesException;  6 Object getAsString( String itemCode ) throws RulesException;  7 } |

次は、各メソッドに関する説明である。

Object getAsNumber( String itemCode ) throws RulesException

ルールエンジンがルール実行中に数値型の項目の値を問い合わせするために、このメソッドを呼び出す。項目のコードは、itemCodeパラメータに引き渡される。項目のコードは、RuleInterface.executeに引き渡されたコード体系に基づくコードである。例えば、executeメソッドにコード体系としてIDを指定した場合、itemCodeパラメータは項目のIDが引き渡される。

このメソッドでは、要求された項目の値をアプリケーションデータオブジェクトなどから抽出し返す必要がある。戻り値の形式は数値型モードの値により倍精度浮動小数点数の場合java.lang.Doubleまたはdouble[]のみが許可され、BigDecimalモードの場合はjava.math.BigDecimalまたはjava.math.BigDecimal[]のみが許可される。そうでない場合は、ルールエンジンは例外(エラーコード1011)を投げてルールの実行が終了する。アプリケーションデータオブジェクトなどに保存された項目値が指定された数値型モードに使用できない値ならこれらの形式に適切に変換を行う必要がある。変換ができない場合はRuleExceptionを投げることもある。 double[]またはjava.math.BigDecimal[]を返す場合は、配列の長さは1以上とする。そうでない場合は、ルールの実行が失敗して例外が投げられる可能性がある。(エラーコード1011)

アプリケーションデータオブジェクトなどでこの項目の値が見つからない場合は、nullを返すことがある。この時、ルールエンジンは項目値が入力されていないとみなし、例外を投げてルールの実行は終了する(エラーコード1000)。アプリケーションデータオブジェクトなどから項目値を取得するプロセスでエラーが発生し項目値を提供できない場合は、RulesExceptionを投げることがある。この場合も、ルールエンジンは例外を投げルールの実行は終了する(エラーコード1000)。

Object getAsString( String itemCode ) throws RulesException

ルールエンジンがルール実行中に文字型の項目の値を問い合わせするために、このメソッドを呼び出す。項目のコードは、itemCodeパラメータに引き渡される。項目のコードは、RuleInterface.executeに引き渡されたコード体系に基づくコードである。このメソッドでは、要求された項目の値をアプリケーションデータオブジェクトなどからjava.lang.Stringまたはjava.lang.String[]形式で返す。この形式以外のオブジェクトを返す場合、ルールエンジンは例外(エラーコード1011)を投げてルールの実行が終了する。アプリケーションデータオブジェクトなどに保存された項目値がjava.lang.Stringまたはjava.lang.String[]でない場合は、適切に変換する必要がある。変換できない場合は、RulesExceptionを投げることができる。java.lang.String[]を返す場合、配列の長さは1以上とし、配列内にnull値を許可しない。そうでない場合は、ルールの実行が失敗して例外が投げる可能性がある(エラーコード1011)。

アプリケーションデータオブジェクトなどからこの項目の値が見つからない場合はnullを返すことがあり、この場合のルールエンジンの動作方式はgetAsNumberメソッドと同一である。

java.util.Propertiesをデータオブジェクトとして使用するアプリケーションがあるとしたら、このアプリケーションは、次の表のようなデータをjava.util.Propertiesに保存する。プロパティのキーが項目のエイリアスと一致する場合を想定する。

[表 21 アプリケーションデータオブジェクト例 - Javaプロパティ]

|  |
| --- |
| TRDTYPE=Lease  AMT=250000000  PROPTYPE=APARTMENT |

次の表は、[表 21 アプリケーションデータオブジェクト例 - Javaプロパティ]のようなjava.util.Propertiesオブジェクトから項目値を提供するItemProviderの実装例である。

[表 22 ItemProvider の実装例 – PropertiesProvider]

|  |
| --- |
| 1 import java.util.\*;  2 import com.innoexpert.rulesclient.\*;  3  4 public class PropertiesProvider implements ItemProvider, NumericModeSelector  5 {  6 private final Properties props;  7  8 public PropertiesProvider( Properties props )  9 {  10 this.props = props;  11 }  12  13 public Object getAsNumber( String code ) throws RulesException  14 {  15 String value = props.getProperty( code );  16 try  17 {  18 return value != null ? Double.valueOf( value ) : null;  19 }  20 catch( NumberFormatException e )  21 {  22 throw new RulesException( e.getMessage(), e );  23 }  24 }  25  26 public Object getAsString( String code ) throws RulesException  27 {  28 return props.getProperty( code );  29 }  30  31 public NumericMode getNumericMode()  32 {  33 return NumericMode.IEEE754;  34 }  35 } |

このクラスは[4行目]でcom.innoexpert.rulesclient.ItemProviderインタフェースとNumericModeSelectorを実装しており、getAsNumber、getAsString、getNumericModeの3つのメソッドを実装している。コンストラクタは、java.util.Propertiesオブジェクトをパラメータとして受け取る。アプリケーションは自身が参照しているPropertiesオブジェクトをコンストラクタに引き渡し、これはPropertiesProviderのコンストラクタからメンバ変数として参照される。

getAsNumberでは、[13~24行目]パラメータとして引き渡されたコードをキーとしてプロパティから検索する。コードに該当する値がない場合nullを返し、項目が未入力であるとみなす。プロパティに保存された値が正しい数値型の文字列でなければNumberFormatExceptionが投げられ、getAsNumberはこれを受け取りRulesExceptionに変換して再び投げる。

getAsStringでは、([26~29行目])パラメータとして引き渡されたコードをキーとしてプロパティから検索された文字値を返す。コードに該当する値がない場合、getAsStringはProperties.getPropertyが返したnullをそのまま返し、項目が入力されていないものとして処理する。

getNumericModeでは、([31~34行目])数値型項目の値が倍精度浮動小数点数であることをルールエンジンに伝える。

プロパティのキーが項目のエイリアスと一致しているため、このPropertiesProviderを使用する際はRuleInterface.executeを呼び出す時にコード体系をエイリアスにする必要がある。

#### com.innoexpert.rulesclient.ResultConsumer

com.innoexpert.rulesclient.ResultConsumerインタフェースは、ルールエンジンがルール実行結果をアプリケーションデータ領域に設定するために定義されたインタフェースである。

ResultConsumerのメソッドを理解するためには、ルールのデータ型への理解が必要である。ルールは、1つ以上のリターン項目を持つ。各項目には名前が付けられており、任意でエイリアスが付けられることもある。また、各項目には数値型、文字型、論理型のうちの1つのデータ型が指定されている。ルールはリターン形式で、**単一**または**多重**のどちらか1つの形式を持つ。単一ルールは、ルール実行結果の行数が常に1つのルールである。一方、多重ルールはルール実行結果の行数が0または1つ以上の場合もあるルールである。

ルール実行結果のデータ型は、ルールのリターン項目の形式とリターン形式によって決定される。実行結果は1つ以上のカラムで構成されるが、各カラムはルールのリターン項目に対応する。カラム数は、ルールのリターン項目の数と同一である。カラム値のデータ型は、ルールが単一ルールまたは多重ルールによって異なる。カラム値のデータ型は、単一ルールの場合にはルールリターン項目の形式が数値型、文字型、論理型の場合それぞれdouble、java.math.BigDecimal、java.lang.String、booleanであり、多重ルールの場合にはそれぞれdouble[]、java.math.BigDecimal[]、java.lang.String[]、boolean[]である。

ResultConsumerには、単一ルールと多重ルールに対し数値型、文字型、論理型項目の結果を設定できる8つのsetメソッドが存在する。ルールエンジンはルールの実行後、各結果カラム別にResultConsumer.setを呼び出してアプリケーションデータ領域にルールの結果を設定する。

ルールエンジンがルールの実行結果をアプリケーションデータ領域に設定するプロセスは、次の図のとおりである。次の図で例として挙げたルールは2つのリターン項目を持っており、項目名はRateとAmountで、項目のデータ型はそれぞれ文字型と数値型である。このルールのリターン形式が単一の場合を想定する。アプリケーションは、このルールを呼び出すためにRuleInterface.executeメソッドにResultConsumer実装オブジェクトを引き渡す。ResultConsumerのsetメソッドは、ルールエンジンが設定したカラム値を適切に変換し、アプリケーションデータ領域に保存するよう実装される。

ルールリターン項目が2つであるため、ルールの実行結果は2つのカラムを持つ。ルールのリターン形式が単一であるため、最初のカラム値の形式はjava.lang.String、2番目のカラム値の形式は数値型モードによりdoubleまたはjava.math.BigDecimalになる。ルールエンジンは、最初の項目の名前とカラム値をパラメータとし、これをアプリケーションデータ領域に設定する。設定が成功した場合、2番目の項目の名前とカラム値をパラメータとし、結果の2番目のカラム値をアプリケーションデータ領域に設定する。ローカルルールエンジン構成では、結果の設定方法が単にルールエンジンがResultConsumerオブジェクトのsetメソッドを直接呼び出すことになる可能性があるが、TCP/IPリモートルールエンジン構成では毎回カラム値を設定するたびにTCP/IP通信を伴う複雑なプロセスになってしまう可能性もある。



[図 4 ResultConsumerダイアグラム]

次の表は、ResultConsumerインタフェースに定義されたメソッドである。

[表 23 com.innoexpert.rulesclient.ResultConsumer]

|  |
| --- |
| 1 package com.innoexpert.rulesclient;  2  3 import java.math.BigDecimal;  4  5 public interface ResultConsumer  6 {  7 void set( String column, double value ) throws RulesException;  8 void set( String column, double[] arr, int offset, int size )  9 throws RulesException;  10 void set( String column, BigDecimal value ) throws RulesException;  11 void set( String column, BigDecimal[] arr, int offset, int size )  12 throws RulesException;  13 void set( String column, String value ) throws RulesException;  14 void set( String column, String[] arr, int offset, int size )  15 throws RulesException;  16 void set( String column, boolean value ) throws RulesException;  17 void set( String column, boolean[] value, int offset, int size )  18 throws RulesException;  19 } |

ResultConsumerには8つのsetメソッドが存在し、どのsetメソッドが呼び出されるのかはルールのリターン項目のデータ型によって決定される。次は、各メソッドに関する説明である。

void set( String column, double value ) throws RulesException

単一形式のルールで数値型のリターン項目の結果を設定するためのメソッドである。valueパラメータに結果が引き渡される。ResultConsumerは、アプリケーションデータ領域に値を設定中にエラーが発生する場合にRulesExceptionを投げることがある。

void set( String column, double[ ] arr, int offset, int size ) throws RulesException

多重形式のルールで数値型のリターン項目の結果を設定するためのメソッドである。arrパラメータが参照する配列のoffset番目からsize分のみの要素が有効な結果値である。アプリケーションデータ領域に値を設定中にエラーが発生する場合にRulesExceptionを投げることがある。

void set( String column, BigDecimal value ) throws RulesException

単一形式のルールで数値型のリターン項目の結果を設定するためのメソッドである。valueパラメータに結果が引き渡される。ResultConsumerは、アプリケーションデータ領域に値を設定中にエラーが発生する場合にRulesExceptionを投げる場合がある。

void set( String column, BigDecimal[ ] arr, int offset, int size ) throws RulesException

多重形式のルールで数値型のリターン項目の結果を設定するためのメソッドである。arrパラメータが参照する配列のoffset番目からsize分のみの要素が有効な結果値である。アプリケーションデータ領域に値を設定中にエラーが発生する場合にRulesExceptionを投げることがある。

void set( String column, String value ) throws RulesException

単一形式のルールで文字型のリターン項目の結果を設定するためのメソッドである。valueパラメータに結果が引き渡される。ResultConsumerは、アプリケーションデータ領域に値を設定中にエラーが発生する場合にRulesExceptionを投げる場合がある。

void set( String column, String[ ] arr, int offset, int size ) throws RulesException

多重形式のルールで文字型のリターン項目の結果を設定するためのメソッドである。arrパラメータが参照する配列のoffset番目からsize分のみの要素が有効な結果値である。アプリケーションデータ領域に値を設定中にエラーが発生する場合にRulesExceptionを投げる場合がある。

void set( String column, boolean value ) throws RulesException

単一形式のルールで論理型のリターン項目の結果を設定するためのメソッドである。valueパラメータに結果が引き渡される。ResultConsumerは、アプリケーションデータ領域に値を設定中にエラーが発生する場合にRulesExceptionを投げる場合がある。

void set( String column, boolean[ ] value, int offset, int size ) throws RulesException

多重形式のルールで論理型のリターン項目の結果を設定するためのメソッドである。arrパラメータが参照する配列のoffset番目からsize分のみの要素が有効な結果値である。アプリケーションデータ領域に値を設定中にエラーが発生する場合にRulesExceptionを投げる場合がある。

多重形式のルールでの各カラム値の配列の長さはすべて同一である。すなわち、カラムが複数の多重ルールを呼び出す場合、その結果を設定するために呼び出されるsetメソッドのsizeパラメータはすべて同一である。

各setメソッドには、設定するカラムがどのリターン項目の結果なのかを指定するcolumnパラメータがある。columnパラメータの値は、ルールを呼び出す時に指定されたコード体系によって異なる方式で指定される。指定されたコード体系がIDまたは名前の場合はリターン項目の名前が指定され、コード体系がエイリアスの場合はリターン項目のエイリアスが指定される。リターン項目次第では、エイリアスが指定されない場合がある。コード体系がエイリアスの場合、エイリアスが指定されていないリターン項目にはsetメソッドは呼び出されない。

次の表は、[表 22 ItemProvider の実装例 – PropertiesProvider]で示したようなjava.util.Propertiesオブジェクトにルール実行結果を設定するResultConsumerの実装例である。

[表 24 ResultConsumer の実装例 – PropertyConsumer]

|  |
| --- |
| 1 import java.math.BigDecimal;  2 import java.util.Properties;  3 4 import com.innoexpert.rulesclient.\*;  5  6 public class PropertiesConsumer implements ResultConsumer  7 {  8  9 private final Properties props;  10  11 public PropertiesConsumer( Properties props )  12 {  13 this.props = props;  14 }  15  16 public void set( String column, double value ) throws RulesException  17 {  18 props.setProperty( column, Double.toString( value ) );  19 }  20  21 public void set( String column, double[] arr, int offset, int size ) throws RulesException  22 {  23 StringBuilder buff = new StringBuilder();  24 for( int i = offset; size > 0; size-- )  25 {  26 if( buff.length() > 0 )  27 buff.append( ',' );  28 buff.append( Double.toString( arr[i++] ) );  29 }  30 props.setProperty( column, buff.toString() );  31 }  32  33 public void set( String column, BigDecimal value ) throws RulesException  34 {  35 props.setProperty( column, value.toString() );  36 }  37  38 public void set( String column, BigDecimal[] arr, int offset, int size ) throws RulesException  39 {  40 StringBuilder buff = new StringBuilder();  41 for( int i = offset; size > 0; size-- )  42 {  43 if( buff.length() > 0 )  44 buff.append( ',' );  45 buff.append( arr[i++].toString() );  46 }  47 props.setProperty( column, buff.toString() );  48 }  49  50 public void set( String column, String value ) throws RulesException  51 {  52 props.setProperty( column, value );  53 }  54  55 public void set( String column, String[] arr, int offset, int size ) throws RulesException  56 {  57 StringBuilder buff = new StringBuilder();  58 for( int i = offset; size > 0; size-- )  59 {  60 if( buff.length() > 0 )  61 buff.append( ',' );  62 buff.append( arr[i++] );  63 }  64 props.setProperty( column, buff.toString() );  65 }  66  67 public void set( String column, boolean value ) throws RulesException  68 { /\* ignore \*/ }  69  70 public void set( String column, boolean[] value, int offset, int size ) throws RulesException  71 { /\* ignore \*/ }  72 } |

このクラスは[4行目]でcom.innoexpert.rulesclient.ResultConsumerインタフェースを実装しており、8つのsetメソッドを実装している。コンストラクタは、java.util.Propertiesオブジェクトをパラメータとして受け取る。アプリケーションは自身が参照しているPropertiesオブジェクトをコンストラクタに引き渡し、これはPropertiesConsumerのコンストラクタからメンバ変数として参照される。

[16~19行目]と[33~36行目]は、単一ルールの数値型リターン項目の結果を設定するメソッドである。ルールエンジンから引き渡された数値を文字列に変換しプロパティに設定する。[50~53行目]は、単一ルールの文字型リターン項目の結果を設定するメソッドである。ルールエンジンから引き渡されたリターン項目名とString値をプロパティに設定する。

[21~31行目]と[38~48行目]は、多重ルールの数値型リターン項目の結果を設定するメソッドである。数値配列のoffset番目の要素からsize分のみの要素を‘,’を区切り記号として1つの文字列に変換している。変換された文字列は、リターン項目名をキーとしてプロパティに設定される。[55~65行目]は、多重ルールの文字型リターン項目の結果を設定するメソッドである。同様に、配列のoffset番目からsize分のみの文字列を1つの文字列に変換し、リターン項目名をキーとしてプロパティに設定する。

この例では、簡略化するために論理型リターン項目の結果は省略する。

#### executeメソッドのその他のパラメータ

次の表は、[表 18 I/Oアダプタ方式のexecuteメソッド]のexecuteメソッドを再度定義したものである。

[表 25 I/Oアダプタ方式のexecuteメソッド]

|  |
| --- |
| 1 package com.innoexpert.rulesclient;  2 3 public interface RuleInterface  4 {  5 void execute( String rule, ItemProvider itemProvider, ResultConsumer consumer,  6 int codetype, String date, Object ctx )  7 throws RulesException, IllegalArgumentException;  8 void execute( String rule, ItemProvider itemProvider, ResultConsumer consumer,  9 int codetype, String date, Object ctx, Trace trace, Map callctx )  10 throws RulesException, IllegalArgumentException;  11 …  12 } |

executeメソッドの最初のパラメータであるruleには、呼び出すルールのコードを引き渡す。コード体系によって、ルールのIDや名前またはエイリアスを使用できる。使用するコード体系は、4番目のパラメータであるcodetypeに定義する。codetypeの値として[表 7 コード体系を指定する定数値]の定数を使用できる。指定されたコード体系は、ルールコードのみでなく、ItemProviderのgetXXXメソッドに引き渡される項目コードやResultConsumerのsetメソッドに引き渡されるリターン項目の名前にも適用される。コード体系によってItemProviderおよびResultConsumerのメソッドに引き渡されるコードの類型については、「1.2.7.2 com.innoexpert.rulesclient.ItemProvider」と「1.2.7.3 com.innoexpert.rulesclient.ResultConsumer」をそれぞれ参照のこと。メソッドの2番目と3番目のパラメータは、ItemProviderオブジェクトとResultConsumerオブジェクトである。5番目のdateパラメータはルール呼び出しの基準日で、‘yyyy-MM-dd’または‘yyyyMMdd’の形式の文字列またはnullが指定できる。nullを指定すると、ルール呼び出し基準日としてルールエンジンのシステムの日付が使用される。6番目のctxオブジェクトはコールバックコンテキストオブジェクトで、コールバックハンドラーのサービスメソッドにそのまま引き渡される。コールバックに関するの詳細は、「1.3 コールバックAPI」を参照のこと。2番目のexecuteメソッドの7番目のtraceパラメータは、ルール実行経路トレースの結果を作成するためのパラメータで、この文書では扱わない。このパラメータをnullに設定し、トレースを無効にできる。8番目のcallctxオブジェクトもこの文書では扱わない。

#### ルールサービス呼び出し例

次の表は、I/Oオブジェクト方式のルールサービス呼び出しの例である[表 17 ルールサービス呼び出し例]のコードをI/Oアダプタ方式に変更したコードである。このコードには、[表 22 ItemProvider の実装例 – PropertiesProvider]と[表 24 ResultConsumer の実装例 – PropertyConsumer]のPropertiesProviderとPropertiesConsumerが使用されている。この例は、アプリケーションがデータを保存するための標準クラスとしてjava.util.Propertiesを使用する。データ保存クラスとしてjava.util.Mapや他のクラスの値オブジェクトを使用する場合は、これらのクラスのためのItemProviderとResultConsumerを作成すればよい。PropertiesProviderとPropertiesConsumerが使用されているため、これらのクラスがWebアプリケーションのクラスパス上に存在する必要がある。

[表 26 ルールサービス呼び出し例]

|  |
| --- |
| 1 <%@page contentType="text/html;charset=utf8"%>  2 <%@page import="java.util.Properties,com.innoexpert.rulesclient.\*"%>  3 <%  4 String tradeType = request.getParameter( "TRDTYP" );  5 String amount = request.getParameter( "AMT" );  6 String message = null;  7  8 if( tradeType != null && tradeType.length() > 0 && amount != null && amount.length() > 0 )  9 {  10 Properties props = new Properties();  11 props.setProperty( "TRDTYP", tradeType );  12 props.setProperty( "AMT", amount );  13  14 ItemProvider prov = new PropertiesProvider( props );  15 ResultConsumer cons = new PropertiesConsumer( props );  16 RuleInterface intf = ClusterManager.getInterface();  17  18 try  19 {  20 intf.execute( "COMMISSION", prov, cons, Constants.CODETYPE\_ALIAS, "2013-07-01", null );  21 }  22 catch( RulesException e )  23 {  24 message = "ERROR: " + e.getMessage();  25 }  26 finally  27 {  28 intf.close();  29 }  30  31 if( message == null )  32 {  33 String rate = props.getProperty( "CMSRATE" );  34 String commission = props.getProperty( "CMSAMT" );  35 message = "Commission is " + commission + ". Rate is " + rate + ".";  36 }  37 }  38 else  39 tradeType = amount = "";  40 %>  41  42 <html>  43 <body>  44 <form id="commission-form" method="post">  45 <li>  46 <label for="TRDTYP">Trade Type:</label>  47 <input type="text" name="TRDTYP" id="TRDTYP" value="<%=tradeType%>"/>  48 </li>  49 <li>  50 <label for="AMT">Amount:</label>  51 <input type="text" name="AMT" id="AMT" value="<%=amount%>"/>  52 </li>  53 <li>  54 <input type="submit" value="submit"/>  55 </li>  56 </form>  57 <%=message != null ? message : "&nbsp"%>  58 </body>  59 </html> |

I/Oオブジェクト方式と比較すると、[10~36行目]が変更されている。例では、[10~12行目]でユーザが入力した値をjava.util.Propertiesオブジェクトに格納している。実際の環境での値オブジェクトへのデータ設定は、アプリケーションフレームワークが担当しており、アプリケーション開発者が考慮する必要はない。[14行目]と[15行目]ではそれぞれPropertiesProviderとPropertiesConsumerを生成しており、そのコンストラクタにPropertiesオブジェクトを引き渡している。[表 22 ItemProvider の実装例 – PropertiesProvider]と[表 23 com.innoexpert.rulesclient.ResultConsumer]のコードのコンストラクタで、このオブジェクトはメンバ変数として参照される。[20行目]でexecuteメソッドが呼び出されている。このメソッド内でルールが呼び出され、ルールエンジンは項目値を照会するためItemProviderに問い合わせする。ルールエンジンがルールの実行を終了した際、その結果はResultConsumerのsetメソッドによってPropertiesオブジェクトに格納されexecuteメソッドが終了する。つまり、executeメソッドが終了した時点で、すでにPropertiesオブジェクトにルール結果が格納されていることになる。[33~35行目]は、Propertiesに格納された結果を使用するコードである。

### ルールエンジンの構成方式に基づくI/Oオブジェクト方式とI/Oアダプタ方式の性能比較

「1.2.2 com.innoexpert.rulesclient.RuleInterface」で、ルールアプリケーションはルールエンジンが使用するプロトコル、つまりルールエンジンの構成方式とは関係なく同様に作成できると述べた。しかし、入出力値の引き渡し方式によってルールエンジンの構成方式別に性能への影響が想定される。I/Oアダプタ方式では、ルールサービスの呼び出しを要求する時を除いた場合も、ルールエンジンが項目値を問い合わせするたびに、またはルール実行結果カラムを設定するたびにルールエンジンとアプリケーションがデータをやり取りする。ローカルルールエンジン構成ではデータをやり取りするためにJavaメソッドの呼び出しが発生するが、TCP/IPリモートルールエンジン構成ではネットワーク上でデータを送受信するため、前者の構成と比較して性能が低下してしまう。ルールを実行する時により多くの項目を参照するほど、ルールのカラム数が多いほどデータをやり取りする回数が増え、データをやり取りする回数が多いほど性能の低下が大きくなる傾向がある。しかし、I/Oオブジェクト方式ではルールを呼び出す時にすべての項目値が引き渡され、カラム数に関係なく一回のデータ交換でルールの結果が引き渡されるため、ルールエンジン構成方式と性能の相関関係が小さい。

ここでは、ルールアプリケーションの開発および構成において参考にできるように、サンプルシステムでルールエンジンの構成方式や入出力値の引き渡し方式による比較資料を提示する。

#### システム構成



[図 5 I/Oオブジェクト方式とI/Oアダプタ方式の比較システム]

ルールエンジンは、ローカルルールサービス以外にもTCPコネクタサーバを利用してリモートルールサービスを提供するように構成されている。リモートルールアプリケーションはTCPコネクタサーバを利用してリモートルールを呼び出すように定義し、ローカルルールアプリケーションはルールエンジンと同一のVMでJavaメソッドを利用してローカルルールを呼び出すように定義した。システムの仕様は次の表のとおりである。

[表 27 テストシステム構成]

|  | I/Oオブジェクト方式システム | I/Oアダプタ方式システム |
| --- | --- | --- |
| CPU | AMD Phenom 9550 Quad-Core | Intel T8300 |
| RAM | 4G | 4G |
| OS | Ubuntu 9.10 | MS Windows XP |
| Java | Java SE 6 | Java SE 6 |

#### テストルール

ルールサービスの性能には、入出力値のやり取り以外にルール実行時間が含まれる。この時間は理論上ではどの方式でも同一であるため、ルール実行時間が性能に及ぼす影響を最小化するために、ビジネスロジックなしで入力された値をそのまま返すルールを作成しテストに使用した。ルールの入力項目数およびリターン項目数は10個とした。

#### アプリケーション

アプリケーションは単一スレッドアプリケーションとして作成され、「1.2.8.2 テストルール」のルールを10,000回繰り返し呼び出した。

#### テスト結果

[表 28 I/Oオブジェクト方式とI/Oアダプタ方式の比較結果]

|  | ローカル | リモート | 備考 |
| --- | --- | --- | --- |
| I/Oオブジェクト方式 | 173ms | 9,297ms | リモートの方がルール1回の呼び出し当たり約0.9124ms多く所要 |
| I/Oアダプタ方式 | 144ms | 89,313ms | リモートの方がルール1回の呼び出し当たり約8.9169ms多く所要 |

[表 28 I/Oオブジェクト方式とI/Oアダプタ方式の比較結果]は、ルールエンジン構成方式別のI/Oオブジェクト方式とI/Oアダプタ方式の性能テスト結果である。所要時間は、ルール10,000回の呼び出しに要した時間である。ローカル構成では、I/Oアダプタ方式がI/Oオブジェクト方式に比べて約17%速いという結果であった。しかし、リモート構成ではI/Oアダプタ方式がI/Oオブジェクト方式に比べて約860%も多く要した。I/Oオブジェクト方式はローカルルールエンジン構成からリモート構成に切替えた際に性能の低下が比較的見受けられないが、I/Oアダプタ方式はそうでないことが確認できる。

上の結果では、アプリケーションがローカルルールエンジン構成とリモートルールエンジン構成の両方で動作する必要がある場合、I/Oオブジェクト方式の性能が良いと言える。一方、アプリケーションがローカルルールエンジン構成のみで動作する場合は、I/Oアダプタ方式が適していると言える。どの呼び出し方式を使用するかは、システムの構成方式やアプリケーションの動作環境、アプリケーションのメンテナンスの必要性などを総合的に考慮して決定する必要がある。

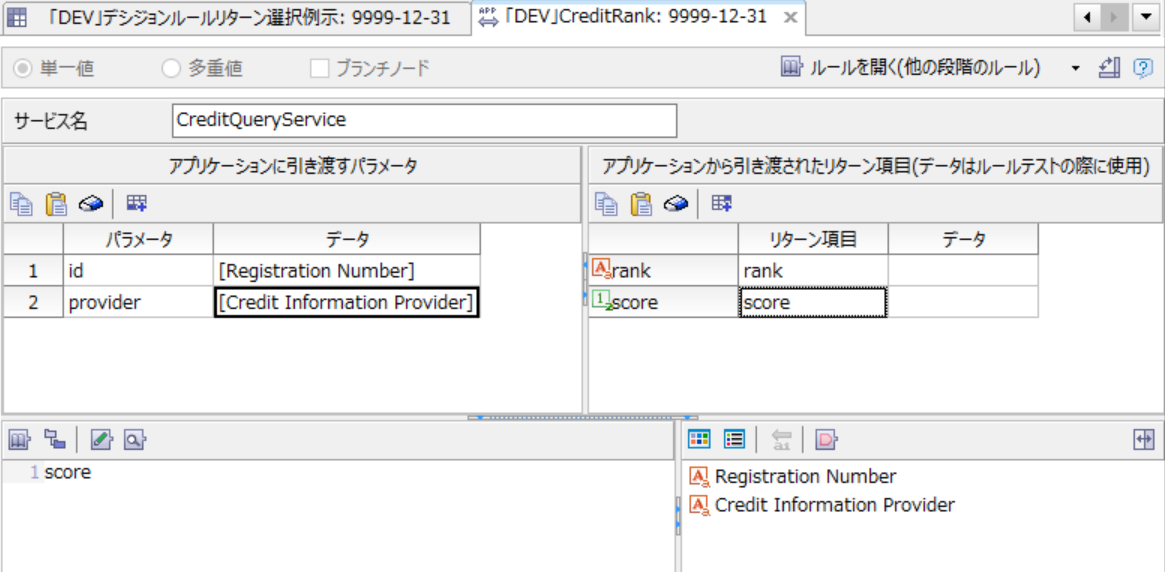
## コールバックAPI

コールバックAPIは、ルールエンジンがルール実行中にアプリケーション側のコードを呼び出せるようにするクラスとインタフェースの集合である。ルールエンジンがアプリケーション側のコードを呼び出せるようにするサービスを、コールバックサービスという。コールバックサービスは、次の目的で作成される。

* ルールサービス中で暗号化モジュールのようなレガシー(Legacy)コードを実行するため
* 入力項目として引き渡せないアプリケーションデータを使用するため、例えば、ルールエンジンはアプリケーションが処理中のデータベーストランザクションを利用してクエリができないが、コールバックサービスを利用することでアプリケーショントランザクションを利用してクエリができる。

### コールバックルール

コールバックサービスは、コールバックルールの実行の一部として呼び出される。コールバックルールは、次の図のような形式を持つ。



[図 6 コールバックルール]

**サービス名**は、このコールバックルールで呼び出すコールバックサービスの名前である。Javaクラス名ではなくコールバックサービス名という別の識別子を使用する事由は、ルールアプリケーションが必ずしもJavaで作成されるわけではないからである。コールバックサービスはルールを呼び出すアプリケーションに実装されている必要があるため、Javaアプリケーションの場合にはJavaクラスで実装されるが、C++で作成されたアプリケーションの場合にはC++で作成される。\*2 また別の事由としては、アプリケーションごとに互いに異なるコールバックサービス実装クラス(これを**コールバックサービス**ハンドラーという)を使用できるようにするためである。例えば、顧客情報を取得するコールバックサービスがあり、コールバックルールでこのサービスを呼び出す構成で、Aというアプリケーションは顧客情報が存在するデータベースサーバに接続可能で、Bというアプリケーションはデータベースサーバに接続が不可能で顧客情報をファイルとして管理していると想定した時、この構成では、2つのコールバックサービスハンドラークラスが必要である。1つはAアプリケーション環境で動作するデータベースサーバから顧客情報を取得するハンドラーで、もう1つはBアプリケーション環境で動作するファイルから顧客情報を取得するハンドラーとなる。コールバックルールはどのような環境で動作するのかわからないため、ここには実装クラスの情報とは無関係な別の識別子であるサービス名を指定し、このサービス名と、このサービスを処理するコールバックサービスハンドラーとのマッピングをアプリケーション別で別途存在する設定ファイルに記述することで、こうした構成をサポートする。コールバックサービスとコールバックサービスハンドラークラスのマッピングがアプリケーション(Webアプリケーションのこともある)別に行われるため、開発者が実装したコールバックサービスハンドラーをアプリケーションに反映させるためには、システム管理者の介入が必要となる場合もある。

**プログラムに引き渡すパラメータデータ**には、コールバックサービスの入力パラメータを定義する。これらのパラメータを、**コールバック入力パラメータ**という。パラメータの名前の付与に関する制約事項はないが、可読性のために簡潔な名前を付けることを推奨する。データ欄には、コールバック入力パラメータの値を評価するルール表現式(Rule Expression)が指定される。ルールエンジンはコールバックサービスを呼び出す前にこのルール表現式を検証し、各パラメータに割り当ててコールバックサービスを呼び出す時にこれを引き渡す。コールバックサービスハンドラーでは、コールバック入力パラメータの名前をキーとしてこれらの値を照会できる。また、評価された値のデータ型に基づいてコールバック入力パラメータのデータ型が決定される。[図 6 コールバックルール]で、コールバック入力パラメータidの値は“[Registration Number]”というルール表現式を評価して決定される。[Registration Number]項目は、文字型の項目であるため、この結果は文字値になるはずで、idパラメータのデータ型は文字型となる。コールバック入力パラメータに割り当てられる値は、数値、文字値、論理値またはこれらの配列値のみ許可される。

**プログラムから引き渡されるリターンデータ**には、コールバックサービスの処理結果を受け取るパラメータの名前を記述する。これらのパラメータを、**コールバック出力パラメータ**という。コールバックサービスハンドラーは、コールバックサービスの処理結果をこれらのパラメータに格納し返す。コールバック出力パラメータは、ルールのリターン項目に対応し、パラメータの値が対応するフィールドの結果として使用される。コールバック出力パラメータのデータ型は、対応するリターン項目のデータ型と同様に指定される。コールバックサービスハンドラーは、コールバック出力パラメータに配列形式で値を割り当てることができるが、このパラメータの値がルールの結果として使用されるため、ルールの形式に反してはならない。例えば、単一形式のコールバックルールの出力パラメータに配列値の設定や、2つ以上の出力パラメータに長さと異なる配列値を設定する場合は、エラーが発生する。

1. コールバックAPIは、JavaとC/C++のみサポートする。

### コールバックサービスの処理フロー

ルールアプリケーションがルールサービスを呼び出し、呼び出されたルールの下位でコールバックルールが実行される場合、その処理プロセスは次のようになる。

1. ルールアプリケーションは、ルールインタフェースクラスタマネージャからルールインタフェースを使用してルールサービスを呼び出す。
2. ルールエンジンは、要求されたルールを実行し、その下位ルールとしてコールバックルールを実行する。
3. ルールエンジンは各コールバック入力パラメータのルール表現式を実行し、コールバック入力パラメータに割り当てる値とコールバック入力パラメータのデータ型を決定する。
4. ルールエンジンは、ルールのリターン項目のデータ型を利用して対応するコールバック出力パラメータのデータ型を決定する。
5. ルールエンジンは、コールバックサービス名とコールバック入力パラメータを1つのJavaオブジェクトに格納する。このオブジェクトを、コールバック要求オブジェクト(Callback Request Object)という。ルールエンジンは、コールバック出力パラメータを1つのJavaオブジェクトに格納する。このオブジェクトを、コールバック応答オブジェクト(Callback Response Object)という。コールバック出力パラメータは、パラメータの名前とデータ型情報のみを持っている。
6. ルールエンジンは、コールバック要求オブジェクトとコールバック応答オブジェクトをルールサービスを呼び出したルールインタフェースに引き渡す。
7. ルールインタフェースは、自身が属するインタフェースクラスタのコールバックレジストリからコールバックサービスに該当するコールバックサービスハンドラーファクトリを検索する。該当するコールバックサービスハンドラーファクトリがない場合は、コールバックレジストリから基本コールバックサービスハンドラーファクトリを検索する。基本コールバックサービスハンドラーファクトリがない場合は、ルールエンジンにエラーを返す。
8. ルールインタフェースは、検索されたコールバックサービスハンドラーファクトリを利用して、コールバックサービスハンドラーを取得する。この時、コールバックサービスハンドラーファクトリは、新しいコールバックサービスハンドラーの生成や、前に生成されてキャッシュされているコールバックサービスハンドラーを返すことがある。これは、コールバックサービスハンドラーファクトリの実装方式によって決定される。
9. ルールインタフェースは、コールバックサービスハンドラーオブジェクトのdoCallbackメソッドを呼び出す。このメソッドのパラメータとして、コールバック要求オブジェクトとコールバック応答オブジェクトが引き渡される。
10. コールバックサービスハンドラーは、コールバック入力オブジェクトからコールバックサービス名と必要なパラメータの値を読み取りサービスロジックを実行する。コールバック出力オブジェクトからコールバック出力パラメータを検索し、コールバックサービスの結果をパラメータの値として設定する。
11. ルールエンジンは、各リターン項目に対応するコールバック出力パラメータを探してその値を該当リターン項目の値として設定し、コールバックルールの実行を終了する。

### com.innoexpert.rulesclient.CallbackHandler

コールバックサービスハンドラーを表現するインタフェースである。コールバックサービスを開発するためには、com.

innoexpert.-rulesclient.CallbackHandlerインタフェースとcom.innoexpert.rulesclient.CallbackHandlerFactoryを実装する必要がある。CallbackHandlerはコールバックサービスロジックに位置し、CallbackHandlerFactoryはCall

backHandlerを生成または提供する役割をする。次の表は、CallbackHandlerインタフェースを示す。

[表 29 com.innoexpert.rulesclient.CallbackHandler]

|  |
| --- |
| 1 package com.innoexpert.rulesclient;  2  3 public interface CallbackHandler  4 {  5 public void doCallback( CallbackReq req, CallbackRes res, Object context )  6 throws CallbackException;  7 } |

CallbackHandlerは、doCallbackという唯一のメソッドを持っている。doCallbackは、3つのパラメータを持つ。reqパラメータはCallbackReq型のオブジェクトで、コールバックサービスの入力情報を持っているオブジェクトである。このオブジェクトには、コールバックサービス名とコールバック入力パラメータについての情報がある。CallbackReqに関する詳細事項については、「1.3.4 com.innoexpert.rulesclient.CallbackReq」を参照のこと。resパラメータはCallback

Res型のオブジェクトで、コールバックサービスの出力情報を持っているオブジェクトである。CallbackResについては、「1.3.5 com.innoexpert.rulesclient.CallbackRes」を参照のこと。最後のパラメータであるcontextパラメータには、[表 6 com.innoexpert.rulesclient.RuleInterface]のRuleInterface.executeメソッドのctxに引き渡されたオブジェクトをそのまま引き渡す。このオブジェクトを、コールバックコンテキストオブジェクト(Callback Context Object)という。

コールバックサービスの実質的なロジックがdoCallbackに位置する。このメソッドではreqパラメータからコールバック入力パラメータを読み取り、入力値を利用して要求したサービスを処理し、その結果をresパラメータが参照するコールバック出力パラメータに設定する必要がある。コールバックロジックの実行中にエラーが発生すると、Callback

Exceptionを投げることがある。CallbackExceptionについては、「1.3.8 com.innoexpert.rulesclient.CallbackException」を参照のこと。

CallbackHandlerクラスを実装する際、メンバ変数の使用には注意する必要がある。CallbackHandlerオブジェクトは、毎回コールバック要求を受け取るたびに生成されて各要求が互いに異なるCallbackHandlerオブジェクトによって処理されることもあり、シングルトン(Singleton)形式で実装され複数のコールバック要求が1つのオブジェクトによって同時に処理されることもある。後者の場合、別途スレッドセーフになるように実装しない場合は、問題が発生することもある。CallbackHandlerオブジェクトはCallbackHandlerFactoryが生成し、生成方式はファクトリクラスをどのように実装するかによって異なる。CallbackHandlerFactoryに関する詳細事項については、「1.3.7 com.innoexpert.rulesclient.CallbackHandlerFactory」を参照のこと。

[表 30 アプリのデータベースコネクションをコールバックコンテキストオブジェクトに引き渡す例]と[表 31 アプリケーショントランザクション上でクエリをするコールバックハンドラー]は、それぞれデータベースコネクションをコールバックコンテキストオブジェクトに引き渡すアプリケーションと、このコネクションを受けてアプリケーションのトランザクション上でデータを照会するコールバックハンドラーの例である。

[表 30 アプリのデータベースコネクションをコールバックコンテキストオブジェクトに引き渡す例]

|  |
| --- |
| 1 java.sql.Connection conn = …  2  3 // Start a database transaction  4  5 RuleReq req = …  6 RuleInterface intf = ClusterManager.getInterface();  7 ResultSet rs;  8  9 try  10 {  11 rs = intf.execute( req, Contants.CODETYPE\_ID, conn );  12 }  13 finally  14 {  15 intf.close();  16 }  17  18 // Terminate the database transaction |

[表 30 アプリのデータベースコネクションをコールバックコンテキストオブジェクトに引き渡す例]の[1~3行目]で、ルールアプリケーションはデータベースコネクションを取得してトランザクションを実行する。このトランザクションは、[5~16行目]においてルール呼び出しを終了する時までは終了されない。一般的に[11行目]で呼び出されたルールの実行中は、データベースルールを利用してデータを照会する場合、このトランザクションで変更されたデータは照会されない。\*3 ルールを実行中、ルールを呼び出したアプリケーションのデータベーストランザクション上でデータベースにアプローチするためにコールバックが使用されることがある。そのために、[11行目]でデータベースコネクションをコールバックコンテキストオブジェクトに引き渡している。

[表 31 アプリケーショントランザクション上でクエリをするコールバックハンドラー]

|  |
| --- |
| 1 public void doCallback( CallbackReq req, CallbackRes res, Object ctx ) throws CallbackException  2 {  3 if( ctx == null || !( ctx instanceof java.sql.Connection ) )  4 throws new CallbackException( "Not a database connection" );  5  6 java.sql.Connection conn = (java.sql.Connection)ctx;  7  8 // query within application's transaction  9 …  10 } |

[表 31 アプリケーショントランザクション上でクエリをするコールバックハンドラー]は、アプリケーショントランザクション上でクエリを実行するコールバックハンドラーを示す。[表 30 アプリのデータベースコネクションをコールバックコンテキストオブジェクトに引き渡す例]の[11行目]でルールが呼び出され、このハンドラーのdoCallbackが呼び出されるためにはコールバックルールが作成され、適切にコールバックハンドラーファクトリクラスがルールインタフェースクラスタに設定される必要があるが、この例では省略する。[表 31 アプリケーショントランザクション上でクエリをするコールバックハンドラー]の[1行目]のctxには[表 30 アプリのデータベースコネクションをコールバックコンテキストオブジェクトに引き渡す例]の[11行目]で引き渡されたコールバックコンテキストオブジェクト、つまりデータベースコネクションオブジェクトが引き渡される。[表 30 アプリのデータベースコネクションをコールバックコンテキストオブジェクトに引き渡す例]のアプリケーションではない他のアプリケーションから呼び出されてnullまたはデータベースコネクションオブジェクトではない他のオブジェクトが引き渡されることがあるため、[3~4行目]でコールバックコンテキストオブジェクトの有効性を検証し、有効でない場合はCallbackExceptionを投げている。[6行目]ではコールバックコンテキストオブジェクトをデータベースコネクションとして型変換し、[8行目]でデータベースにクエリをしている。

1. トランザクション分離レベル(Transaction Isolation Level)とオートコミット(Auto Commit)オプションによって照会できることもある。

### com.innoexpert.rulesclient.CallbackReq

com.innoexpert.rulesclient.CallbackReqオブジェクトは、コールバックサービスの入力情報を照会できるインタフェースである。ルールエンジンまたはルールインタフェースクラスタによって生成され、CallbackHandler.doCallbackメソッドにパラメータとして引き渡される。次の表は、CallbackReqインタフェースに定義されたメソッドを示す。

[表 32 com.innoexpert.rulesclient.CallbackReq]

|  |
| --- |
| 1 package com.innoexpert.rulesclient;  2  3 import java.util.Iterator;  4  5 public interface CallbackReq  6 {  7 public String getRuleID();  8 public String getDate();  9 public String getServiceName();  10 public int countParams();  11 public Iterator<CallbackParam> getParams();  12 public CallbackParam lookupParam( String name );  13 public CallbackParam lookupRequiredParam( String name ) throws CallbackException;  14 NumericMode getNumericMode();  15 } |

次は、各メソッドに関する説明である。

String getRuleID()

コールバックサービスを呼び出したコールバックルールのIDを返す。ルールサービスを呼び出す際、指定したコード体系の種類に問わずルールIDが返される。

String getDate()

コールバックサービスを呼び出したルールサービスのルール実行基準日を返す。RuleInterface.executeまたはRuleReqオブジェクトに指定されたルール実行基準日が返される。executeメソッドまたはRuleReqオブジェクトのルール実行基準日がnullや指定されない場合においては、ルールサービスが開始された時点のルールエンジンのシステム日付が引き渡される。

String getServiceName()

コールバックサービスを呼び出したコールバックルールに指定されたコールバックサービス名を返す。1つのコールバックサービスハンドラークラスが、複数のコールバックサービスを処理するように設定できる。コールバックサービスに該当するコールバックサービスハンドラーがルールアプリケーション設定に指定されてない場合、デフォルトコールバックハンドラーがコールバックサービスを処理する。コールバックサービス名に基づいて異なる処理をする場合に、このメソッドが使用できる。

int countParams()

コールバック入力パラメータの数を返す。

Iterator<CallbackParam> getParams()

コールバック入力パラメータのIterator型オブジェクトを返す。

CallbackParam lookupParam( String name )

パラメータとして指定された名前のコールバック入力パラメータを検索する。指定された名前のコールバック入力パラメータが与えられない場合は、nullを返す。コールバックハンドラーは、この入力パラメータでコールバックルールから引き渡されたコールバックサービスの処理に必要な入力値を読み取ることができる。

CallbackParam lookupRequiredParam( String name ) throws CallbackException

パラメータとして指定された名前のコールバック入力パラメータを検索する。指定された名前のコールバック入力パラメータが与えられない場合は、CallbackExceptionが投げられる。コールバックサービスを処理するに当たって、あるコールバック入力パラメータは必須である場合や、あるパラメータはオプションである場合がある。必須入力パラメータを検索する際にlookupParamメソッドを使用した場合、次の表のように毎回nullであるかチェックする必要があるという手間がかかってしまうことがある。

[表 33 lookupParamメソッドを利用した必須コールバック入力パラメータの照会]

|  |
| --- |
| 1 CallbackParam required = cbreq.lookupParam( "some name" );  2  3 if( required == null )  4 throw new CallbackException( "…" ); // The input parameter is not specified.  5  6 // Use the parameter |

lookupRequiredParamメソッドを利用すると、[表 33 lookupParamメソッドを利用した必須コールバック入力パラメータの照会]のコードを次の表のように簡略化できる。

[表 34 lookupRequiredParamメソッドを利用したコードの簡略化]

|  |
| --- |
| 1 // if not specified, CallbackException is thrown.  2 CallbackParam required = cbreq.lookupRequiredParam( "some name" );  3  4 // Use the parameter |

NumericMode getNumericMode()

コールバックサービスが動作するルールエンジンプロキシの数値型モードを返却する。

### com.innoexpert.rulesclient.CallbackRes

com.innoexpert.rulesclient.CallbackResオブジェクトは、コールバックサービスの出力パラメータを照会できるインタフェースである。ルールエンジンまたはルールインタフェースクラスタによって生成され、CallbackHandler.doCall

backメソッドにパラメータとして引き渡される。次の表はCallbackResインタフェースに定義されたメソッドを示す。

[表 35 com.innoexpert.rulesclient.CallbackRes]

|  |
| --- |
| 1 package com.innoexpert.rulesclient;  2  3 import java.util.Iterator;  4  5 public interface CallbackRes  6 {  7 public int countParams();  8 public Iterator<CallbackParam> getParams();  9 public CallbackParam lookupParam( String name );  10 public CallbackParam lookupRequiredParam( String name ) throws CallbackException;  11 } |

次は、各メソッドに関する説明である。

int countParams()

コールバック出力パラメータの数を返す。

Iterator<CallbackParam> getParams()

コールバック出力パラメータに関するIterator型オブジェクトを返す。

CallbackParam lookupParam( String name )

パラメータとして指定された名前のコールバック出力パラメータを検索する。指定された名前のコールバック出力パラメータが付与されていない場合は、nullを返す。doCallbackが呼び出された時点では、このコールバック出力パラメータには名前とデータ型のみが指定されていて、値は指定されていない。コールバックハンドラーは、サービス処理の結果をこの出力パラメータに格納する。ハンドラーはコールバックルールから引き渡されたすべてのコールバック出力パラメータに値を格納し、配列形式の値である場合、その配列の長さはすべて同一である必要がある。

public CallbackParam lookupRequiredParam( String name ) throws CallbackException

パラメータとして指定された名前のコールバック出力パラメータを検索する。指定された名前のコールバック出力パラメータが与えられない場合は、CallbackExceptionが投げられる。同一コールバックサービスを呼び出すコールバックルールが多数存在することがあり、各コールバックルールが必要とするコールバック出力パラメータが互いに異なる場合がある。コールバックハンドラーは、すべてのコールバックルールが必須で引き渡されるコールバック出力パラメータがある場合、CallbackReq.lookupRequiredParamメソッドと同一方法でコードを簡略化できる。

### com.innoexpert.rulesclient.CallbackParam

com.innoexpert.rulesclient.CallbackParamインタフェースは、コールバック入力パラメータまたはコールバック出力パラメータを表現するインタフェースである。コールバックパラメータは、ルールエンジンまたはルールインタフェースクラスタによって生成されCallbackHandler.doCallbackメソッドに引き渡される。

次の表は、com.innoexpert.rulesclient.CallbackParamインタフェースに定義されたメソッドである。

[表 36 com.innoexpert.rulesclient.CallbackParam]

|  |
| --- |
| 1 package com.innoexpert.rulesclient;  2  3 import java.math.BigDecimal;  4  5 public interface CallbackParam  6 {  7 String getName();  8 short getType();  9 int getRowCount();  10 double getNumValue( int index )  11 throws IndexOutOfBoundsException, UnsupportedOperationException;  12 double[] getNumValues() throws UnsupportedOperationException;  13 BigDecimal getDecValue( int index )  14 throws IndexOutOfBoundsException, UnsupportedOperationException;  15 BigDecimal[] getDecValues() throws UnsupportedOperationException;  16 String getStrValue( int index )  17 throws IndexOutOfBoundsException, UnsupportedOperationException;  18 String[] getStrValues() throws UnsupportedOperationException;  19 boolean getBoolValue( int index )  20 throws IndexOutOfBoundsException, UnsupportedOperationException;  21 boolean[] getBoolValues() throws UnsupportedOperationException;  22 void removeAll() throws UnsupportedOperationException;  23 void addNumValue( double val ) throws UnsupportedOperationException;  24 void addNumValues( double[] val, int offset, int len )  25 throws UnsupportedOperationException;  26 void addDecValue( BigDecimal val ) throws UnsupportedOperationException;  27 void addDecValues( BigDecimal[] val, int offset, int len )  28 throws UnsupportedOperationException;  29 void addStrValue( String val ) throws UnsupportedOperationException;  30 void addStrValues( String[] val, int offset, int len )  31 throws UnsupportedOperationException;  32 void addBoolValue( boolean val ) throws UnsupportedOperationException;  33 void addBoolValues( boolean[] val, int offset, int len )  34 throws UnsupportedOperationException;  35 } |

次は、各メソッドに関する説明である。

String getName()

コールバックパラメータの名前を返す。

short getType()

コールバックパラメータのデータ型を返す。データ型の値については、「1.2.4 データ型と関連定数」を参照のこと。

int getRowCount()

コールバックパラメータの値の配列の長さを返す。

double getNumValue( int index ) throws IndexOutOfBoundsException, UnsupportedOperationException

コールバックパラメータの値の配列の中からindex番目の倍精度浮動小数点数を返す。indexは、0ベース(0-base)のインデックスである。インデックスが0より小さい、または値の配列の長さ以上の場合、IndexOutOfBoundsExceptionが投げられる。このコールバックパラメータのデータ型が数値型でない場合、UnsupportedOperationExceptionが投げられる。

double[ ] getNumValues() throws UnsupportedOperationException

コールバックパラメータの倍精度浮動小数点数の配列を返す。このパラメータのデータ型が数値型でない場合は、UnsupportedOperationExceptionが投げられる。返される配列オブジェクトの長さはgetRowCount()によって戻り値の配列の長さと同一またはそれより長い場合があり、getRowcount()が指定する範囲内の値のみが有効な値である。

BigDecimal getDecValue( int index ) throws IndexOutOfBoundsException, UnsupportedOperationException

コールバックパラメータの値の配列の中からindex番目のBigDecimal値を返す。 indexは、0ベース(0-base)のインデックスである。インデックスが0より小さい、または値の配列の長さ以上の場合、IndexOutOfBoundsExceptionが投げられる。このコールバックパラメータのデータ型が数値型でなければ、UnsupportedOperationExceptionが投げられる。

BigDecimal[] getDecValues() throws UnsupportedOperationException

コールバックパラメータのBigDecimal値の配列を返す。このパラメータのデータ型が数値型でない場合は、UnsupportedOperationExceptionが投げられる。返される配列オブジェクトの長さはgetRowCount()によって戻り値の配列の長さと同一またはそれより長い場合があり、getRowcount()が指定する範囲内の値のみが有効な値である。

String getStrValue( int index ) throws IndexOutOfBoundsException, UnsupportedOperationException

コールバックパラメータの値の配列の中からindex番目の文字値を返す。indexは、0ベース(0-base)のインデックスである。インデックスが0より小さい、または値の配列の長さ以上の場合、IndexOutOfBoundsExceptionが投げられる。このコールバックパラメータのデータ型が文字型でなければ、UnsupportedOperationExceptionが投げられる。

String[ ] getStrValues() throws UnsupportedOperationException

コールバックパラメータの文字値の配列を返す。このパラメータのデータ型が文字型でない場合は、UnsupportedOperationExceptionが投げられる。返される配列オブジェクトの長さはgetRowCount()によって戻り値の配列の長さと同一またはそれより長い場合があり、getRowcount()が指定する範囲内の値のみが有効な値である。

boolean getBoolValue( int index ) throws IndexOutOfBoundsException, UnsupportedOperationException

コールバックパラメータの値の配列の中からindex番目の論理値を返す。indexは、0ベース(0-base)のインデックスである。インデックスが0より小さい、または値の配列の長さ以上の場合、IndexOutOfBoundsExceptionが投げられる。このコールバックパラメータのデータ型が論理型でなければ、UnsupportedOperationExceptionが投げられる。

boolean[ ] getBoolValues() throws UnsupportedOperationException

コールバックパラメータの論理値の配列を返す。このパラメータのデータ型が論理型でない場合は、UnsupportedOperationExceptionが投げられる。返される配列オブジェクトの長さはgetRowCount()によって戻り値の配列の長さと同一またはそれより長い場合があり、getRowcount()が指定する範囲内の値のみが有効な値である。

void addNumValue( double val ) throws UnsupportedOperationException

コールバックパラメータの値の配列に数値を追加する。コールバックパラメータのデータ型が数値型でない場合、UnsupportedOperationExceptionが投げられる。

void addNumValues( double[ ] val, int offset, int len ) throws UnsupportedOperationException

コールバックパラメータの値の配列に数値の配列を追加する。valパラメータによって与えられた配列の

offset番目の値からlen分のみの数値が値の配列に追加される。コールバックパラメータのデータ型が数値型でない場合、UnsupportedOperationExceptionが投げられる。

void addDecValue( BigDecimal val ) throws UnsupportedOperationException

コールバックパラメータの値の配列にBigDecimal数値を追加する。コールバックパラメータのデータ型が数値型でない場合、UnsupportedOperationExceptionが投げられる。

void addDecValues( BigDecimal[ ] val, int offset, int len ) throws UnsupportedOperationException

コールバックパラメータの値の配列にBigDecimal数値の配列を追加する。valパラメータによって与えられた配列の

offset番目の値からlen分のみの数値が値の配列に追加される。コールバックパラメータのデータ型が数値型でない場合、UnsupportedOperationExceptionが投げられる。

void addStrValue( String val ) throws UnsupportedOperationException

このコールバックパラメータの値の配列に文字値を追加する。コールバックパラメータのデータ型が文字型でない場合、UnsupportedOperationExceptionが投げられる。

void addStrValues( String[ ] val, int offset, int len ) throws UnsupportedOperationException

このコールバックパラメータの値の配列に文字値の配列を追加する。valパラメータによって与えられた配列のoffset番目の値からlen分のみの文字値が値の配列に追加される。コールバックパラメータのデータ型が文字型でない場合、UnsupportedOperationExceptionが投げられる。

void addBoolValue( boolean val ) throws UnsupportedOperationException

コールバックパラメータの値の配列に論理値を追加する。コールバックパラメータのデータ型が論理型でない場合、UnsupportedOperationExceptionが投げられる。

void addBoolValues( boolean[ ] val, int offset, int len ) throws UnsupportedOperationException

コールバックパラメータの値の配列に論理値の配列を追加する。valパラメータによって与えられた配列のoffset番目の値からlen分のみの論理値が値の配列に追加される。コールバックパラメータのデータ型が論理型でない場合、UnsupportedOperationExceptionが投げられる。

### com.innoexpert.rulesclient.CallbackHandlerFactory

com.innoexpert.rulesclient.CallbackHandlerFactoryは、コールバックサービスハンドラーファクトリを表現するインタフェースである。コールバックサービスハンドラーファクトリは頻繁にコールバックサービスハンドラーとペアで実装され、ユーザ定義コールバックサービスハンドラーオブジェクトを生成する役割がある。

コールバックサービスは、次のような順序で処理される。

1. コールバックサービスハンドラーファクトリクラスの名前とこのファクトリが処理すべきサービス名がルールインタフェースクラスタの設定にペアで登録される。ルールインタフェースクラスタが初期化される時に、コールバックサービス別に1つのファクトリインスタンスが生成される。
2. アプリケーションがルールインタフェースを利用してルールサービスを呼び出す。
3. ルールエンジンがサブルールの実行中にコールバックルールを実行した場合、ルールインタフェースにコールバックサービス要求をする。
4. ルールインタフェースはクラスタから要求されたサービスを処理するコールバックサービスハンドラーファクトリインスタンスを検索し、検索されたファクトリからコールバックサービスハンドラーを取得する。コールバックサービスハンドラーのdoCallbackメソッドを呼び出してコールバックサービスを処理し、ハンドラーはファクトリに返す。処理された結果はルールエンジンに返す。
5. ルールエンジンは残りのサブルールを実行する。ルール実行中に他のコールバックルールを実行する場合、3から繰り返す。
6. ルールサービスの結果をアプリケーションに返す。

次の表は、com.innoexpert.rulesclient.CallbackHandlerFactoryに定義されたメソッドである。

[表 37 com.innoexpert.rulesclient.CallbackHandlerFactory]

|  |
| --- |
| 1 package com.innoexpert.rulesclient;  2  3 public interface CallbackHandlerFactory  4 {  5 public CallbackHandler createCallbackHandler() throws CallbackException;  6 public void destroyCallbackHandler( CallbackHandler handler );  7 } |

次は、各メソッドに関する説明である。

CallbackHandler createCallbackHandler() throws CallbackException

コールバックサービスハンドラーオブジェクトを返す。ルールインタフェースは、毎回コールバックサービスを要求されるたびにこのメソッドを呼び出してコールバックサービスハンドラーをインポートしコールバックサービスを処理する。適切なコールバックサービスハンドラーオブジェクトを返すことができない時は、CallbackExceptionを投げることがある。

void destroyCallbackHandler( CallbackHandler handler )

ルールインタフェースは、コールバックサービスハンドラーを利用してコールバックサービスを処理した後、このメソッドを利用してコールバックサービスハンドラーをファクトリに返す。handlerパラメータは、コールバックサービスの処理に使用したハンドラーオブジェクトである。

createCallbackHandlerメソッドで常に新しいコールバックサービスハンドラーオブジェクトを生成し、返す必要はない。コールバックサービスハンドラーファクトリがコールバックサービスハンドラーを管理する方式には制限がないが、次のような方式がある。

* シングルトンハンドラー方式
* コールバックサービスごとに新しいコールバックハンドラーオブジェクトを生成する方式
* ハンドラープール方式

次は、各類型に関する説明および例である。後に説明する方法ほど実装するのが複雑になるので、コールバックサービスの実装条件によって適当な方式を選択する必要がある。

#### シングルトンハンドラー方式

この方式は最も単純な方式で、1つのファクトリが1つのハンドラーのみを生成し、これを提供する方式である。次の表は、この方式の例を示す。

[表 38 シングルトンハンドラー方式の例]

|  |
| --- |
| 1 public class CallbackHandlerImpl implements CallbackHandler  2 {  3 public void doCallback( CallbackReq req, CallbackRes res, Object res ) throws CallbackException  4 {  5 // do something  6 }  7 }  8  9 public class CallbackHandlerFactoryImpl implements CallbackHandlerFactory  10 {  11 private CallbackHandlerImpl handler = new CallbackHandlerImpl();  12  13 public CallbackHandler createCallbackHandler()  14 {  15 return handler;  16 }  17  18 public void destroyCallbackHandler( CallbackHandler handler )  19 {  20 // do nothing  21 }  22 } |

[1~7行目]はコールバックサービスハンドラーの実装部分で、[9~22行目]はコールバックサービスハンドラーファクトリの実装部分である。[11行目]でハンドラーインスタンスを生成してファクトリのメンバに設定し、createCallbackHandlerメソッドではこれを返す。そのため、すべてのコールバックサービス要求に対して(同時に発生してもそうでなくとも)、同一ハンドラーオブジェクトが返される。この方法はコールバックサービスの呼び出すごとにハンドラーオブジェクトを生成しないため、メモリの使用面で利便性が高い。但し、1つのコールバックサービスハンドラーインスタンスを複数のスレッドで使用できるため、メンバ変数の使用に注意が必要である。この方式は、次の表のように、ハンドラーとファクトリクラスをまとめて簡略化することもできる。

[表 39 シングルトンハンドラー方式の別の例]

|  |
| --- |
| 1 public class CallbackHandlerFactoryImpl  2 implements CallbackHandlerFactory, CallbackHandler  3 {  4 public CallbackHandler createCallbackHandler()  5 {  6 return this;  7 }  8  9 public void destroyCallbackHandler( CallbackHandler handler )  10 {  11 // do nothing  12 }  13  14 public void doCallback( CallbackReq req, CallbackRes res, Object res ) throws CallbackException  15 {  16 // do something  17 }  18 } |

#### コールバックサービスごとに新しいコールバックハンドラーオブジェクトを生成する方式

「1.3.7.1 シングルトンハンドラー方式」の方式は単純な構造であるが、コールバックハンドラーがメンバ変数を使用できないというデメリットがある。実際にコールバックサービスのロジックが複雑になるに伴い追加のメソッドが作られ、データを共有するためにメンバ変数を使用する必要性が生じる。メンバ変数を使用するためにはコールバックサービスを実行するたびに毎回ハンドラーオブジェクトを生成する必要があり、最も簡単な方法としてCallbackHandlerFactoryImpl.createCallbackHandlerでコールバックハンドラーを生成して返す方法がある。次の表は、この方式の例を示す。

[表 40 コールバックサービスごとに新しいコールバックハンドラーオブジェクトを生成する方式の例]

|  |
| --- |
| 1 public class CallbackHandlerImpl implements CallbackHandler  2 {  3 private String var1;  4  5 public void doCallback( CallbackReq req, CallbackRes res, Object res ) throws CallbackException  6 {  7 // use var1  8 }  9  10 private void someOtherMethod()  11 {  12 // use var1  13 }  14 }  15  16 public class CallbackHandlerFactoryImpl implements CallbackHandlerFactory  17 {  18 public CallbackHandler createCallbackHandler()  19 {  20 return new CallbackHandlerImpl();  21 }  22  23 public void destroyCallbackHandler( CallbackHandler handler )  24 {  25 // do something  26 }  27 } |

[1~14行目]は、コールバックサービスハンドラーの実装部分である。「1.3.7.1 シングルトンハンドラー方式」と異なり、このクラスにはメンバ変数([3行目])とメンバメソッド([10~13行目])が定義されている。[16~27行目]はコールバックサービスハンドラーファクトリの実装部分で、createCallbackHandler([18行目])でコールバックハンドラーを生成して返している。毎回コールバックサービスが呼び出されるたびにコールバックサービスハンドラーが別途生成されるため、ハンドラー内でメンバ変数を自由に使用できるというメリットがある。

#### ハンドラープール方式

生成または初期化に長い時間がかかるオブジェクト、例えばデータベースコネクションやリモートアプリケーションサーバとのTCP/IPセッションなどをコールバックサービスハンドラーのメンバとして使用する必要がある場合がある。この場合、「1.3.7.2 コールバックサービスごとに新しいコールバックハンドラーオブジェクトを生成する方式」のように毎回ハンドラーオブジェクトを生成するよりは、一度生成されたハンドラーオブジェクトを再利用するほうが性能およびリソースの活用度を向上できる。次の表は、このような方式の例を示す。

[表 41 ハンドラープール方式の例]

|  |
| --- |
| 1 public class CallbackHandlerImpl implements CallbackHandler  2 {  3 private Socket sock;  4  5 public CallbackHandlerImpl()  6 {  7 sock = new Socket( ... );  8 }  9  10 public void doCallback( CallbackReq req, CallbackRes res, Object res ) throws CallbackException  11 {  12 // use the Socket object  13 }  14 }  15  16 public class CallbackHandlerFactoryImpl implements CallbackHandlerFactory  17 {  18 private ObjectPool<CallbackHandlerImpl> pool  19 = new ObjectPool<CallbackHandlerImpl> ();  20  21 public CallbackHandler createCallbackHandler()  22 {  23 return pool.borrowObject();  24 }  25  26 public void destroyCallbackHandler( CallbackHandler handler )  27 {  28 pool.returnObject( (CallbackHandlerImpl)handler );  29 }  30 } |

[1~9行目]はコールバックサービスハンドラーの実装部分で、メンバ変数としてSocketオブジェクトを使用しコンストラクタでソケットを生成して確立している。上記ハンドラーオブジェクトを「1.3.7.2 コールバックサービスごとに新しいコールバックハンドラーオブジェクトを生成する方式」の方式でコールバックサービスごとに生成した場合、ソケットの確立による性能低下が想定される。これを防止するために、コールバックサービスハンドラーファクトリ([16~30行目])ではプールを使用している。createCallbackHandlerではプールからハンドラーオブジェクトを取得して返す。ハンドラーオブジェクトの使用が終了後に呼び出されるdestroyCallbackHandlerでは、これを再びプールに返す。プールを使用することで、ソケットの生成頻度を最小化できる。もちろん、実際のコードは例外の処理やプールが枯渇した時の処理、プールの実装など、より複雑に実装できるが、大枠は[表 44 データ型の再定義]と類似している。

### com.innoexpert.rulesclient.CallbackException

各種のコールバックメソッドでエラーが発生した時に投げられる例外クラスである。次の表のように、java.lang.

Exceptionを継承して4つのコンストラクタが定義されている。

[表 42 com.innoexpert.rulesclient.CallbackException]

|  |
| --- |
| 1 package com.innoexpert.rulesclient;  2  3 public class CallbackException extends Exception  4 {  5 public CallbackException()  6 public CallbackException( String s )  7 public CallbackException( String message, Throwable cause )  8 public CallbackException( Throwable cause )  9 } |

# C/C++ API

C/C++ APIを利用してルールサービスを呼び出すCまたはC++アプリケーションを作成し実行できる。このルールサーバがJavaで作成されているため、TCP/IPを利用したリモートルールサーバ呼び出しのみがサポートされ、アプリケーションにルールサーバを埋め込むローカルルールサーバ構成はサポートされない。

C/C++ APIライブラリは、次のプラットフォームをサポートする。

* UNIX系列(IBM AIX、HP-UX、Linuxなど)
* Microsoft Windows 2000 Server/XP以上(32/64ビット)

## ルールサービス呼び出しAPI

### APIの呼び出し順序

ルールサービスの呼び出しは、次の順序で行われる。

1. ルールインタフェースクラスタハンドルの生成および初期化
2. ルールトランザクションハンドルの生成および入力値の設定
3. ルールサービスの呼び出し
4. ルールトランザクションハンドルから結果を抽出またはエラー処理後ルールトランザクションハンドルを解放
5. 繰り返してルールを呼び出すためには、2~4のプロセスを繰り返す
6. ルールインタフェースクラスタハンドルの解放

次の表は、ルールサービスを呼び出す簡単なアプリケーションコードの例である。このコード例を基に、ルールサービスの呼び出し順序を説明する。

[表 43 ルールサービスを呼び出す簡単なCアプリケーション]

|  |
| --- |
| 1 #include <stdio.h>  2 #include "ircapi.h"  3  4 static char const\* lpszProperty = "MAIN.TYPE=TCP;RULECLIENT1.ADDRESS=127.0.0.1; RULECLIENT1.PORT=25800;";  5  6 int main( int argc, char\* argv[] )  7 {  8 IRCHANDLE hCluster;  9  10 hCluster = IRCAllocHandle( HTYPE\_CLUSTER );  11  12 if( !hCluster )  13 {  14 printf( "Cluster handle allocation failure.\n" );  15 return 0;  16 }  17  18 if( !IRCInitialize( hCluster, lpszProperty ) )  19 {  20 char lpszError[1024];  21 IRCGetLastError( hClient, lpszError, sizeof(lpszError) );  22 IRCFreeHandle( hCluster );  23 printf( "ERROR: %s\n", lpszError );  24 return 0;  25 }  26  27 do  28 {  29 IRCHANDLE hRule;  30 IRCDESCRIPTOR dResult;  31  32 hRule = IRCAllocHandle( HTYPE\_RULE );  33 IRCRuleSet( hRule, "COMMISSION", "2013-07-01" );  34 IRCRuleAddStrItem( hRule, "TRDTYP", "Lease" );  35 IRCRuleAddNumItem( hRule, "AMT", 250000000.0 );  36  37 if( !IRCExecute( hCluster, hRule, rci\_codetype\_alias, 0, IRCFALSE ) )  38 {  39 char lpszError[1024];  40 IRCGetLastError( hRule, lpszError, sizeof(lpszError) );  41 IRCFreeHandle( hRule );  42 IRCFreeHandle( hCluster );  43 printf( "ERROR: %s\n", lpszError );  44 return 8;  45 }  46  47 dResult = IRCRuleGetResult( hRule );  48  49 printf( "Rate: %s\n", IRCRuleResultGetStrValue( dResult, 1, 1 ) );  50 printf( "Amount: %.0lf\n", IRCRuleResultGetNumValue( dResult, 2, 1 ) );  51  52 IRCFreeHandle( hRule );  53 }  54 while( 0 );  55  56 IRCFreeHandle( hCluster );  57  58 return 0;  59 } |

この例は、Commissionというルールを呼び出すCアプリケーションである。[6行目]は、アプリケーションのエントリーポイント(Entry Point)であるmain関数である。[10~16行目]は、ルールインタフェースクラスタハンドルを生成するプロセスである。クライアントハンドルとも呼ばれるルールインタフェースクラスタハンドルは、ルールインタフェースクラスタ構造体を示すvoid\*である。このハンドルを生成することで、新しいルールインタフェースクラスタを生成する。[18~25行目]は、クラスタを初期化するプロセスである。[4行目]に指定されたクラスタの設定が初期化ルーチンに引き渡され([18行目])、設定に基づいてクラスタが構成される。クラスタハンドルを生成して初期化するプロセスは、1つのアプリケーションで1度のみ実行すればよい。一度生成されたクラスタを利用して繰り返し、または同時に複数のスレッドを利用してルールサービスを呼び出すことができる。

[27~53行目]は、ルールサービスを呼び出す部分である。クラスタハンドルが一度初期化されると何度も繰り返してルールサービスの呼び出しができることを示すために、[54行目]にwhile文を入れた。[32~35行目]は、ルールトランザクションハンドルを生成して入力値を設定する部分である。ルールトランザクションハンドルは1つのトランザクションと関連するデータ、例えば呼び出すルールコードや入力値、結果などを保存するための空間を示すvoid\*である。

[37~45行目]は、ルールサービスを呼び出す部分である。[47~52行目]は、ルールサービスの呼び出し結果を使用してルールトランザクションハンドルを解放する部分である。

アプリケーションは、ルールサービスの呼び出しをそれ以上する必要がない場合、一般的にアプリケーションがシャットダウンされる時に[56行目]のようにルールインタフェースクラスタハンドルを解放する。

APIと関連する定数および関数などは、ircapi.hに宣言されている。したがって、[2行目]ではこのヘッダファイルをインクルードしている。

### ライブラリとアプリケーションビルド

アプリケーションをコンパイルするためには、-Iオプションを利用してircapi.hが存在するディレクトリをインクルードパスに含める必要がある。アプリケーションをビルドするためには、次の共有ライブラリ(Shared)をリンクする必要がある。

* libircapi.so

共有ライブラリの拡張子はプラットフォームによって異なることがある。例えば、IBM AIXは.a、HP-UXは.sl、UBUNTUは.soなどになることがある。この他に、プラットフォームによってシステムライブラリのリンクが必要なこともある。

アプリケーションを実行するためには、上のライブラリが位置するパスをライブラリパス環境変数(LD\_LIBRARY\_PATH、SHLIB\_PATH、LIBPATHなど)に追加する必要がある。

### ハンドルとディスクリプタ

「2.1.1 APIの呼び出し順序」の例のアプリケーションで、ハンドル(Handle)の概念を紹介した。ルールインタフェースクラスタやルールトランザクションデータ構造は、多くのデータ型を使用して複雑に定義されている。これらのデータ構造は、APIのバージョンが上がるに伴い変更されることもある。このデータ構造をアプリケーションに公開する場合、アプリケーションと関係のないデータ型について宣言されている多くのヘッダファイルをインクルードする必要もあり、APIのバージョンが上がるに伴いアプリケーションの再ビルドが必要になる可能性がある。こうした複雑さをなくすために、アプリケーションではこれらのデータ構造をvoid\*で参照した。アプリケーションがこれらのデータ(ルールインタフェースクラスタやルールトランザクション)を必要とする時には、適切なハンドルの類型をパラメータとして引き渡し、動的に割り当てて使用を終えた後にハンドルを解放する。

ハンドルと似た概念として、ディスクリプタ(Descriptor)がある。ディスクリプタもデータ構造の詳細な形式をアプリケーションに隠すための用途として使用されるが、ハンドルと異なり割り当てまたは解放する作業がない。ディスクリプタはハンドルによって参照される一部のメモリ領域をvoid\*で管理するため、ハンドルが解放されるとハンドルが参照するメモリも解放される。したがってディスクリプタのメモリも解放される。

ハンドルとディスクリプタのどちらもデータ型がvoid\*であるため、これを区分するために次の表のようにハンドルはIRCHANDLE、ディスクリプタはIRCDESCRIPTORとして形式を再定義した。

[表 44 データ型の再定義]

|  |
| --- |
| 1 typedef void\* IRCHANDLE;  2 typedef void\* IRCDESCRIPTOR; |

### 定数と列挙型(enum)

ircpapi.hには、データ型を表現するために次の表の定数が定義されている。

[表 45 データ型定数]

|  |
| --- |
| 1 #define RULES\_TYPE\_NUMBER 1  2 #define RULES\_TYPE\_STRING 2  3 #define RULES\_TYPE\_BOOLEAN 3 |

RULES\_TYPE\_NUMBER、RULES\_TYPE\_STRING、RULES\_TYPE\_BOOLEANは、それぞれ数値型、文字型、論理型を表現する定数である。これらの定数は、ルール呼び出し結果のカラム形式を照会する時に使用される。

「2.1.3 ハンドルとディスクリプタ」で、ハンドルを割り当てる時にハンドルの形式を指定すると説明した。ハンドルの類型には、ルールインタフェースクラスタとルールトランザクションがある。この類型を指定できるように、次の表のような定数が指定されている。

[表 46 ハンドル類型定数]

|  |
| --- |
| 1 #define HTYPE\_CLUSTER 0x01  2 #define HTYPE\_RULE 0x10 |

HTYPE\_CLUSTERはルールインタフェースプールハンドルを、HTYPE\_RULEはルールトランザクションハンドルを表す。この他に、次の表のように、利便性を考慮した定数およびデータ型が定義されている。

[表 47 便利な定数]

|  |
| --- |
| 1 typedef int IRCBOOL;  2 #define IRCTRUE (int)(1)  3 #define IRCFALSE (int)(0) |

intをIRCBOOLとして再定義し、論理定数を表現するIRCTRUEとIRCFALSEが定義されている。IRCTRUEは真を、IRCFALSEは偽を表す。

ルールサービスを呼び出すためには、呼び出すルールのコードや入力項目のコードなどが指定される必要がある。コードを指定する方式としては、ID、名前、エイリアスの体系があり、これをコード体系という。コード体系に関する詳細内容については、「1.2.3 コード体系」を参照のこと。ルールの呼び出しに使用するコード体系を表現する列挙型が、次の表のように定義されている。

[表 48 コード体系の列挙型]

|  |
| --- |
| 1 typedef enum \_rci\_codetype  2 {  3 rci\_codetype\_id = RULES\_CODETYPE\_ID,  4 rci\_codetype\_name = RULES\_CODETYPE\_NAME,  5 rci\_codetype\_alias = RULES\_CODETYPE\_ALIAS  6 } rci\_codetype; |

rci\_codetype\_idは使用されたコード体系がIDであることを、rci\_codetype\_nameとrci\_codetype\_aliasはそれぞれ名前とエイリアスであることを表す。これらの値はルールサービスを呼び出す時にパラメータとして引き渡され、1つのルールトランザクションに使用されたルールコードや項目コードはすべて同一コード体系を使用する。つまり、ルールのコードをエイリアスで指定した場合は項目のコードをエイリアスで指定する必要があり、ルールサービスを呼び出す時にrci\_codetype\_aliasを指定する必要がある。指定されたコード体系は、ルール呼び出し結果のカラム名にも影響を与える。コード体系がIDまたは名前である場合、ルール呼び出し結果のカラム名にはルールリターン項目の名前が使用され、コード体系がエイリアスである場合はカラム名にルールリターン項目のエイリアスが使用される。

### ハンドル関連の関数

次の表は、ハンドルと関連する関数である。

[表 49 ハンドル関連の関数]

|  |
| --- |
| 1 IRCHANDLE IRC\_API IRCAllocHandle( int nType );  2 void IRC\_API IRCFreeHandle( IRCHANDLE hHandle );  3 IRCBOOL IRC\_API IRCInitialize( IRCHANDLE hCluster, char const\* lpszProperties );  4 IRCBOOL IRC\_API IRCFinalize( IRCHANDLE hCluster );  5 void IRC\_API IRCGetLastError( IRCHANDLE handle, char\* lpMsg, int nMsgLen ); |

IRCHANDLE IRC\_API IRCAllocHandle( int nType )

nTypeに指定された類型のハンドルをメモリに割り当てる。メモリの割り当てに成功した場合は0ではない値を、失敗した場合は0を返す。nTypeには、[表 46 ハンドル類型定数]の定数を定義する。

void IRC\_API IRCFreeHandle( IRCHANDLE hHandle )

hHandleで与えられたハンドルのメモリを解放する。hHandleはIRCAllocHandleを利用して正常に割り当てられたハンドル値である必要がある。

IRCBOOL IRC\_API IRCInitialize( IRCHANDLE hCluster, char const\* lpszProperties )

hClusterパラメータで与えられたハンドルが示すルールインタフェースクラスタを初期化する。ルールインタフェースクラスタは、ハンドル割り当て後に初期化する必要がある。クラスタの属性は、lpszPropertiesパラメータとして与えられる文字列で指定できる。この文字列を属性文字列といい、「2.1.8 ルールインタフェースクラスタの属性文字列」で詳しく説明する。初期化に成功した場合はIRCTRUEを、そうでない場合はIRCFALSEを返す。失敗した場合は、IRCGetLastError関数を利用してエラーメッセージを取得できる。

IRCBOOL IRC\_API IRCFinalize( IRCHANDLE hCluster )

hClusterパラメータで与えられたハンドルが示すルールインタフェースクラスタを消滅させる。ルールインタフェースクラスタは、それ以上使用する必要がない場合、この関数を利用してリソースを解放する必要がある。解放に成功した場合はIRCTRUEを、そうでない場合はIRCFALSEを返す。失敗した場合は、IRCGetLastError関数を利用してエラーメッセージを取得できる。

void IRC\_API IRCGetLastError( IRCHANDLE handle, char\* lpMsg, int nMsgLen )

ハンドルを利用した関数の呼び出しでエラーが発生した場合、該当ハンドルと関係する最後のエラーメッセージを取得する。hContextは、エラーが発生した関数に引き渡したハンドルである。lpMsgはエラーメッセージを格納するバッファーのポインタで、nMsgLenはこのバッファーの有効な長さである。ルールの呼び出しのように、ルールインタフェースクラスタのハンドルとルールトランザクションハンドルがどちらも引き渡された関数でエラーが発生した場合、hContextにはルールトランザクションハンドルを引き渡す。[表 43 ルールサービスを呼び出す簡単なCアプリケーション]の[37行目から40行目]を参照のこと。

### ルール入力設定関数

ルールサービス呼び出しの入力値を設定する関数は、次の表のとおりである。

[表 50 項目値設定関数]

|  |
| --- |
| 1 IRCBOOL IRC\_API IRCRuleSet( IRCHANDLE hRule, char const\* lpszRule, char const\* lpszDate, rci\_codetype codetype, IRCBOOL wantTrace );  2 IRCDESCRIPTOR IRC\_API IRCRuleAddNumItem( IRCHANDLE hRule, char const\* lpszItem, double val );  3 IRCDESCRIPTOR IRC\_API IRCRuleAddStrItem( IRCHANDLE hRule, char const\* lpszItem, char const\* val );  4 void IRC\_API IRCRuleItemAddNumValue( IRCDESCRIPTOR dItem, double val );  5 void IRC\_API IRCRuleItemAddStrValue( IRCDESCRIPTOR dItem, char const\* val );  6 IRCDESCRIPTOR IRC\_API IRCRuleFindItem( IRCHANDLE hRule, char const\* lpszItem ); |

IRCBOOL IRC\_API IRCRuleSet( IRCHANDLE hRule, char const\* lpszRule, char const\* lpszDate, rci\_codetype codetype, IRCBOOL wantTrace )

トランザクションハンドルに、呼び出すルールのコードとルール呼び出し基準日を設定する。hRuleパラメータはルールトランザクションハンドル、lpszRuleパラメータは呼び出すルールコード、lpszDateはルール実行基準日である。lpszDateはnullが可能で、この場合のルール実行基準日はリモートルールエンジンのシステム日付となる。Codetypeは呼び出すルールコードのコード体系で、[表 48 コード体系の列挙型]の値の中の1つである。wantTraceパラメータはIRCFALSEに設定する。hRuleが有効なトランザクションハンドルでない場合やlpszRuleが0である場合は、この関数はIRCFALSEを返す。そうでない場合は、IRCTRUEを返す。

呼び出すルールコードのコード体系としてIDや名前またはエイリアスを使用できるが、IRCExecute関数に使用されたコード体系を指定する必要がある。

IRCDESCRIPTOR IRC\_API IRCRuleAddNumItem( IRCHANDLE hRule, char const\* lpszItem, double val )

トランザクションハンドルに数値型の項目を追加する。hRuleパラメータはルールトランザクションハンドル、lpszItemパラメータは追加する項目のコード、valは項目に割り当てる数値である。関数は、トランザクションハンドルに追加された項目を示すディスクリプタを返す。このディスクリプタとIRCRuleItemAddNumValue関数を利用して数値配列に要素を追加できる。lpszItemパラメータに指定するコードは、項目のIDや名前またはエイリアスを使用でき、IRCExecute関数に指定したパラメータによってどれを使えるかが決まる。

IRCDESCRIPTOR IRC\_API IRCRuleAddStrItem( IRCHANDLE hRule, char const\* lpszItem, char const\* val )

トランザクションハンドルに文字型の項目を追加する。hRuleパラメータはルールトランザクションハンドル、lpszItemパラメータは追加する項目のコード、valは項目に割り当てる文字値である。関数は、トランザクションハンドルに追加された項目を示すディスクリプタを返す。このディスクリプタとIRCRuleItemAddStrValue関数を利用して、文字値配列に要素を追加できる。lpszItemパラメータに指定するコードは、項目のIDや名前またはエイリアスを使用でき、IRCExecute関数に指定したパラメータによってどれを使えるかが決まる。

void IRC\_API IRCRuleItemAddNumValue( IRCDESCRIPTOR dItem, double val )

トランザクションハンドルに存在する数値型の項目の値の配列に数値を追加する。dItemパラメータはトランザクションハンドルに存在する項目を示すディスクリプタでIRCRuleAddNumItemによって返された値である。valは追加する数値である。ディスクリプタが示す項目が数値型の項目でない場合は、C++例外(Exception)が投げられる。同一ディスクリプタにこの関数を繰り返し呼び出して複数の値を追加できる。

void IRC\_API IRCRuleItemAddStrValue( IRCDESCRIPTOR dItem, char const\* val )

トランザクションハンドルに存在する文字型の項目の値の配列に文字値を追加する。dItemパラメータはトランザクションハンドルに存在する項目を示すディスクリプタでIRCRuleAddStrItemによって返された値である。valは追加する文字値である。ディスクリプタが示す項目が文字型の項目でない場合は、C++例外が投げられる。同一ディスクリプタにこの関数を繰り返し呼び出して複数の値を追加できる。

IRCDESCRIPTOR IRC\_API IRCRuleFindItem( IRCHANDLE hRule, char const\* lpszItem );

トランザクションハンドルに存在する項目を探して返す。hRuleパラメータはルールトランザクションハンドル、lpszItemパラメータは検索対象の項目コードである。項目が存在する場合、該当項目を返し、存在しない場合0を返す。項目のコードはID、名称またはエイリアスを使用でき、使用したコード体系はIRCExecute関数に指定される。

### ルールサービス呼び出しおよびルール結果照会関数

ルールサービスを呼び出してその結果を使用する関数は、次の表のとおりである。

[表 51 ルールサービス呼び出しおよびルール結果照会関数]

|  |
| --- |
| 1 IRCBOOL IRC\_API IRCExecute( IRCHANDLE hCluster, IRCHANDLE hRule, void\* ctx );  2 IRC\_API char const\* IRCGetErrorMessage( IRCHANDLE hRule );  3 IRC\_API char const\* IRCGetErrorCode( IRCHANDLE hRule );  4 IRC\_API int IRCGetAdditionalInfoCount( IRCHANDLE hRule );  5 IRC\_API char const\* IRCGetAdditionalInfoKey( IRCHANDLE hRule, int index );  6 IRC\_API char const\* IRCGetAdditionalInfoValue( IRCHANDLE hRule, int index );  7 IRC\_API char const\* IRCGetAdditionalInfoValueFromKey( IRCHANDLE hRule, char const \*key );  8 IRC\_API char const\* IRCGetAdditionalRuleStacks( IRCHANDLE hRule );  9 IRCDESCRIPTOR IRC\_API IRCRuleGetResult( IRCHANDLE hRule );  10 IRC\_API char const\* IRCRuleGetTrace( IRCDESCRIPTOR dResult );  11 IRC\_API int IRCRuleResultGetColumnCount( IRCDESCRIPTOR dResult );  12 IRC\_API int IRCRuleResultGetRowCount( IRCDESCRIPTOR dResult );  13 IRC\_API IRCBOOL IRCRuleResultGetInfo( IRCDESCRIPTOR dResult, int\* pColSize, int\* pRowSize );  14 IRC\_API short IRCRuleResultGetType( IRCDESCRIPTOR dResult, int nCol );  15 IRC\_API char const\* IRCRuleResultGetColumnName( IRCDESCRIPTOR dResult, int nCol );  16 IRC\_API double IRCRuleResultGetNumValue( IRCDESCRIPTOR dResult, int nCol, int nRow );  17 IRC\_API char const\* IRCRuleResultGetStrValue( IRCDESCRIPTOR dResult, int nCol, int nRow );  18 IRCBOOL IRC\_API IRCRuleResultGetBoolValue( IRCDESCRIPTOR dResult, int nCol, int nRow );  19 IRC\_API double IRCRuleResultGetNumValueByName( IRCDESCRIPTOR dResult, char const\* lpszCol, int nRow );  20 IRC\_API char const\* IRCRuleResultGetStrValueByName( IRCDESCRIPTOR dResult, char const\* lpszCol, int nRow );  21 IRC\_API IRCBOOL IRCRuleResultGetBoolValueByName( IRCDESCRIPTOR dResult, char const\* lpszCol, int nRow ); |

次は、関数についての説明である。

IRCBOOL IRCExecute(IRCHANDLE hCluster, IRCHANDLE hRule, void\* ctx )

ルールサービスを呼び出す関数である。hClientパラメータはルールサービスの呼び出しに使用するルールインタフェースクラスタハンドルであり、hRuleは入力値が設定されたルールトランザクションハンドルである。pCtxはコールバックコンテキストオブジェクトであり、詳細は「2.2 コールバックAPI」で説明する。

ルールサービスの呼び出しが成功した場合IRCTRUEを返し、IRCRuleGetResultを利用してルール結果に対するディスクリプタが取得できる。ルールサービスの呼び出しが失敗した場合IRCFALSEを返し、IRCGetLastError関数、IRCGetErrorMessage関数、IRCGetErrorCode関数を利用してエラーメッセージとエラーコードを取得できる。この時、パラメータとしてhRuleを引き渡す必要がある。

IRC\_API char const\* IRCGetErrorMessage( IRCHANDLE hRule )

トランザクションハンドルを利用し、エラーメッセージを返す。hRuleはルールサービス呼び出しに使用されたトランザクションハンドルである。

IRC\_API char const\* IRCGetErrorCode( IRCHANDLE hRule )

トランザクションハンドルを利用し、エラーコードを返す。hRuleはルールサービス呼び出しに使用されたトランザクションハンドルである。

IRC\_API int IRCGetAdditionalInfoCount( IRCHANDLE hRule )

ルールサービスの呼び出しが失敗した場合、失敗した原因に対する詳細情報の数を返す。hRuleはルールサービス呼び出しに使用されたトランザクションハンドルである。

IRC\_API char const\* IRCGetAdditionalInfoKey( IRCHANDLE hRule, int index )

ルールサービスの呼び出しが失敗した場合、失敗した原因に対する詳細情報のうち、indexに該当するキー値を返す。hRuleはルールサービス呼び出しに使用されたトランザクションハンドルである。

IRC\_API char const\* IRCGetAdditionalInfoValue( IRCHANDLE hRule, int index )

ルールサービスの呼び出しに失敗した場合、失敗した原因に対する詳細情報のうち、indexに該当する値を返す。hRuleはルールサービスの呼び出しに使用されたトランザクションハンドルである。

IRC\_API char const\* IRCGetAdditionalInfoValueFromKey( IRCHANDLE hRule, char const \*key )

ルールサービスの呼び出しに失敗した場合、失敗した原因に対する詳細情報のうち、キー値に該当する値を返す。hRuleはルールサービスの呼び出しに使用されたトランザクションハンドルである。

IRC\_API char const\* IRCGetAdditionalRuleStacks( IRCHANDLE hRule )

ルールサービスの呼び出しに失敗した場合、失敗した原因に対する詳細情報内のRULESTACKSの値を返す。RULESTACKSはルールサービスの呼び出し時に呼び出しされたルールの実行順序である。hRuleはルールサービス呼び出しに使用されたトランザクションハンドルである。

IRCDESCRIPTOR IRCRuleGetResult( IRCHANDLE hRule );

トランザクションハンドルを利用し、呼び出した最後のルールサービスの結果に対するディスクリプタを取得する。hRuleは、ルールサービスの呼び出しに使用されたトランザクションハンドルである。返されるディスクリプタは、hRuleが参照しているメモリ領域の一部を参照している。したがって、ルール結果の照会を終える前にhRuleハンドルを解放してはならない。ルールサービスを呼び出さずにこのメソッドを呼び出した場合は、nullが返される。

IRC\_API char const\* IRCRuleGetTrace( IRCDESCRIPTOR dResult )

ルールサービス結果でルール実行経路トレースを返す。ルール実行経路トレース作成はIRCRuleSet関数の5番目のパラメータの値で設定する。ルール実行経路トレースを使用する場合、ルール実行性能が多少減少し、出力オブジェクトのサイズが多少増加できる。したがって、ルール実行経路トレースはエラーの解決の目的のみで使用することを推奨する。

IRC\_API int IRCRuleResultGetColumnCount( IRCDESCRIPTOR dResult )

ルールサービス結果のカラム数を取得する。dResultパラメータはルールサービス結果のディスクリプタである。

IRC\_API int IRCRuleResultGetRowCount( IRCDESCRIPTOR dResult )

ルールサービス結果の行数を取得する。dResultパラメータはルールサービス結果のディスクリプタである。

IRCBOOL IRCRuleResultGetInfo( IRCDESCRIPTOR dResult, int\* pColSize, int\* pRowSize )

ルールサービス結果のカラム数と行数を取得する。dResultパラメータはルールサービス結果のディスクリプタである。カラム数と行数はそれぞれpColSizeとpRowSizeが示すint領域に設定される。dResultが正しいルール結果ディスクリプタでない場合はIRCFALSEを返し、そうでない場合IRCTRUEを返す。

short IRCRuleResultGetType( IRCDESCRIPTOR dResult, int nCol )

dResultパラメータが示すルール結果ディスクリプタからnCol番目のカラムのデータ型を返す。返されるデータ型は、[表 45 データ型定数]のとおりである。nColは、1ベース(1-base)インデックスである。つまり、最初のカラムのデータ型を取得する場合、nColには1を引き渡す必要がある。

char const\* IRCRuleResultGetName( IRCDESCRIPTOR dResult, int nCol )

dResultパラメータが示すルール結果ディスクリプタからnCol番目のカラム名を返す。返されるカラム名は、RCExecuteに引き渡されたコード体系によって決定される。コード体系がIDまたは名前で指定された場合、カラム名はルールのリターン項目の名前である。コード体系がエイリアスで指定された場合、カラム名はルールのリターン項目のエイリアスである。

double IRCRuleResultGetNumValue( IRCDESCRIPTOR dResult, int nCol, int nRow )

ルールサービス結果のnCol番目のカラムのnRow番目の行の数値を返す。dResultパラメータはルール結果ディスクリプタで、nColとnRowはそれぞれカラムと行の番号で1ベースインデックスである。カラムのデータ型が数値型である場合のみ、呼び出す必要がある。

char const\* IRCRuleResultGetStrValue( IRCDESCRIPTOR dResult, int nCol, int nRow )

ルールサービス結果のnCol番目のカラムのnRow番目の行の文字値を返す。dResultパラメータはルール結果ディスクリプタで、nColとnRowはそれぞれカラムと行の番号で1ベースインデックスである。カラムのデータ型が文字型である場合にのみ、呼び出す必要がある。

IRCBOOL IRCRuleResultGetBoolValue( IRCDESCRIPTOR dResult, int nCol, int nRow )

ルールサービス結果のnCol番目のカラムのnRow番目の行の論理値を返す。dResultパラメータはルール結果ディスクリプタで、nColとnRowはそれぞれカラムと行の番号で1ベースインデックスである。カラムのデータ型が論理型である場合にのみ、呼び出す必要がある。

double IRCRuleResultGetNumValueByName( IRCDESCRIPTOR dResult, const char \*lpszCol, int nRow )

ルールサービス結果から特定のカラムのnRow番目の行の数値を返す。dResultパラメータはルール結果ディスクリプタで、lpszColは値を照会対象のカラム名である。結果カラム名は、ルールサービスの呼び出しに指定されたコード体系によってルールのリターン項目名、IDまたはエイリアスを指定する必要がある。データ型が数値型の場合のみ、呼び出す必要がある。

char const\* IRCRuleResultGetStrValueByName( IRCDESCRIPTOR dResult, const char \*lpszCol, int nRow )

ルールサービス結果から特定のカラムのnRow番目の行の文字値を返す。dResultパラメータはルール結果ディスクリプタで、lpszColは値を照会対象のカラム名である。結果カラム名は、ルールサービスの呼び出しに指定されたコード体系によってルールのリターン項目名、IDまたはエイリアスを指定する必要がある。データ型が文字型の場合のみ、呼び出す必要がある。

IRCBOOL IRCRuleResultGetBoolValueByName( IRCDESCRIPTOR dResult, const char \*lpszCol, int nRow )

ルールサービス結果から特定のカラムのnRow番目の行の論理値を返す。dResultパラメータはルール結果ディスクリプタで、lpszColは値を照会対象のカラム名である。結果カラム名は、ルールサービスの呼び出しに指定されたコード体系によってルールのリターン項目名、IDまたはエイリアスを指定する必要がある。データ型が論理型の場合のみ、呼び出す必要がある。

### ルールインタフェースクラスタの属性文字列

IRCInitializeにはルールインタフェースクラスタの構成設定情報を含む属性文字列が引き渡される。属性文字列は、次の表のような形式を持つ。

[表 52 属性文字列の形式]

|  |
| --- |
| 属性名=属性値;[属性名=属性値;]\* |

属性は、ルールインタフェースクラスタ関連属性とルールインタフェースプール関連属性に区分できる。属性は必須属性と任意属性に区分され、任意属性は値を指定しない場合、デフォルト値に設定される。任意属性には、「(オプション)」と表記している。ルールインタフェースクラスタおよびルールインタフェースプールの動作方式、それぞれの属性に関する詳細事項については、「InnoRules Installation and Operation Guide 3. Rule System Concepts」を参照のこと。

次は、ルールインタフェースクラスタに関連する属性である。

MAIN.TYPE

ルールエンジンの構成方式である。C/C++環境では現在TCP/IPを利用したリモート呼び出しのみがサポートされるため、常にTCPと設定する必要がある。

MAIN.POOLTYPE

(オプション)1つのシステムには、複数のインスタンスのリモートルールエンジンが構成できる。ルールインタフェースプールは、1つのルールエンジンに接続されている多数のTCP/IPセッションを管理する。1つのルールインタフェースクラスタは、複数のルールインタフェースプールを管理できる。これらのプールは優先順位を指定してフェイルオーバー(Fail Over)モード、または順序を指定して負荷分散(Load Balancing)モードで構成できる。MAIN.POOLTYPEがFAILOVERである場合はフェイルオーバーモード、LOADBALANCEである場合は負荷分散モードで構成される。指定しない場合は、LOADBALANCEが指定される。

MAIN.SVRCNT

(オプション)ルールインタフェースクラスタを構成するルールインタフェースプールの数を指定する。1以上を指定する。指定しない場合は、1が設定される。

MAIN.WHENEXHAUSTEDACTION

(オプション)アプリケーションがIRCExecuteメソッドを呼び出した際には、クラスタで管理するすべてのルールインタフェースが使用中でルールサービスを呼び出すために使用するインタフェースがない場合の動作方式を指定する。BLOCKを指定すると、他のアプリケーションがルールインタフェースを返すまで待機する。ここでのアプリケーションとは、同一プロセスでルールサービスを呼び出すスレッドを意味する。FAILを指定すると、IRCExecuteはIRCFALSEを返す。デフォルト値はBLOCKである。

MAIN.CHECKCONNECTION

(オプション)IRCExecuteでクラスタからルールインタフェースを取得する際に接続が有効かを確認する。接続が無効の場合は、有効な接続を取得するまで再試行する。YESを設定した場合は有効性を確認し、この時リモートルールエンジンの一部に障害が発生していても使用可能なルールエンジンが存在する場合は、ルールサービスは常に呼び出しが可能である。なお、NOを設定した場合は有効性を確認せず、ルールエンジンが動作していない、または障害がある場合はルールの呼び出しに失敗することもある。デフォルト値はYESである。

MAIN.AUTORECOVERY

(オプション)ルールインタフェースクラスタの自動復元について設定する。YESを設定した場合は自動復元を行い、NOを設定した場合は自動復元されない。デフォルト値はYESである。

MAIN.RECOVERYCHECKINTERVAL

(オプション)ルールインタフェースクラスタを自動復元するために障害が発生したルールインタフェースプールの復元有無をチェックする周期で、秒単位で設定する。デフォルト値は10である。

次は、ルールインタフェースプールに関連する属性である。属性の名前にnで表記された部分はプールのインデックスである。このインデックスはフェイルオーバーモードのクラスタではプールの優先順位として、負荷分散モードのクラスタではラウンドロビンの順序として使用される。インデックスは1ベースである。例えば、最初のルールインタフェースプールの最大有効ルールインタフェースの数を設定する属性の名前はRULECLIENT1.MAXACTIVEである。

RULECLIENTn.MAXACTIVE

(オプション)このプールを利用して同時にアプリケーションが使用できるルールインタフェースの最大数を定義する。つまり、同時生成可能なリモートルールエンジンへのTCP/IPの最大数である。デフォルト値は100である。

RULECLIENTn.MAXIDLE

(オプション)アプリケーションがルールインタフェースを使用した後には、再びプールに保管される。この属性はプールに保管するルールインタフェースの数、つまりプールに保管するTCP/IPセッションの最大数である。デフォルト値は100である。

RULECLIENTn.ADDRESS

(オプション)リモートルールエンジンのTCP/IPアドレスである。デフォルト値はlocalhostである。

RULECLIENTn.PORT

(オプション)リモートルールエンジンのTCP/IPポートである。デフォルト値は25800である。

RULECLIENTn.TIMEOUT

(オプション)リモートルールエンジンにルールサービス要求を引き渡した後、この属性に指定した時間内に応答がない場合はタイムアウトとみなしてIRCExecute関数はIRCFALSEを返す。時間の単位はミリ秒で、0を指定するとタイムアウトを適用しない。デフォルト値は30000である。

RULECLIENTn.CONNECTIONTIMEOUT

(オプション)リモートルールエンジンに新しいTCP/IPセッションを確立する際、この属性に指定された時間内に接続されない場合はタイムアウトとみなして関連関数はエラーを発生させる。時間の単位はミリ秒で、0を指定するとタイムアウトを適用しない。デフォルト値は3000である。

次のようなシステム構成について想定する。

1. 2つのリモートルールエンジンが構成されており、これらはserver1、server2というホストの25800ポートで動作する。
2. ルールエンジンはフェイルオーバー方式で運用され、server1で動作するルールエンジンがアクティブ、server2で動作するルールエンジンがスタンバイである。
3. アプリケーションサーバはマルチスレッディングで同時処理しており、同時接続は最大100で、普段は20程度である。
4. リモートルールエンジンに障害が発生した場合においても、極力アプリケーションには影響がないようにする。
5. アプリケーションは、最大30秒以内に成功または失敗の応答をクライアントに返す。

1と2の条件に基づいて、属性文字列は次のように構成される。

|  |
| --- |
| MAIN.TYPE=TCP;MAIN.SVRCNT=2;MAIN.POOLTYPE=FAILOVER;RULECLIENT1.ADDRESS=server1;RULECLIENT1.PORT=25800;RULECLIENT2.ADDRESS=server2;RULECLIENT2.PORT=25800; |

3の条件に基づいて、次のように修正される。

|  |
| --- |
| MAIN.TYPE=TCP;MAIN.SVRCNT=2;MAIN.POOLTYPE=FAILOVER;RULECLIENT1.ADDRESS=server1;RULECLIENT1.PORT=25800;RULECLIENT1.MAXACTIVE=100;RULECLIENT1.MAXIDLE=20;RULECLIENT2.ADDRESS=server2;RULECLIENT2.PORT=25800;RULECLIENT2.MAXACTIVE=100;RULECLIENT2.MAXIDLE=20; |

4の条件に基づいて、次のように修正される。

|  |
| --- |
| MAIN.TYPE=TCP;MAIN.SVRCNT=2;MAIN.POOLTYPE=FAILOVER;MAIN.CHECKCONNECTION=YES;RULECLIENT1.ADDRESS=server1;RULECLIENT1.PORT=25800;RULECLIENT1.MAXACTIVE=100;RULECLIENT1.MAXIDLE=20;RULECLIENT2.ADDRESS=server2;RULECLIENT2.PORT=25800;RULECLIENT2.MAXACTIVE=100;RULECLIENT2.MAXIDLE=20; |

5の条件に基づいて、次のように修正される。

|  |
| --- |
| MAIN.TYPE=TCP;MAIN.SVRCNT=2;MAIN.POOLTYPE=FAILOVER;MAIN.CHECKCONNECTION=YES;RULECLIENT1.ADDRESS=server1;RULECLIENT1.PORT=25800;RULECLIENT1.TIMEOUT=30000;RULECLIENT1.MAXACTIVE=100;RULECLIENT1.MAXIDLE=20;RULECLIENT2.ADDRESS=server2;RULECLIENT2.PORT=25800;RULECLIENT2.MAXACTIVE=100;RULECLIENT2.MAXIDLE=20; RULECLIENT2.TIMEOUT=30000; |

## コールバックAPI

C/C++ APIにはコールバックサービスを実装して、ルールインタフェースに登録できるAPIが提供される。

### コールバックサービスハンドラー

コールバックサービスハンドラーは、ルールエンジンからのコールバックサービス要求を処理する関数である。この関数は次の表のとおりである。

[表 53 コールバックサービスハンドラー]

|  |
| --- |
| 1 typedef int (\*IRC\_CALLBACKFUNC)( IRCDESCRIPTOR cbinfo, IRCDESCRIPTOR in,  2 IRCDESCRIPTOR out, void\*ctx,  3 char\* msg, int msglen ); |

cbInfoパラメータは、呼び出されたコールバックサービスの情報を保持しているディスクリプタである。このディスクリプタを利用してコールバックサービスを呼び出したコールバックルールやルール呼び出し基準日、コールバックサービス名などの情報を照会できる。inはコールバック入力パラメータセットを持っているディスクリプタで、outはコールバック出力パラメータセットを保持しているディスクリプタである。outはコールバックサービスの結果の格納にも使用される。ctxはコールバックコンテキストで「2.1.7 ルールサービス呼び出しおよびルール結果照会関数」のIRCExecuteに引き渡されたpCtxがそのまま引き渡され、ルールを呼び出したルーチンとコールバックサービスハンドラーとのデータ交換に使用される。この関数は、コールバックサービスの処理に成功した場合はIRCTRUEを返し、失敗した場合はIRCFALSEを返す。失敗した場合は、msgが示す領域にその事由メッセージを格納できる。msgLenはmsgバッファーの有効なサイズである。

アプリケーションはこの形式の関数を、次の表の関数を利用してコールバックレジストリに登録できる。C APIは、Java APIと異なり、ルールインタフェースクラスタでコールバックレジストリを管理しない。コールバックレジストリはライブラリ内の静的変数で管理され、すべてのインタフェースクラスタはこれを共有する。

[表 54 コールバックサービスハンドラー登録関数]

|  |
| --- |
| 1 IRCBOOL IRC\_API IRCBindCallbackService( IRCHANDLE hCluster, char const\* name, IRC\_CALLBACKFUNC func );  2 void IRC\_API UnbindCallbackService( IRCHANDLE hCluster, char const\* name ); |

次は、これらの関数についての説明である。

IRCBOOL IRC\_API IRCBindCallbackService( IRCHANDLE hCluster, char const\* name, IRC\_CALLBACK

FUNC func )

コールバックレジストリにコールバックサービスハンドラーを登録する。nameパラメータはコールバックサービス名であり、funcはコールバックサービスハンドラーに対する関数ポインタである。nameパラメータがnullである場合、funcはデフォルトコールバックサービスハンドラーとして指定される。指定された名前のコールバックサービスに対してハンドラーが既に登録されている場合は、ハンドラーはfuncに交換される。デフォルトコールバックハンドラーを登録する際、既に他のデフォルトコールバックハンドラーが登録されていれば、新しいハンドラーに交換される。funcがnullの場合、この関数はIRCFALSEを返し、そうでない場合はIRCTRUEを返す。

void IRC\_API UnbindCallbackService( IRCHANDLE hCluster, char const\* name )

コールバックレジストリでnameパラメータとして指定されたサービスのハンドラーを削除する。nameがnullの場合は、デフォルトコールバックサービスハンドラーが登録されていればこれを削除する。

### コールバックパラメータの照会および値の設定

コールバックサービスハンドラー内でコールバックサービスの入出力情報を照会し出力値を設定できる関数が、次の表のように提供される。

[表 55 コールバック入力パラメータの照会]

|  |
| --- |
| 1 void IRCCallbackReqGet( IRCDESCRIPTOR cbinfo, char\* ruleId, int ruleIdLen,  2 char\* date, int dateLen, char\* service, int serviceLen );  3 IRCDESCRIPTOR IRCCallbackLookupParam( IRCDESCRIPTOR params, char const\* paramName );  4 int IRCCallbackGetParamCount( IRCDESCRIPTOR params );  5 char const\* IRCCallbackGetParamName( IRCDESCRIPTOR params, int index );  6 void IRCCallbackGetParamLists( IRCDESCRIPTOR params,  7 IRCDESCRIPTOR\* paramDescs, int size );  8 void IRCCallbackParamGet( IRCDESCRIPTOR param, char\* name, int namelen,  9 short\* type, int\* size );  10 IRC\_API int IRCCallbackParamGetLength( IRCDESCRIPTOR param );  11 double IRCCallbackParamGetNumber( IRCDESCRIPTOR param, int row );  12 char const\* IRCCallbackParamGetString( IRCDESCRIPTOR param, int row );  13 IRCBOOL IRCCallbackParamGetBoolean( IRCDESCRIPTOR param, int row );  14 void IRCCallbackParamAddNumber( IRCDESCRIPTOR param, double num );  15 void IRCCallbackParamAddString( IRCDESCRIPTOR param, char const\* str );  16 void IRCCallbackParamAddBoolean( IRCDESCRIPTOR param, IRCBOOL b ); |

次は、各関数についての説明である。

void IRCCallbackReqGet( IRCDESCRIPTOR cbinfo, char\* ruleId, int ruleIdLen, char\* date, int dateLen, char\* service, int serviceLen )

コールバックサービスの呼び出しに関連する情報を提供する関数である。cbInfoパラメータは、コールバックサービスハンドラーにパラメータとして提供されたコールバックサービス情報のディスクリプタである。ruleIdはコールバックサービスを呼び出したコールバックルールのIDを受け取るバッファーのポインタ、dateはルール呼び出し基準日を受け取るバッファーのポインタ、serviceはコールバックサービス名を受け取るバッファーのポインタである。ruleIdLen、dateLen、serviceLenは、それぞれruleId、date、serviceのバッファーの長さである。ruleId、date、serviceにnullを引き渡すことができる。その場合、該当する情報のみ提供されない。

IRCDESCRIPTOR IRCCallbackLookupParam( IRCDESCRIPTOR params, char const\* paramName)

コールバックパラメータセットから特定のコールバックパラメータのディスクリプタを検索する関数である。paramsパラメータは、コールバック入力パラメータセットのディスクリプタまたはコールバック出力パラメータセットのディスクリプタである。paramNameは、検索するコールバックパラメータの名前である。指定された名前のコールバックパラメータが存在しない場合は、nullを返す。

int IRCCallbackGetParamCount( IRCDESCRIPTOR params )

コールバックパラメータセットに保存されているコールバックパラメータの数を返す関数である。paramsは、コールバック入力パラメータセットのディスクリプタまたはコールバック出力パラメータセットのディスクリプタである。

char const\* IRCCallbackGetParamName( IRCDESCRIPTOR params, int index )

コールバックパラメータセットから特定のインデックスのコールバックパラメータの名前を返す。paramsはコールバックパラメータセットのディスクリプタで、indexは名前を取得する0ベースのインデックスである。

void IRCCallbackGetParamLists( IRCDESCRIPTOR params, IRCDESCRIPTOR \* paramDescs, int size )

コールバックパラメータセットからコールバックパラメータのディスクリプタを取得する。paramsは、コールバックパラメータセットのディスクリプタである。paramDescsは、コールバックパラメータディスクリプタを格納する配列で、sizeは配列の大きさである。配列の大きさがコールバックパラメータの数より小さい場合は、配列の大きさの分のみ埋め込まれる。配列の大きさがコールバックパラメータの数より大きい場合は、コールバックパラメータの数の分のみ埋め込まれる。

void IRCCallbackParamGet( IRCDESCRIPTOR param, char\* name, int namelen, short\* type, int\* size )

コールバックパラメータの情報をインポートする関数である。paramは、情報を取得するコールバックパラメータのディスクリプタである。name、nameLenは、パラメータの名前を格納するバッファーおよびバッファーの長さである。typeとsizeは、それぞれコールバックパラメータのデータ型と値の配列の長さである。データ型に関連する定数については、[表 45 データ型定数]を参照のこと。nameはnullの設定は可能だが、typeとsizeはnullは許容しない。

IRC\_API int IRCCallbackParamGetLength( IRCDESCRIPTOR param )

コールバックパラメータの値配列の長さを返す関数である。Paramは値を取得するコールバックパラメータのディスクリプタである。

double IRCCallbackParamGetNumber( IRCDESCRIPTOR param, int row )

数値型のコールバックパラメータの値の配列の中から特定のインデックスの数値を返す。paramは値を取得するコールバックパラメータのディスクリプタで、このパラメータのデータ型は必ず数値型とする。rowは値の配列内でのインデックスで、値の配列の長さより小さい必要がある。

char const\* IRCCallbackParamGetString( IRCDESCRIPTOR param, int row )

文字型のコールバックパラメータの値の配列の中から特定のインデックスの文字値を返す。paramは値を取得するコールバックパラメータのディスクリプタで、このパラメータのデータ型は必ず文字型とする。rowは値の配列内でのインデックスで、値の配列の長さより小さい必要がある。

IRCBOOL IRCCallbackParamGetBoolean( IRCDESCRIPTOR param, int row )

論理型のコールバックパラメータの値の配列の中から特定のインデックスの論理値を返す。paramは値を取得するコールバックパラメータのディスクリプタで、このパラメータのデータ型は必ず論理型とする。rowは値の配列内でのインデックスで、値の配列の長さより小さい必要がある。

void IRCCallbackParamAddNumber( IRCDESCRIPTOR param, double num )

数値型の出力コールバックパラメータの値の配列に値を追加する。paramはコールバック出力パラメータのディスクリプタで、このパラメータのデータ型は必ず数値型とする。numは追加する数値である。

void IRCCallbackParamAddString( IRCDESCRIPTOR param, char const\* str )

文字型の出力コールバックパラメータの値の配列に値を追加する。paramはコールバック出力パラメータのディスクリプタで、このパラメータのデータ型は必ず文字型とする。strは追加する文字値である。

void IRCCallbackParamAddBoolean( IRCDESCRIPTOR param, IRCBOOL b )

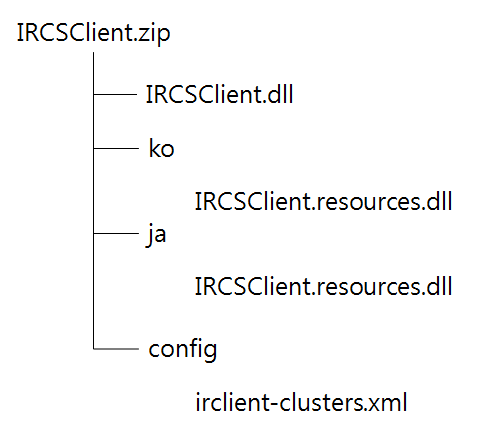
論理型の出力コールバックパラメータの値の配列に値を追加する。paramはコールバック出力パラメータのディスクリプタで、このパラメータのデータ型は必ず論理型とする。bは追加する論理値である。

# C#

C# APIを利用してルールサービスを呼び出すC#アプリケーションを作成して実行できる。TCP/IPを利用したリモートルールエンジンの呼び出しのみがサポートされる。

## 必要なライブラリ

配布されるルールサービスAPIライブラリは、API、Resource、設定ファイルで構成されている。(次の図)



[図 7 ルールAPIライブラリの構成]

[表 56 ルールAPIライブラリの構成]

| ファイル名 | 説明 | 備考 |
| --- | --- | --- |
| IRCSClient.dll | ルールAPIライブラリにReferenceを追加して使用する。 |  |
| IRCSClient.resources.dll | 言語別メッセージ処理のためのライブラリ | 韓国語 ko  日本語 ja |
| irclient-clusters.xml | ルールインタフェースクラスタの構成のための設定ファイル |  |

## 初期化モジュール

ルールインタフェースクラスタを初期化するAPIについて説明する。

ルールの初期化は、ルールアプリケーションの起動時に一度のみ実行される。

設定ファイル(irclient-clusters.xml)を使用できない場合は、Propertiesオブジェクトによって設定情報を指定できる。

### 初期化モジュールの呼び出し順序

初期化モジュールの呼び出しするためには、次の手順に従う。

1.設定情報を定義する。(irclient-clusters.xmlまたはPropertiesオブジェクト)

2.ライブラリリソースについての言語を設定する。OSの基本言語を使用する場合は、省略できる。

3.Initializerの初期化メソッドを呼び出す。

次の表は設定ファイルによってルールサービスの初期化を実行する例のアプリケーションコードで、[表 58 Propertiesオブジェクトによるルールサービス初期化アプリケーション]はPropertiesオブジェクトによって初期化を実行するアプリケーションサンプルコードである。このサンプルコードを基に、ルールサービスの初期化の順序を説明する。

[表 57 設定ファイルによるルールサービス初期化アプリケーション]

|  |
| --- |
| 1  2 public partial class MainModule  3 {  4 public MainModule ()  5 {  6 InnoRules.Resources.Msg.SetCultureInfo("ko");  7 SimpleInitializer.Initialize(@"D:\work\conf\irclient-clusters.xml");  8 }  9 } |

この例は、設定ファイルによる初期化を実行する。[6行目]はメッセージ処理に関連する言語を設定する部分で、OSの言語を同様に使用する場合は省略できる。[7行目]は、ルールインタフェースクラスタを初期化するプロセスである。

[表 58 Propertiesオブジェクトによるルールサービス初期化アプリケーション]

|  |
| --- |
| 1  2 public partial class MainModule  3 {  4 public MainModule ()  5 {  6 InnoRules.Resources.Msg.SetCultureInfo("ko");  7 InnoRules.Utility.Properties prop = new InnoRules.Utility.Properties();  8 prop.Put("POOLTYPE", "loadbalance");  9 prop.Put("WHENEXHAUSTEDACTION", "block");  10 prop.Put("AUTORECOVERY", "yes");  11 prop.Put("CHECKCONNECTION", "yes");  12 prop.Put("RECOVERYCHECKINTERVAL", "30");  13 prop.Put("SVRCNT", "2");  14 prop.Put("SVR1.NAME", "rulesclient@1");  15 prop.Put("SVR1.MAXACTIVE", "2");  16 prop.Put("SVR1.MAXIDLE", "4");  17 prop.Put("SVR1.ADDRESS", "100.2.50.1");  18 prop.Put("SVR1.PORT", "25800");  19 prop.Put("SVR1.CHARSET", "UTF-8");  20 prop.Put("SVR2.NAME", "rulesclient@2");  21 prop.Put("SVR2.MAXACTIVE", "2");  22 prop.Put("SVR2.MAXIDLE", "4");  23 prop.Put("SVR2.ADDRESS", "100.2.50.2");  24 prop.Put("SVR2.PORT", "25800");  25 prop.Put("SVR2.CHARSET", "UTF-8");  26  27 SimpleInitializer.Initialize("rulesclient", prop);}  28 }  29 } |

この例は、Propertiesによる初期化を実行する。[6行目]はメッセージ処理に関連する言語を設定する部分で、OSの言語を同様に使用する場合は省略できる。[8~25行目]は設定情報をPropertiesに設定するプロセスで、[27行目]はルールインタフェースクラスタを初期化するプロセスである。

### InnoRules.Client.SimpleInitializer

ルールサービスに対する初期化を実行するクラスである。SimpleInitializerクラスには、次の表のメソッドが定義されている。

[表 59 InnoRules.Client.SimpleInitializer]

|  |
| --- |
| 1 namespace InnoRules.Client  2 {  3 public class SimpleInitializer  4 {  5 public static void Initialize(String xmlFile)  6 public static void Initialize(String clusterName, Properties props)  7 }  8 } |

public static void Initialize(String xmlFile)

設定情報をxmlファイル(irclient-clusters.xml)に定義して初期化を実行する。

ファイル名を含むFull Pathを使用する。

public static void Initialize(String clusterName, Properties props)

設定情報をInnoRules.Util.Propertiesに定義して初期化を実行する。

clusterName名は設定情報を適用するクラスタの名前で、Propertiesに使用するクラスタ名と同一名称を使用する。

Propertiesに定義できる構成要素は、次のとおりである。

[表 60 クラスタ設定情報]

| 構成要素 | 説明 | 設定ファイル構成要素のマッピング |
| --- | --- | --- |
| POOLTYPE | クラスタリング方式 | type |
| WHENEXHAUSTEDACTION | インタフェース破棄時の動作方式 | when-exhausted-action |
| AUTORECOVERY | クラスタ自動復元の実行有無 | auto-recovery |
| CHECKCONNECTION | インタフェース有効性検証の実行有無 | check-connection |
| RECOVERYCHECKINTERVAL | プール障害復旧チェックの周期 | recovery-check-interval |
| SVRCNT | プール数 | プール集合の数 |
| プール名.NAME \*4 | プールに対するクラスタ名  クラスタ名 +〝@″+ 1からの連番形式で構成 | irclient-clusterのNAME |
| プール名.MAXACTIVE | プールに対する最大有効インタフェース数 | propertyのMAXACTIVE |
| プール名.MAXIDLE | プールに対するアイドル状態の最大インタフェース数 | propertyのMAXIDLE |
| プール名.ADDRESS | プールに対するリモートホストのアドレス | propertyのADDRESS |
| プール名.PORT | プールに対するリモートホストのポート | propertyのPORT |
| プール名.CHARSET | プールに対するエンコード情報 | propertyのCHARSET |

1. プール名はSVR+1からの連番形式で必ず使用し、プール名を重複して使用する場合はエラーが発生する。

### InnoRules.Resources.Msg

ルールサービスのエラーまたは情報に関するメッセージ処理のためのクラスである。Msgクラスには次の表のメソッドが定義されている。

[表 61 InnoRules.Resources.Msg]

|  |
| --- |
| 1 namespace InnoRules.Resources  2 {  3 public class Msg  4 {  5 public static void SetCultureInfo(String lang)  6 }  7 } |

public static void SetCultureInfo(String lang)

メッセージについての言語を指定する。設定しない場合は、OSの基本言語に設定される。

韓国語の場合は“ko”、日本語の場合は“ja”を指定する。

### InnoRules.RulesClient.Release

ルールサービスAPIに関するバージョン情報を照会するためのクラスである。Releaseクラスには、次の表のメソッドが定義されている。

[表 62 InnoRules.Client.Release]

|  |
| --- |
| 1 namespace InnoRules.Client  2 {  3 public class Release  4 {  5 public static String GetReleaseInfo()  6 }  7 } |

public static String GetReleaseInfo()

バージョン情報を照会する。

## ルールサービス呼び出しAPI

### ルールインタフェースの取得

ルールサービスは、ルールインタフェース(Rule Interface)と呼ばれるルールエンジンプロキシオブジェクトを利用して呼び出す。アプリケーションは、実行環境内に構成されているルールインタフェースクラスタからルールインタフェースを取得してルールサービスを呼び出す。ルールインタフェースクラスタは、ルールインタフェースクラスタマネージャから取得できる。ルールインタフェースクラスタマネージャは、複数のルールインタフェースクラスタを管理する。各クラスタには名前が設定されており、この名前を利用してクラスタマネージャからクラスタを検索できる。

InnoRules.RulesClient.ClusterManagerクラスは、ルールインタフェースクラスタマネージャを表現するクラスである。次の表はClusterManagerクラスの一部のメソッドを示す。

[表 63 InnoRules.RulesClient.ClusterManager]

|  |
| --- |
| 1 namespace.InnoRules.RulesClient  2 {  3 public class ClusterManager  4 {  5 public static InterfaceCluster Get( String clusterName ) ;  6 public static InterfaceCluster Get() ;  7 …  8 }  9 } |

ClusterManagerクラスには、ルールインタフェースクラスタを提供する2つのGetメソッドがある。ルールインタフェースクラスタは、InnoRules.RulesClient.InterfaceClusterインタフェースで表現される。

最初のメソッドはクラスタの名前をパラメータとして受け取り、この名前を持つクラスタを返す。指定された名前のクラスタが存在しない場合は、RulesExceptionを投げる。2番目のメソッドはパラメータを受け取らないメソッドで、“rulesclient”という名前のルールインタフェースクラスタを返す。“rulesclient”は、基本ルールインタフェースクラスタの名前である。このメソッドは、名前が“rulesclient”であるルールインタフェースクラスタがマネージャに登録されていない場合、RulesExceptionを投げる。

次の表は、InterfaceClusterクラスの一部のメソッドを示す。

[表 64 InnoRules.RulesClient.InterfaceCluster]

|  |
| --- |
| 1 namespace.InnoRules.RulesClient  2 {  3 public interface InterfaceCluster  4 {  5 RuleInterface GetInterface() ;  6 String GetName();  7 …  8 }  9 } |

GetInterfaceは、ルールインタフェースクラスタからルールインタフェースを取得するメソッドである。InnoRules.RulesClient.RuleInterfaceは、ルールインタフェースを表すインタフェースである。取得したRuleInterfaceオブジェクトは、使用終了後、RuleInterface.Close()メソッドを呼び出してルールインタフェースクラスタに返される必要がある。ルールインタフェースクラスタから取得したルールインタフェースを返さない場合、リソースのリークの発生または全体的なアプリケーションの性能が低下することがある。ルールインタフェースクラスタを構成するすべてのルールインタフェースプールから有効なルールインタフェースをアプリケーションに返すことができない場合、RulesExceptionが投げられる。TCP/IPを利用してリモートのルールエンジンに接続できない場合、ルールインタフェースクラスタは有効なルールインタフェースを返すことができない。

GetNameメソッドは、このルールインタフェースクラスタの名前を返す。

次の表は、“dev”というルールインタフェースクラスタからルールインタフェースを取得し、ルールインタフェースを使用後に返すソースの例である。例では、finallyブロックでRuleInterface.Close()を呼び出すことがポイントである。RuleInterfaceのリークを防ぐ方法として、ルールインタフェースクラスタからルールインタフェースを取得した後、try-finallyブロックでRuleInterface.close()を呼び出すことを推奨する。

[表 65 ルールインタフェースを取得して返すサンプル]

|  |
| --- |
| 1 using InnoRules.Client;  2 using InnoRules.Tcp;  3 …  4  5 InterfaceCluster cluster = ClusterManager.Get( "dev" );6 RuleInterface intf = cluster.GetInterface();  7  8 try  9 {  10 // use RuleInterface object and invoke rule services  11 }  12 finally  13 {  14 intf.Close();  15 } |

[表 65 ルールインタフェースを取得して返すサンプル]で、ルールインタフェースの取得は2段階で行われる。最初の段階はルールインタフェースクラスタマネージャから“dev”という名前のルールインタフェースクラスタを取得するもので、2番目の段階はルールインタフェースクラスタからルールインタフェースをインポートするものである。ルールインタフェースクラスタマネージャは、この2段階を一度に実行するメソッドを提供する。

[表 66 InnoRules.RulesClient.ClusterManager.GetInterface()]

|  |
| --- |
| 1 namespace.InnoRules.RulesClient  2 {  3 public class ClusterManager  4 {  5 // equivalent to Get( clusterName ).GetInterface()  6 public static RuleInterface GetInterface( String clusterName );  7  8 // equivalent to Get().GetInterface()  9 public static RuleInterface GetInterface() ;  10 …  11 }  12 } |

ClusterManagerクラスは、2つのGetInterfaceメソッドを提供する([表 66 InnoRules.RulesClient.ClusterManager.GetInterface()])。このメソッドは、同一パラメータを持つGetメソッドの呼び出し後に返されるInterfaceClusterのgetInterfaceを呼び出すことと同様である。つまり、ClusterManager.Get(clusterName).GetInterface()はClusterManager.GetInterface(clusterName)と同一である。次の表のソースコードは[表 65 ルールインタフェースを取得して返すサンプル]のコードをこのメソッドを使用して変換したもので、[表 65 ルールインタフェースを取得して返すサンプル]と同様に動作する。

[表 67 ルールインタフェースを取得して返すサンプル]

|  |
| --- |
| 1 using InnoRules.Client;  2 using InnoRules.Tcp;  3 …  4  5 RuleInterface intf = ClusterManager.GetInterface( "dev" );  6  7 try  8 {  9 // use RuleInterface object and invoke rule services  10 }  11 finally  12 {  13 intf.Close();  14 } |

### InnoRules.RulesClient.RuleInterface

RuleInterfaceインタフェースには、ルールサービスを呼び出せる多様なメソッドが定義されている。このメソッドは、ルールエンジンとの通信プロトコルに依存しないように定義されている。したがって、プロトコルに依存しないアプリケーションを作成でき、アプリケーションの運用中にプロトコルが変更されてもアプリケーションの修正やリコンパイルが不要となる。次の表は、RuleInterfaceに定義されたルールサービス呼び出しメソッドを示す。

[表 68 InnoRules.RulesClient.RuleInterface]

|  |
| --- |
| 1 namespace InnoRules.RulesClient;  2 {  3 public interface RuleInterface  4 {  5 ResultSet Execute ( RuleReq req, int codetype );  6 ResultSet Execute ( RuleReq req, int codetype, Object ctx );  7 void Execute ( String rule, ItemProvider itemProvider, ResultConsumer consumer,  8 int codetype, String date, Object ctx );  9 void Execute ( String rule, ItemProvider itemProvider, ResultConsumer consumer,  10 int codetype, String date, Object ctx, Trace trace,  11 HashMap<Object, Object>callctx );  12 …  13 } |

Executeメソッドは、入出力値の引き渡し方式によって2種類に分類できる。最初と2番目のメソッドは、RuleReqオブジェクトをパラメータとして受け取りResultSetオブジェクトを返す。3番目と4番目のメソッドは、ItemProviderとResultConsumerオブジェクトをパラメータとして受け取る。前者をI/Oオブジェクト(I/O Object)方式といい、後者をI/Oアダプタ(I/O Adapter)方式という。各方式については「3.3.5 I/Oオブジェクト方式」と「3.3.6 I/Oアダプタ方式」で説明し、「1.2.8 ルールエンジンの構成方式に基づくI/Oオブジェクト方式とI/Oアダプタ方式の性能比較」で各方式のメリットとデメリットを説明する。

### コード体系

I/Oオブジェクト方式では、呼び出すルールコードと入力項目のコードおよび値がRuleReq入力オブジェクトに保存されて引き渡される。I/Oアダプタ方式では、呼び出すルールのコードがExecuteメソッドのパラメータとして引き渡される。コードとして使用できる値は、次のとおりである。

* ルールまたは項目のID
* ルールまたは項目の名前
* ルールまたは項目のエイリアス

IDは、ルールシステムが管理目的ですべてのルールと項目に付与するアルファベットと数字からなる10桁のコードである。ルールシステムで自動的に付与されたIDは、‘#S’で始まる。IDは固有のものが付与されるため、ルール同士または項目同士で重複することはない。IDは、一度付与されると変更できない。

名前は、ルール管理者がすべてのルールと項目に付与する127文字以内のコードである。いくつかの特殊文字を除き、大部分の文字が使用できる。ルールビルダーでのすべてのルールの内容は名前で表現されるため、大概は業務に合わせた名前が付与される。名前もIDと同様ルールシステム内では固有のものであるため、ルール同士または項目同士で重複はしないが、変更は可能である。

IDはルールシステムから識別および管理のために付与され、名前はルール管理者が業務的な意味を込めて付与する。一般的にITシステムでは様々な制約条件によって業務的な意味を持つ名前をそのまま使用せず、制限された文字を使う。例えばアルファベットの大文字と数字、‘\_’で構成された別のコード体系を運用する。エイリアスは、ルールまたは項目に、このコード体系に基づくコードを付与できるようにする。ルールのエイリアスは、IDまたは名前とは異なり、すべてのルールと項目に付与する必要はない。エイリアスは必要なルールと項目に任意で付与でき、付与された後でも変更が可能である。ただし、付与されるとルール同士または項目同士では重複できない。

ルールサービスを呼び出す時にどのルールを呼び出すか、どの項目の値を設定するかを指定するために、ID、名前、エイリアスの中から1つのコード体系を選択できる。ただし、1つのルールの呼び出しには1つのコード体系のみ使用できる。例えば、ルールを指定するためにIDを使用する場合は、項目もIDを利用して指定する必要がある。この文書の以降の部分においてのコードとは、ID、名前またはエイリアスのうちの1つを意味する。

ルールおよび項目のコードの指定にどのコード体系が使用されたかは、[表 68 InnoRules.RulesClient.RuleInterface]のRuleInterface.Executeメソッドのcodetypeパラメータに指定する。InnoRules.RulesClient.Constantsには、次の表のようにcodetypeの値で指定できる定数が定義されている。

[表 69 コード体系を指定する定数の値]

|  |
| --- |
| 1 namespace InnoRules.RulesClient  2 {  3 public class Constants  4 {  5 public const short CODETYPE\_ID = 0x00;  6 public const short CODETYPE\_NAME = 0x01;  7 public const short CODETYPE\_ALIAS = 0x02;  8 …  9 }  10 } |

CODETYPE\_IDは使用されたコード体系がIDであることを意味し、CODETYPE\_NAMEは名前を、CODETYPE

\_ALIASはエイリアスを意味する。

### データ型と関連定数

ルールサーバは、3つの基本データ型をサポートする。

* 数値型
* 文字型
* 論理型

項目の形式やルール実行結果カラムの形式は、上記のデータ型のうちの1つに該当する。項目またはルール結果カラムがどのような形式なのかを識別するのに使われる定数が、次の表のようにInnoRules.RulesClient.Constantsに定義されている。

[表 70 データ型定数]

|  |
| --- |
| 1 namespace InnoRules.RulesClient  2 {  3 public class Constants  4 {  5 public const short TYPE\_NONE = 0x00;  6 public const short TYPE\_NUMBER = 0x01;  7 public const short TYPE\_STRING = 0x02;  8 public const short TYPE\_BOOLEAN = 0x03;  9 …  10 }  11 } |

TYPE\_NONEは、データ型が指定されていないことを意味する。TYPE\_NUMBERは数値型を、TYPE\_STRINGとTYPE\_BOOLEANはそれぞれ文字型と論理型を意味する。

### I/Oオブジェクト方式

I/Oオブジェクト方式は、呼び出すルールのコードや入力項目の値を1つの入力オブジェクトにすべて格納してルールを呼び出し、その結果を1つのオブジェクトに格納して受け取るルール呼び出し方式である。次の図は、I/Oオブジェクト方式のルールサービスの呼び出しを示す。



[図 8 I/Oオブジェクト方式のルールサービスの呼び出し]

ルールコードとルール実行に必要な項目値は、RuleReqという入力オブジェクト(Input Object)に格納してルールエンジンに引き渡される。ルールエンジンはそのオブジェクトに入力された情報のみを利用してルールを実行し、その結果をResultSetという出力オブジェクト(Output Object)に格納して返す。ルールエンジンがルール実行中にRuleReqに設定された項目以外の項目値が必要な場合は、アプリケーションにエラーを返す。

次の表は、RuleInterfaceに定義されたI/Oオブジェクト方式のExecuteメソッドである。このメソッドはすべてパラメータとしてRuleReqオブジェクトを受け取り、ResultSetを返す。

[表 71 I/Oオブジェクト方式のexecuteメソッド]

|  |
| --- |
| 1 namespace InnoRules.RulesClient;  2 {  3 public interface RuleInterface  4 {  5 ResultSet Execute ( RuleReq req, int codetype );  6 ResultSet Execute ( RuleReq req, int codetype, Object ctx );  7 …  8 }  9 } |

#### InnoRules.Tcp.RuleReq

InnoRules.Tcp.RuleReqクラスは、I/Oオブジェクト方式で入力オブジェクトとして使用されるクラスである。このクラスには呼び出すルールのコードおよび入力項目値を設定し、設定された値を照会するメソッドの他にも、ルール実行に使用される付加情報を設定するメソッドが定義されている。RuleReqクラスには、次の表のメソッドが定義されている。

[表 72 InnoRules.Tcp.RuleReq]

|  |
| --- |
| 1 namespace InnoRules.Tcp  2 {  3 public class RuleReq  4 {  5 public void SetRuleCode( String ruleCode );  6 public String GetRuleCode();  7  8 public void SetDate( String date );  9 public String GetDate();  10  11 public Item AddNumberItem( String code );  12 public Item AddStringItem( String code );  13 public Item AddBooleanItem( String code );  14 public ReadOnlyCollection<Item> GetItems();  15 public void ResetItems();  16  17 public void SetWantTrace( bool wantTrace );  18 public bool GetWantTrace();  19  20 public void SetContext( Hashtable context );  21 public Hashtable SetContext();  24  25 public ItemProvider CreateItemProvider();  26  27 } |

次は、各メソッドに関する説明である。

void SetRuleCode( String ruleCode )

呼び出すルールのコードを設定する。ruleCodeパラメータはルールのコードであり、このコードはRuleInterface.Executeに引き渡されるコード体系のコードとする。例えば、ExecuteメソッドにConstants.CODETYPE\_IDがコード体系として指定された場合、ruleCodeパラメータは呼び出すルールのIDとなる。

String GetRuleCode()

RuleReqオブジェクトに設定されたルールコードを返す。

void SetDate( String date )

ルール実行の基準日付を指定する。1つのルールには、日付別に異なる規則が定義できる。これをルールのバージョンという。ルールを呼び出す時に基準日を指定すると、ルールエンジンはその日付のバージョンを検索して実行する。基準日が指定されていない場合は、ルールエンジンのシステム日付を使用する。dateパラメータは‘yyyy-MM-dd’または‘yyyyMMdd’形式の基準日である。nullに設定される場合は、基準日を指定しない。

String GetDate()

RuleReqオブジェクトに設定された基準日を返す。設定されていない場合は、nullを返す。

ReadOnlyCollection<Item> GetItems()

RuleReqオブジェクトに設定されている項目、つまりItemのReadOnlyCollectionオブジェクトを返す。

void ResetItems()

RuleReqオブジェクトに設定されている項目をリセットする。設定された項目情報は、すべて削除される。ただし、ルールコードや基準日のような項目以外の情報はそのまま保持される。

void SetWantTrace( bool wantTrace )

ルールを実行する際、実行経路トレース可否について設定する。TRUEを設定すると、ルールエンジンは実行経路トレースを作成し、これを出力オブジェクトに格納して返す。FALSEを設定すると、実行経路トレースを作成しない。実行経路トレースを作成する場合、ルール実行性能がわずかに低下し、出力オブジェクトの大きさがわずかに増加することがある。そのため、実行経路トレースは問題解決の目的のみでの使用を推奨する。

bool GetWantTrace()

実行経路トレースオプションの設定状況について返す。設定されていればTRUEを、そうでなければFALSEを返す。

void SetContext( Hashtable context )

ルールを実行する時、必要なルール実行情報を格納したcontextを設定する。このオブジェクトはルールを実行時、追加で使用されるルール実行結果のキャッシュまたはルール実行経路トレースの可否などルールインタフェースを実装したクラスで使用できる情報を格納して使用できる。

Hashtable GetContext()

ルールを実行する時、使用可能なcontextオブジェクトを返す。

ItemProvider CreateItemProvider()

I/Oオブジェクト方式のオブジェクトを、I/Oアダプタ方式のオブジェクトに変換して返す。

#### InnoRules.RulesClient.Item

InnoRules.RulesClient.Itemインタフェースは、ルール呼び出しの入力で使用される項目のコードと値を保存するためのインタフェースである。アプリケーションには項目値のタイプによって、次のクラスを使用する。

* 数値型：InnoRules.RulesClient.Item.NumberItem
* 文字型：InnoRules.RulesClient.Item.StringItem
* 論理型：InnoRules.RulesClient.Item.BooleanItem

Itemインタフェースには、次の表のメソッドが定義されている。

[表 73 InnoRules.RulesClient.Item]

|  |
| --- |
| 1 namespace InnoRules.RulesClient  2 {  3 public interface Item  4 {  5 String GetCode();  6 byte GetItemType();  7 int Length();  8  9 Item Add( double value ) ;  10 Item Add( String value ) ;  11 Item Add( bool value ) ;  12  13 double GetNumber( int index ) ;  14 String GetString( int index ) ;  15 bool GetBoolean( int index ) ;  16 } |

次は、各メソッドに関する説明である。

String GetCode()

Itemオブジェクトの項目コードを返す。

byte GetType()

Itemオブジェクトのデータ型を返す。返されるデータ型コードについては、「3.3.4 データ型と関連定数」のデータ型と関連定数を参照のこと。

int Length()

Itemオブジェクトの項目値の配列の長さを返す。

Item Add( double value )

Itemの配列に数値を追加する。数値を使用できないStringItemまたはBooleanItemのオブジェクトの場合は、UnsupportedOperationExceptionが投げられる。このメソッドは、オブジェクト自身を返す。

Item Add ( String value )

Itemの配列に文字値を追加する。文字値を使用できないNumberItemまたはBooleanItemのオブジェクトの場合は、UnsupportedOperationExceptionが投げられる。このメソッドは、オブジェクト自身を返す。

Item Add ( bool value )

Itemの配列に論理値を追加する。論理値を使用できないStringItemまたはNumberItemのオブジェクトの場合は、UnsupportedOperationExceptionが投げられる。このメソッドは、オブジェクト自身を返す。

double GetNumber ( int index )

Item値の配列のindex番目の数値を返す。indexは、0ベース(0-base)のインデックスである。配列の長さがindexより大きくない場合は、IndexOutOfBoundsExceptionが投げられる。数値を使用できないStringItemまたはBooleanItemのオブジェクトの場合は、UnsupportedOperationExceptionが投げられる。

String GetString ( int index )

Item値の配列のindex番目の文字値を返す。indexは、0ベース(0-base)のインデックスである。配列の長さがindexより大きくない場合は、IndexOutOfBoundsExceptionが投げられる。文字値を使用できないNumberItemまたはBooleanItemのオブジェクトの場合は、UnsupportedOperationExceptionが投げられる。

bool GetBoolean ( int index )

Item値の配列のindex番目の論理値を返す。indexは、0ベース(0-base)のインデックスである。配列の長さがindexより大きくない場合は、IndexOutOfBoundsExceptionが投げられる。論理値を使用できないStringItemまたはNumberItemのオブジェクトの場合は、UnsupportedOperationExceptionが投げられる。

Item.addメソッドは自身の参照を返すので、addメソッドを連結するコーディングパターンを使用できる。[表 74 NumberItemの例]はそのようなパターンを使用して[表 75 簡略化されたNumberItemの例]のコードを修正したもので、2つのコードは同様に動作する。

[表 74 NumberItemの例]

|  |
| --- |
| 1 NumberItem item = new NumberItem( "ITEMIDHERE" );  2 item.Add( 1.0 );  3 item.Add( 2.0 );  4 item.Add( 3.0 ); |

[表 75 簡略化されたNumberItemの例]

|  |
| --- |
| 1 NumberItem item = new NumberItem( "ITEMIDHERE" );  2 item.Add( 1.0 ).Add( 2.0 ).Add( 3.0 ); |

#### com.innoexpert.rulesclient.ResultSet

ルール呼び出し結果データをテーブル形式で表現するインタフェースである。ResultSetは、データの中で現在の行を示すカーソルを保持している。RuleInterface.Executeメソッドで返された直後、このカーソルは最初の行のすぐ前を示す。Nextメソッドはカーソルを1行ずつ進め、それ以上進めない時、つまりデータの最後ではFALSEを返す。したがって、while文の条件式にNextメソッドを使用できる。

ResultSetの各カラムには、インデックスと名前が付与される。特定カラム値を取得するためにに、カラムインデックスまたはカラム名が使用できる。カラムインデックスは1ベース(1-base)で、最初のカラムのインデックスは1である。カラム名は、RuleInterface.Executeを呼び出した時に指定されたコード体系によって異なる。コード体系をIDまたは名前で指定した場合、ResultSetのカラム名は呼び出したルールのリターン項目名に対応される。最初のカラム名は呼び出したルールの最初のリターン項目名で設定され、2番目以降のカラムも同一方式が適用される。コード体系をエイリアスで指定した場合、カラム名は呼び出したルールのリターン項目のエイリアスに対応する。ルールリターン項目のうち、一部のみにエイリアスが付与された場合は、エイリアスが付与されていないカラム値は名前を利用して取得できない。もちろん、カラムインデックスを利用して取得することは可能である。

ResultSetには、次の表のメソッドが定義されている。

[表 76 InnoRules.Client.ResultSet]

|  |
| --- |
| 1 namespace InnoRules.Client  2 {  3 interface ResultSet  4 {  5 bool Next();  6 bool Previous();  7 String GetString ( int columnIndex );  8 bool GetBoolean( int columnIndex );  9 double GetDouble( int columnIndex );  10 short GetShort( int columnIndex );  11 int GetInteger ( int columnIndex );  12 String GetString( String columnName );  13 bool GetBoolean( String columnName );  14 double GetDouble( String columnName );  15 short GetShort( String columnName );  16 int GetInteger( String columnName );  17 ResultSetMetaData GetMetaData();  18 int FindColumn( String columnName );  19 void BeforeFirst();  20 void AfterLast();  21 bool IsFirst();  22 bool IsLast();  23 bool IsBeforeFirst();  24 bool IsAfterLast();  25 bool First();  26 bool Last();  27 int GetRow();  28 int GetRowCount();  29 bool Absolute( int row );  30 bool Relative( int rows );  31 …  32 }  33 } |

次は、各メソッドに関する説明である。

bool Next()

カーソルを次の行に移動させる。次の行に移動できない時、つまり現在のカーソルの位置が最後の行の次にある場合は、FALSEを返す。そうでない場合は、TRUEを返す。

bool Previous()

カーソルを前の行に移動させる。前の行に移動できない時、つまり現在のカーソルの位置が最初の行の前にある場合は、FALSEを返す。そうでない場合は、TRUEを返す。

String GetString( int columnIndex )

bool GetBoolean( int columnIndex )

double GetDouble( int columnIndex )

short GetShort( int columnIndex )

int GetInteger(int columnIndex)

現在の行のcolumnIndex番号のカラム値をそれぞれdouble、short、int形式で返す。columnIndexは1-baseインデックスで、有効範囲から外れた場合はIndexOutOfBoundsExceptionが投げられる。照会するデータの類型とフィールドのデータ類型が不一致の場合でも、適切な変換を実行する。例えば、ルールの最初のフィールドが数値型で、ResultSet.GetString(1)を呼び出した場合は、APIはルール結果の数値をStringに変換する。ただし、フィールドの値を適切な数値型または論理型に変換できない場合は、FormatExceptionが投げられる。

String GetString(String columnName)

bool GetBoolean(String columnName)

double GetDouble(String columnName)

short GetShort(String columnName)

int GetInteger(String columnName)

カラム名がcolumnNameであるカラムの値をそれぞれdouble、short、int形式で返す。columnNameは、ルールサービスの呼び出しに使用されたコード体系がIDまたは名前の場合はルールのリターン項目の名前であり、コード体系がエイリアスの場合はルールのリターン項目のエイリアスである。columnNameに該当するカラムが存在しない場合は、IndexOutOfBoundsExceptionが投げられる。

ResultSetMetaData GetMetaData()

ResultSetのカラム情報を持っているResultSetMetaDataオブジェクトを返す。

int FindColumn(String columnName)

カラム名がColumnNameであるカラムのindexを返す。

void BeforeFirst()

ResultSetのカーソルを最初の行の前に移動させる。このメソッドを呼び出してGetString、GetBoolean、GetDouble、GetShort、GetIntegerのようなメソッドを呼び出す場合は、nullまたはエラーを返す。このメソッドを呼び出した後は、Next()メソッドを呼び出し、必ず結果値を参照する。

void AfterLast()

ResultSetのカーソルを最後の行の次に移動させる。このメソッドを呼び出してGetString、GetBoolean、GetDouble、GetShort、GetIntegerのようなメソッドを呼び出す場合は、nullまたはエラーを返す。このメソッドを呼び出した後は、Previous()メソッドを呼び出し、必ず結果値を参照する。

bool IsFirst()

ResultSetのカーソルが最初の行の場合は、TRUEを返す。そうでない場合は、FALSEを返す。

bool IsLast()

ResultSetのカーソルが最後の行の場合は、TRUEを返す。そうでない場合は、FALSEを返す。

bool IsBeforeFirst()

ResultSetのカーソルが最初の行の前にある場合は、TRUEを返す。そうでない場合は、FALSEを返す。

bool IsAfterLast()

ResultSetのカーソルが最後の行の次にある場合は、TRUEを返す。そうでない場合は、FALSEを返す。

bool First()

ResultSetのカーソルを最初の行に移動させる。正常に移動した場合は、TRUEを返す。ResultSetの結果、行が存在しない場合はFALSEを返す。

bool Last()

ResultSetのカーソルを最後の行に移動させる。正常に移動した場合は、TRUEを返す。ResultSetの結果、行が存在しない場合はFALSEを返す。

int GetRow()

ResultSetの現在の行のindexを返す。

int GetRowCount()

結果データの全体の行数を返す。

bool Absolute(int row)

ResultSetのカーソルをrowで指定した行に移動させる。入力された値が正の数の場合は、最初のRowの前から指定された値の分のみ移動させる。つまり、1が入力された場合はカーソルを最初のRowに移動させる。入力された値が負の数の場合は、最後のRowの次から指定された値の分のみカーソルを戻す。つまり、-1が入力された場合はカーソルを最後のRowに移動させる。0が入力された場合は、IllegalArgumentExceptionが投げられる。入力された値を最初のRowから最後のRowの範囲内で移動させる場合はTRUEを返す。入力された値が最初のRowから最後のRowの範囲外にある場合はFALSEを返し、入力された値が正の数である場合は、カーソルを最後のRowの次に移動させ、負の数である場合は最初のRowの前に移動させる。ResultSetの結果、行が存在しない場合はFALSEを返す。

bool Relative(int rows)

ResultSetのカーソルを現在の位置から指定されたrowsの分のみ移動させる。入力された値が正の数の場合は、現在の位置から指定された値の分のみカーソルを先に進める。つまり、1が入力された場合はnext()と同一である。入力された値が負の数の場合は、指定された値の分のみカーソルを戻す。つまり、-1が入力された場合はprevious()と同一である。0が入力された場合は、現在の位置から移動しない。移動させる位置が最初のRowから最後のRowの範囲内である場合はTRUEを返し、移動させる位置が最初のRowから最後のRowの範囲外である場合はFALSEを返す。カーソルを移動させる位置が正の数の場合は、最後のRowの次に移動させ、負の数の場合は最初のRowの前に移動させる。ResultSetの結果、行が存在しない場合はFALSEを返す。

#### InnoRules.Client.ResultSetMetaData

ResultSetMetaDataインタフェースはResultSetのカラム情報を提供するインタフェースで、カラム数、名前、データ型などを提供する。各カラムはルールのリターン項目に対応するが、カラム名はルールサービスの呼び出しに使用されたコード体系によってはルールのリターン名またはエイリアスになることがある。カラム名とルールのリターン項目との関係については、「3.3.5.3 com.innoexpert.rulesclient.ResultSet」を参照のこと。

ResultSetMetaDataには、次の表のメソッドが定義されている。

[表 77 InnoRules.Client.ResultSetMetaData]

|  |
| --- |
| 1 namespace InnoRules.Client  2 {  3 public class ResultSetMetaData  4 {  5 int GetColumnCount();  6 String GetColumnName( int columnIndex );  7 short GetColumnType( int columnIndex );  8 List<String> GetTraceList();  9 }  10 } |

次は、各メソッドに関する説明である。

int GetColumnCount()

カラム数を返す。

String GetColumnName(int columnIndex)

columnIndex番目のカラム名を返す。columnIndexは1ベースインデックスで、最初のカラムのインデックスは1である。columnIndexが0以下またはカラム数より大きい場合は、IndexOutOfBoundsExceptionが投げられる。

short GetColumnType(int columnIndex)

カラムのデータ型コードを返す。データ型コードについては、「3.3.4 データ型と関連定数」と関連定数を参照する。columnIndexは1ベースのインデックスであり、columnIndexが0以下またはカラム数より大きいとIndexOutOfBoundsExceptionが投げられる。

List<String> GetTraceList()

List形式のルール実行経路トレースの結果を返す。

#### ルールサービス呼び出し例

次の表は、手数料を計算するルールサービスを呼び出すソースコードの例である。

[表 78 ルールサービス呼び出し例]

|  |
| --- |
| 1  2 static String service()  3 {  4 RuleInterface intf = null;  5 String message = "";  6 try  7 {  8  9 RuleReq req = new RuleReq();  10 req.SetRuleCode("COMMISSION");  11 req.SetDate( "2013-07-01" );  12  13 Item item = req.AddNumberItem( "AMT" );  14 item.Add(100000);  15 item = req.AddStringItem( "TRDTYP" );  16 item.Add( "A" );  17  18 intf = ClusterManager.GetInterface();  19  20 ResultSet rs = intf.Execute(req, Constants.CODETYPE\_ALIAS);  22  23 if( rs != null && rs.Next() )  24 {  25 String rate = rs.GetString( "CMSRATE" );  26 String commission = rs.GetString( "CMSAMT" );  27 message = "Commission is " + commission + ". Rate is " + rate + ".";  28 }  29 }  30 catch (Exception e)  31 {  32 message = "ERROR: " + e.Message;  33 }  34 finally  35 {  36 if (intf != null)  37 intf.Close();  38 }  39 return message;  40 } |

ルールサービスを呼び出す例で、手数料計算ルールを呼び出して結果メッセージを作成する。[9~11行目]ではRuleReqオブジェクトを生成して呼び出すルールのエイリアスである“COMMISSION”を設定し、ルール呼び出しの基準日を“2013-07-01”に設定している。システムの現在の日付を使用する場合は、[11行目]を削除するか、nullを設定する。[13~16行目]では、入力項目の値を設定している。このルールは2つの項目を必要とし、そのエイリアスはそれぞれTRDTYPとAMTである。[18行目]ではルールインタフェースクラスタでマネージャからルールインタフェースを取得しており、その結果を利用して[20行目]でルールサービスを呼び出し、[37行目]でクラスタに返している。try-finallyブロックを利用し、ルール呼び出し中に例外が投げられた場合でもルールインタフェースがクラスタに返されるよう保障する。[30~33行目]では、ルール呼び出し中に投げられるエラーを処理している。呼び出すルールと項目の値を指定する際に、コード体系としてエイリアスを使用しているため、[20行目]のExecuteメソッドのパラメータとしてConstants.CODETYPE\_ALIASが引き渡されている。

[23~28行目]では、ルール呼び出しの結果を抽出してメッセージを作成している。手数料計算ルールは単一ルールであるため、結果の行数は常に1つで、rs.next()がif文の条件式に使用されているが、呼び出したルールが多重ルールである場合はrs.next()をwhile文の条件式に使用することもある。[25行目]と[26行目]では、ルールの結果から手数料率と手数料金額を抽出している。ルール呼び出しに使用されたコード体系がエイリアスであるため、ここでもルールリターンのエイリアスが使用されている。

例では、次の順序でルールサービスが呼び出されている。

1. RuleReqオブジェクト設定
2. ルールインタフェースのインポート
3. ルールの呼び出し
4. ルールインタフェースの返却
5. ルール結果の使用

ルールインタフェースは、ルールインタフェースクラスタのプールで管理される。あるアプリケーションがインタフェースを使用中の場合、別のアプリケーションはそのインタフェースを使用できない。そのため、ルールインタフェースの占有時間を短く設定することを推奨する。アプリケーションが必要以上にルールインタフェースを参照している場合、別のアプリケーションがインタフェースの取得のため待機状況となる可能性もある。これはアプリケーションの性能を低下させる要因となるため、ルールインタフェースを使用しないコード、つまりRuleReqオブジェクトの設定やルール結果の使用などは、ルールインタフェースを取得する前やルールインタフェースを返した後に実行することを推奨する。

ただし、アプリケーションの特性によっては、異なることもあり、バッチアプリケーションのように他のアプリケーションとルールインタフェースを共有して使用しない環境では、毎回ルールを呼び出すたびにルールインタフェースを取得し返す必要はない。バッチアプリケーションのような場合は、アプリケーションが初期化される時にルールインタフェースを取得し、終了時に返す方が良い場合もある。

### I/Oアダプタ方式

ルールサービスの入力として使用されるデータは、様々な場所に位置している場合がある。あるデータは値オブジェクト(Value Object)やマップ、JavaBeansのようなビジネスアプリケーションのデータオブジェクトに存在している場合や、あるデータがデータベーステーブル上に存在している場合もある。さらに、あるデータは値オブジェクトまたはデータベーステーブルに存在するデータを加工して作成されている場合もある。

ルールには、そのルールを実行するために必要な項目が定義される。これらの項目内のある項目は、ルールが実行されるたびに使用されることもあれば、ある項目は特定条件を満たす時のみ使用されることもある。ルールアプリケーションはルールの構成を認識できないため、ルールの入力として定義された項目のうちどの項目が実際に使用される項目かを把握することができない。そのため、I/Oオブジェクト方式ではルールを呼び出す時にルールに定義されたすべての入力項目の値をRuleReqオブジェクトに格納する必要がある。もし、この入力項目の値がデータベースに存在する場合、その値の使用有無に問わず、アプリケーションはクエリを実行して値を読み込む必要がある。

ルールの入力項目が変更される場合にも、アプリケーションのメンテナンスに問題がある可能性がある。“age”および “gender”というフィールドを持つアプリケーションデータオブジェクトを想定し、あるルールで“age”のみを使用した判定を行う場合I/Oオブジェクト方式を使用するアプリケーションは、データオブジェクトのフィールドのうち“age”のみをRuleReqに設定してルールを呼び出す。しかし、ルールの判定規則が変更され“gender”を追加で使用する場合、アプリケーションコードの修正が必要である。アプリケーションデータオブジェクトにはすでに“gender”というフィールドはあるが、RuleReqに設定されていないため、このフィールドの値を設定するようにアプリケーションコードを修正する必要がある。ルールの入力項目が変更される時にアプリケーションを修正しないためには、最初からRuleReqに“age”のみでなく“gender”も設定する必要があるが、実際の環境ではアプリケーションデータオブジェクトには数多くのフィールドがあるため、現実的には不可能である。もしアプリケーションがルールにどの項目値を設定すべきか考慮せずに、ルールエンジンが必要とする項目をアプリケーションデータオブジェクトから取得することで、こうした問題を解決に効果がある。

I/Oアダプタ方式は、ルールの実行に必要な項目の値をルール呼び出し時に引き渡すのではなく、ルールエンジンがルールを実行している最中に必要な項目値をアプリケーションに問い合わせする方式である。この方式では、ルールを呼び出す際は呼び出すルールのコードと基準日のみを引き渡す。



[図 9 I/Oアダプタ方式のルールサービスの呼び出し]

[図 9 I/Oアダプタ方式のルールサービスの呼び出し]は、I/Oアダプタ方式でルールサービスを呼び出すプロセスを示す。1のプロセスで、アプリケーションはルールコードと基準日を引き渡してルールサービスを呼び出す。ルールエンジンはルールを実行し、実行中に必要な項目値がある場合、アプリケーション側にその項目の値を問い合わせする(2のプロセス)。アプリケーション側では、その項目の値を検索してルールエンジンに送る(3のプロセス)。ルールエンジンはルールの残りの部分を実行し、追加の項目が必要な場合、2と3のプロセスを繰り返す。ルールエンジンはルールの実行を終え、結果をアプリケーション側に設定する(4のプロセス)。

次の表は、[表 68 InnoRules.RulesClient.RuleInterface]のRuleInterfaceメソッドのうちのI/Oアダプタ方式であるExecuteメソッドのリストである。

[表 79 I/Oアダプタ方式のexecuteメソッド]

|  |
| --- |
| 1 namespace InnoRules.RulesClient;  2 {  3 public interface RuleInterface  4 {  5 void Execute ( String rule, ItemProvider itemProvider, ResultConsumer consumer,  6 int codetype, String date, Object ctx );  7 void Execute ( String rule, ItemProvider itemProvider, ResultConsumer consumer,  8 int codetype, String date, Object ctx, Trace trace,  9 HashMap<Object, Object> callctx );  10 …  11 } |

#### InnoRules.RulesClient.ItemProvider

InnoRules.RulesClient.ItemProviderインタフェースは、ルールエンジンに項目値を提供するために定義されたインタフェースである。ItemProviderには、項目のデータ型によってその値を提供するメソッドが定義されている。アプリケーションは次の図のようにアプリケーションデータオブジェクトの取得またはデータベースを取得して、項目値を提供するItemProviderオブジェクトを生成しExecuteメソッドにパラメータとして引き渡す。ItemProviderは、業務データが保存されている場所または使用されるオブジェクトの形式などによって様々な形式で実装される。しかし、アプリケーションが使用するデータオブジェクトの形式が同一の場合、1つのItemProvider実装クラスを作成してすべてのアプリケーションが共有することもできる。ルールエンジンは、ルールの実行中に項目の値が必要な場合にItemProviderインタフェースを利用して値を問い合わせする。



[図 10 ItemProviderダイアグラム]

次の表は、ItemProviderに定義されたメソッドである。

[表 80 InnoRules.RulesClient.ItemProvider]

|  |
| --- |
| 1 namespace InnoRules.RulesClient  2 {  3 public interface ItemProvider  4 {  5 Object GetAsNumber( String itemCode ) ;  6 Object GetAsString( String itemCode ) ;  7 Object GetAsBoolean( String itemCode ) ;  8 }  9 } |

次は、各メソッドに関する説明である。

Object GetAsNumber( String itemCode )

ルールエンジンがルール実行中に数値型の項目の値を問い合わせするために、このメソッドを呼び出す。項目のコードは、itemCodeパラメータに引き渡される。項目のコードは、RuleInterface.Executeに引き渡されたコード体系に基づくコードである。例えば、Executeメソッドにコード体系としてIDを指定した場合、itemCodeパラメータは項目のIDが引き渡される。

このメソッドでは、要求された項目の値をアプリケーションデータオブジェクトなどから抽出し返す必要がある。戻り値の形式はdoubleまたはdouble[]のみが許可される。そうでない場合は、ルールエンジンは例外(エラーコード1011)を投げてルールの実行が終了する。アプリケーションデータオブジェクトなどに保存された項目値がdoubleまたはdouble[]でない場合は、これらの形式で適切に変換する必要がある。変換できない場合は、RulesExceptionを投げることもある。double[]を返す場合は、配列の長さは1以上とする。そうでない場合は、ルールの実行が失敗して例外を投げる可能性がある(エラーコード1011)。

アプリケーションデータオブジェクトなどでこの項目の値が見つからない場合は、nullを返すことがある。この時、ルールエンジンは項目値が未入力とみなし、例外を投げてルールの実行は終了する(エラーコード1000)。アプリケーションデータオブジェクトなどから項目値を取得するプロセスでエラーが発生し項目値を提供できない場合は、RulesExceptionを投げることがある。この場合も、ルールエンジンは例外を投げルールの実行は終了する(エラーコード1000)。

Object GetAsString( String itemCode )

ルールエンジンがルール実行中に文字型の項目の値を問い合わせ実行するために、このメソッドを呼び出す。項目のコードは、itemCodeパラメータに引き渡される。項目のコードは、RuleInterface.Executeに引き渡されたコード体系に基づくコードである。このメソッドでは、要求された項目値をアプリケーションデータオブジェクトなどからStringまたはString[]の形式で返す。この形式以外のオブジェクトを返すと、ルールエンジンは例外(エラーコード1011)を投げ、ルールの実行が終了する。アプリケーションデータオブジェクトなどに保存された項目値StringまたはString[]でない場合は、適切に返す必要がある。返すことができない場合はRulesExceptionを投げることがある。String[]を返す場合、配列の長さは1より大きくし、配列内の値がnullである要素がないようにする。そうでない場合は、ルールの実行が失敗して例外を投げる可能性がある(エラーコード1011)。

アプリケーションデータオブジェクトなどからこの項目の値が見つからない場合はnullを返すことがあり、この場合のルールエンジンの動作方式はGetAsNumberメソッドと同一である。

Object GetAsBoolean( String itemCode )

ルールエンジンがルール実行中に論理型の項目の値をクエリ実行するために、このメソッドを呼び出す。項目のコードは、itemCodeパラメータに引き渡される。項目のコードは、RuleInterface.Executeに引き渡されたコード体系に基づくコードである。このメソッドでは、要求された項目値をアプリケーションデータオブジェクトなどからboolまたはbool[]の形式で返す。この形式以外のオブジェクトを返す場合、ルールエンジンは例外(エラーコード1011)を投げ、ルール実行が終了する。アプリケーションデータオブジェクトなどに保存された項目値boolまたはbool[]でない場合は、適切に返す必要がある。返すことができない場合はRulesExceptionを投げることがある。bool[]を返す場合の配列の長さは1より大きい必要がある。そうでない場合は、ルールの実行が失敗して例外を投げる可能性がある(エラーコード1011)。

アプリケーションデータオブジェクトなどからこの項目の値が見つからない場合はnullを返すことがあり、この時のルールエンジンの動作方式はGetAsNumberメソッドと同一である。

Hashtableをデータオブジェクトとして使用するアプリケーションがある場合、このアプリケーションは、次の表のようなデータをHashtableに保存する。プロパティのキーが項目のエイリアスと一致すると想定する。

[表 81 アプリケーションデータオブジェクトの事例]

|  |
| --- |
| TRDTYPE=Lease  AMT=250000000  PROPTYPE=APARTMENT |

次の表は、[表 81 アプリケーションデータオブジェクトの事例]のようなHashtableオブジェクトから項目値を提供するItemProvider実装の事例である。

[表 82 ItemProviderの実装事例 – HashtableProvider]

|  |
| --- |
| 1 class HashtableProvider : ItemProvider  2 {  3 private Hashtable datas;  4  5 public HashtableProvider(Hashtable datas)  6 {  7 this.datas = datas;  8 }  9  10 public Object GetAsNumber(String code)  11 {  12 if (!datas.ContainsKey(code))  13 return null;  14 Object value = datas[code];  15 try  16 {  17 double retvalue = Convert.ToDouble(value);  18 return retvalue;  19 }  20 catch (FormatException e)  21 {  22 throw new RulesException(e.ToString());  23 }  24 }  25  26 public Object GetAsString(String code)  27 {  28 if (!datas.ContainsKey(code))  29 return null;  30 return datas[code];  31 }  32  33 public Object GetAsBoolean(String code)  34 {  35 return null;  36 }  37 } |

このクラスはInnoRules.RulesClient.ItemProviderインタフェースを実装しており([1行目])、GetAsNumber、GetAs

String、GetAsBooleanの3つのメソッドを実装している。コンストラクタは、Hashtableオブジェクトをパラメータとして受け取る。アプリケーションは、自身が参照しているHashtableオブジェクトをコンストラクタに渡し、これはHashtableProviderのコンストラクタでメンバ変数として参照される。

GetAsNumberでは、([10~24行目])パラメータとして引き渡されたコードをキーとしてHashtableで検索する。コードに該当する値がない場合nullを返し、項目が入力されていないものとみなす。Hashtableに保存された値が正しい数字形式の文字列でない場合FormatExceptionが投げられ、GetAsNumberはそれをキャッチしてRulesExceptionに変換し再び投げる。

GetAsStringでは、([26~31行目])パラメータとして引き渡されたコードをキーとしてHashtableで検索された文字値を返す。コードに該当する値がない場合、nullを返し、項目が入力されていないと見なして処理する。例を簡略化するために、getAsBooleanは常にnullを返している。このItemProviderは、論理型の項目は常に入力されていないものとして処理する。

Hashtableのキーが項目のエイリアスと一致しているので、このHashtableProviderを使用する場合、RuleInterface.Executeの呼び出しの際にはコード体系としてエイリアスを使う。

#### InnoRules.RulesClient.ResultConsumer

InnoRules.RulesClient.ResultConsumerインタフェースは、ルールエンジンがルールの実行結果をアプリケーションデータ領域に設定するために定義されたインタフェースである。

ResultConsumerのメソッドを理解するためには、ルールのデータ型への理解が必要である。ルールは、1つ以上のリターン項目を持つ。各リターン項目には名前が付けられており、任意でエイリアスが付けられることもある。また、各リターン項目には数値型、文字型、論理型のうちの1つのデータ型が指定されている。ルールはリターン形式で、単一または多重のどちらか1つの形式を持つ。単一ルールは、ルール実行結果の行数が常に1つのルールである。一方、多重ルールはルール実行結果の行数が0または1つ以上の場合もあるルールである。

ルール実行結果のデータ型は、ルールのリターン項目の形式とリターン形式によって決定される。実行結果は1つ以上のカラムで構成されるが、各カラムはルールのリターン項目に対応する。カラム数は、ルールのリターン項目の数と同一である。カラム値のデータ型は、ルールが単一ルールまたは多重ルールによって異なる。カラム値のデータ型は、単一ルールの場合はルールリターン項目の形式が数値型、文字型、論理型の時にそれぞれdouble, String, boolであり、多重ルールの場合はそれぞれdouble[]、String[]、bool[]である。

ResultConsumerには、単一ルールと多重ルールに対し数値型、文字型、論理型リターン項目の結果を設定できる8つのsetメソッドが存在する。ルールサーバはルールの実行後、各結果カラム別にResultConsumer.setを呼び出してアプリケーションデータ領域にルールの結果を設定する。

ルールエンジンがルールの実行結果をアプリケーションデータ領域に設定するプロセスは、次の図のとおりである。次の図で例としてあげたルールは2つのリターン項目を保持しており、リターン項目の名前はRate、Amountでリターン項目の形式はそれぞれ文字型と数値型である。このルールのリターン形式が単一である場合、アプリケーションは、このルールを呼び出すためにRuleInterface.executeメソッドにResultConsumer実装オブジェクトを引き渡す。ResultConsumerのsetメソッドは、ルールエンジンが設定したカラム値を適切に変換し、アプリケーションデータ領域に保存するよう実装される。

ルールリターン項目が2つあるため、ルールの実行結果は2つのカラムを持つ。ルールのリターン形式が単一なので最初のカラム値の形式はString、2番目のカラム値の形式はdoubleになる。ルールエンジンは、最初のリターン項目の名前とカラム値をパラメータとし、これをアプリケーションデータ領域に設定する。設定が成功すると、2番目のリターン項目の名前とカラム値をパラメータとし、結果の2番目のカラム値をアプリケーションデータ領域に設定する。



[図 11 ResultConsumerダイアグラム]

次の表は、ResultConsumerインタフェースに定義されたメソッドである。

[表 83 InnoRules.RulesClient.ResultConsumer]

|  |
| --- |
| 1 namespace InnoRules.RulesClient  2 {  3 public interface ResultConsumer  4 {  5 void Set( String column, double value );  6 void Set( String column, double[ ] arr, int offset, int size );  7 void Set( String column, String value ) ;  8 void Set( String column, String[ ] arr, int offset, int size );  9 void Set( String column, bool value );  10 void Set( String column, bool[ ] value, int offset, int size )  11 }  12 } |

ResultConsumerには8つのSetメソッドが存在し、どのSetメソッドが呼び出されるのかはルールのリターン項目のデータ型によって決定される。次は、各メソッドに関する説明である。

void Set( String column, double value )

単一形式のルールで数値型のリターン項目の結果を設定するためのメソッドである。valueパラメータに結果が引き渡される。ResultConsumerは、アプリケーションデータ領域に値を設定中にエラーが発生した場合にRulesExceptionを投げることがある。

void Set( String column, double[ ] arr, int offset, int size )

多重形式のルールで数値型のリターン項目の結果を設定するためのメソッドである。arrパラメータが参照する配列のoffset番目からsize分のみの要素が有効な結果値である。アプリケーションデータ領域に値を設定中にエラーが発生する場合にRulesExceptionを投げることがある。

void Set( String column, String value )

単一形式のルールで文字型のリターン項目の結果を設定するためのメソッドである。valueパラメータに結果が引き渡される。ResultConsumerは、アプリケーションデータ領域に値を設定中にエラーが発生する場合にRulesExceptionを投げることがある。

void Set( String column, String[ ] arr, int offset, int size )

多重形式のルールで文字型のリターン項目の結果を設定するためのメソッドである。arrパラメータが参照する配列のoffset番目からsize分のみの要素が有効な結果値である。アプリケーションデータ領域に値を設定中にエラーが発生する場合にRulesExceptionを投げることがある。

void Set( String column, bool value )

単一形式のルールで論理型のリターン項目の結果を設定するためのメソッドである。valueパラメータに結果が引き渡される。ResultConsumerは、アプリケーションデータ領域に値を設定中にエラーが発生する場合にRulesExceptionを投げることがある。

void Set( String column, bool[ ] value, int offset, int size ) throws RulesException

多重形式のルールで論理型のリターン項目の結果を設定するためのメソッドである。valueパラメータが参照する配列のoffset番目からsize分のみの要素が有効な結果値である。アプリケーションデータ領域に値を設定中にエラーが発生する場合にRulesExceptionを投げることがある。

多重形式のルールでの各カラム値の配列の長さはすべて同一である。言い換えれば、カラムが複数の多重ルールを呼び出すと、その結果を設定するために呼び出されるsetメソッドのsizeパラメータはすべて同一である。

各setメソッドには、設定するカラムがどのリターン項目の結果なのかを指定するcolumnパラメータがある。columnパラメータの値は、ルールを呼び出す時に指定されたコード体系によって異なる方式で指定される。指定されたコード体系がIDまたは名前の場合はリターン項目の名前が指定され、コード体系がエイリアスの場合はリターン項目のエイリアスが指定される。リターン項目によっては、エイリアスが指定されない場合がある。コード体系がエイリアスの場合、エイリアスが指定されていないリターン項目にはsetメソッドは呼び出されない。

次の表は、[表 81 アプリケーションデータオブジェクトの事例]のようなHashtableオブジェクトにルール実行結果を設定するResultConsumer実装の事例である。

[表 84 ResultConsumerの実装事例 – HashtableConsumer]

|  |
| --- |
| 1 class HashtableConsumer : ResultConsumer  2 {  3 private Hashtable datas;  4 public HashtableConsumer(Hashtable datas)  5 {  6 this.datas = datas;  7 }  8  9 public void Set(String key, double value)  10 {  11 if (datas.ContainsKey(key))  12 datas[key] = value;  13 else  14 datas.Add(key, value);  15 }  16 public void Set(String key, String value)  17 {  18 if (datas.ContainsKey(key))  19 datas[key] = value;  20 else  21 datas.Add(key, value);  22 }  23 public void Set(String column, bool value)  24 {  25 /\* ignore \*/  26 }  27 public void Set(String key, double[ ] arr, int offset, int size)  28 {  29 StringBuilder buff = new StringBuilder();  30 for (int i = offset; size > 0; size--)  31 buff.Append(arr[i]).Append(',');  32 Set(key, buff.ToString());  33 }  34 public void Set(String key, String[ ] arr, int offset, int size)  35 {  36 StringBuilder buff = new StringBuilder();  37 for( int i = offset; size > 0; size-- )  38 buff.Append(arr[i]).Append(',');  39 Set(key, buff.ToString());  40 }  41 public void Set(String column, bool[ ] value, int offset, int size)  42 {  43 /\* ignore \*/  44 }  45 } |

このクラスはInnoRules.RulesClient.ResultConsumerインタフェースを実装しており([1行目])、8つのSetメソッドを実装している。コンストラクタは、Hashtableオブジェクトをパラメータとして受け取る。アプリケーションは自身が参照しているHashtableオブジェクトをコンストラクタに引き渡し、これはHashtableConsumerのコンストラクタでメンバ変数として参照される。

[9~15行目]は、単一ルールの数値型リターン項目の結果を設定するメソッドである。ルールエンジンから引き渡された数値を文字列に変換しプロパティに設定する。[16~22行目]は、単一ルールの文字型リターン項目の結果を設定するメソッドである。ルールエンジンから引き渡されたリターン項目名とString値をプロパティに設定する。

[27~33行目]は、多重ルールの数値型リターン項目の結果を設定するメソッドである。数値配列のoffset番目の要素からsize分のみの要素を‘,’を区切り記号として1つの文字列に変換している。変換された文字列は、リターン項目名をキーとしてプロパティに設定される。[34~40行目]は、多重ルールの文字型リターン項目の結果を設定するメソッドである。同様に、配列のoffset番目からsize分のみの文字列を1つの文字列に変換し、リターン項目名をキーとしてプロパティに設定する。

この例では、簡略化するために論理型リターン項目の結果は省略する。

#### executeメソッドのその他のパラメータ

次の表に、[表 79 I/Oアダプタ方式のexecuteメソッド]のExecuteメソッドを再び示す。

[表 85 I/Oアダプタ方式のExecuteメソッド]

|  |
| --- |
| 1 namespace InnoRules.RulesClient;  2 {  3 public interface RuleInterface  4 {  5 void Execute ( String rule, ItemProvider itemProvider, ResultConsumer consumer,  6 int codetype, String date, Object ctx );  7 void Execute ( String rule, ItemProvider itemProvider, ResultConsumer consumer,  8 int codetype, String date, Object ctx, Trace trace,  9 HashMap<Object, Object> callctx );  10 …  11 } |

Executeメソッドの最初のパラメータであるruleには、呼び出すルールのコードを引き渡す。コード体系によって、ルールのIDや名前またはエイリアスを使用できる。使用するコード体系は、4番目のパラメータであるcodetypeに定義する。codetypeの値として、[表 69 コード体系を指定する定数の値]の定数が使用できる。指定されたコード体系は、ルールコードのみでなく、ItemProviderのGetXXXメソッドに引き渡される項目コードやResultConsumerのsetメソッドに引き渡されるリターン項目の名前にも適用される。コード体系によってItemProviderおよびResultConsumerのメソッドに引き渡されるコードの類型については、「3.3.6.1 InnoRules.RulesClient.ItemProvider」と「3.3.6.2 InnoRules.RulesClient.ResultConsumer」をそれぞれ参照のこと。メソッドの2番目と3番目のパラメータは、ItemProviderオブジェクトとResultConsumerオブジェクトである。dateパラメータはルール呼び出しの基準日で、‘yyyy-MM-dd’または‘yyyyMMdd’形式の文字列、またはnullを指定できる。nullを指定すると、ルール呼び出しの基準日としてルールエンジンのシステム日付が使用される。traceパラメータはルール実行経路トレースの結果を作成するためのパラメータで、この文書では扱っていない。このパラメータをnullとして設定し、トレースを無効にすることができる。callctxオブジェクトもこの文書では扱っていないので、nullに設定することを推奨する。

#### ルールサービス呼び出し例

次の表は、I/Oオブジェクト方式のルールサービス呼び出しの例である[表 78 ルールサービス呼び出し例]のコードをI/Oアダプタ方式に変更したコードである。このコードには、[表 82 ItemProviderの実装事例 – HashtableProvider]と[表 84 ResultConsumerの実装事例 – HashtableConsumer]のHashtableProviderとHashtableConsumerが使用されている。この例では、アプリケーションがデータを保存するための標準クラスとしてHashtableを使用する。データ保存クラスとして他のクラスの値オブジェクトを使用する場合は、これらクラスのためのItemProviderとResultConsumerを作成する。

[表 86 ルールサービス呼び出し例]

|  |
| --- |
| 1  2 static String service()  3 {  4  5 Hashtable datas = new Hashtable();  6 datas.Add("AMT", 10000000);  7 datas.Add("TRDTYP", "A");  8  9 String message = null;  10 RuleInterface intf = null;  11  12 try  13 {  14 ItemProvider prov = new HashtableProvider(datas);  15 ResultConsumer cons = new HashtableConsumer(datas);  16  17 intf = ClusterManager.GetInterface();  18 intf.Execute("COMMISSION", prov, cons, Constants.CODETYPE\_ALIAS, "2013-07-01",  19 null);  20 }  21 catch (Exception e)  22 {  23 message = "ERROR: " + e.Message;  24 }  25 finally  26 {  27 if (intf != null)  28 intf.Close();  29 }  30 if (message==null)  31 {  32 String rate = Convert.ToString(datas["CMSRATE"]);  33 String commission = Convert.ToString(datas["CMSAMT"]);  34 message = "Commission is " + commission + ". Rate is " + rate + ".";  35 }  36 return message;  37 } |

例では、[5~7行目]で入力値をHashtableオブジェクトに適用している。[14行目]と[15行目]ではそれぞれHashtable

ProviderとHashtableConsumerを生成しており、そのコンストラクタにHashtableオブジェクトを引き渡している。[18行目]でExecuteメソッドが呼び出されている。このメソッド内でルールが呼び出され、ルールエンジンは項目値を調べるためにItemProviderに問い合わせする。ルールエンジンがルールの実行を終わらせると、その結果はResultConsumerのSetメソッドによりHashtableオブジェクトに適用されExecuteメソッドが終了する。つまり、Executeメソッドが終了した時点では既にHashtableオブジェクトにルール結果が格納されている。[30~35行目]は、Hashtableに適用された結果を使用するコードである。

# REST API

JavaやC/C#アプリケーション開発者は、ルールサービスを呼び出すためにJavaやC/C#のRule APIを使用することができる。しかし、JavaやC/C#アプリケーションでないため、Rule APIを使用できない場合やセキュリティポーリシーなどの事由で使用しない場合がある。InnoRulesはこの場合に備えて、HTTPメッセージを使用してルールを呼び出すREST APIを提供する。

REST APIはアプリケーション開発者がJSON形式のHTTPメッセージを使い、ルールサービスを呼び出す方法を提供する。REST APIが正常に動作するためには、ルールシステムでルールRESTサービスを有効にする必要がある。ルールRESTサービスを有効にする方法は「InnoRules Installation and Operation Guide」を参照のこと。

InnoRulesのREST APIで使用されるJSONは、JSONメッセージ規格を遵守する必要がある。JSONメッセージ規格は次のサイトを参照のこと。

* https://www.json.org

## HTTP Client

HTTP ClientとはInnoRules ServerとHTTPメッセージを送受信できるすべてのアプリケーション、またはシステムを意味する。HTTP ClientがRESTルールサービスを利用するためには次を遵守するべきである。

### URL

RESTルールサービスを利用するために、HTTP ClientはURLを指定してアクセスする必要がある。URLの構成要素の一部はルールシステム管理者がサービスウィザードを使いInnoRules Server上にREST Serviceを設定する時、指定する必要がある。

URLの構成は以下のとおりである。

* http://ip-address:port/innorules/services/rest/system-alias/rule-code[/date]
* ip-addressとport:InnoRules ServerのIPアドレスとポート
* system-alias:ルールシステム管理者が指定したシステムエイリアス
* rule-code:ルールシステム管理者が指定した呼出コードタイプに該当するコードを入力する。

詳細は「4.2.1 リクエストメッセージ」を参照のこと。

* date:ルール実行の基準日を入力する。基準日は省略可能であり、省略した場合InnoRules Serverではシステム日をルール実行基準日として使用する。日付形式はInnoRules Serverの設定時の日付形式と合わせる必要がある。
* 例)http://11.22.33.44:25802/innorules/services/rest/dev/samplerule/20200101

上記例はRESTサービス設定時、システムエイリアスをdev、呼出コードタイプをエイリアスに設定した場合、ルールのエイリアスがsampleruleであるルールを基準日20200101で呼び出している。

### HTTPヘッダー

送信メッセージにはHTTPヘッダーを指定しなければならない。

* Content-Type

HTTP送信メッセージのMIME Typeとcharsetを入力する必要がある。MIME Typeは、現在「application/json」をサポートしている。charsetはデフォルトで「ISO-8859-1」に設定されている。JSONメッセージにISO-8859-1文字セット以外の文字を使用する場合はcharsetを変更する必要がある。

次はContent-Typeの例である。

* Content-Type:application/json; charset=utf-8

### HTTPメソッド

ルールRESTサービスはPOSTメソッドで呼び出す必要がある。

## JSONメッセージ

ルールの入/出力情報はJSON形式のHTTPメッセージで送信される。ルールで使用するJSONメッセージの形式は次のとおりである。

### リクエストメッセージ

リクエストメッセージにはルール実行に必要な入力項目を入力する。

次の表はリクエストメッセージの例である。

[表 87 リクエストメッセージ例]

|  |
| --- |
| {  "codeType": "NAME",  "input": {  "item1":"value",  "item2":1.0,  "item3":[ "value1","value2", "value3" ],  "item4":[ 2.0, -3, 4.11 ]  }  } |

リクエストメッセージにはcodeTypeとinputオブジェクトを必ず入力する。

* codeType

codeTypeには呼出コードタイプを入力する。呼出コードタイプはルール呼出の時に入力するルールと項目値がどのような種類の値か(名前、コード、エイリアス)を識別するために利用される。codeTypeがない場合はルールコードで動作する。

* IDまたは0
* NAMEまたは1
* ALIASまたは2
* Input

Inputオブジェクトにはルール入力項目をkey:value形式で指定する。入力項目のkeyにはルール呼出コードタイプに該当する項目名称を入力し、valueはJSON仕様に沿って値を入力する。配列を入力する場合は文字型と数値型を混用してはいけない。

### レスポンスメッセージ

レスポンスメッセージにはルール実行結果が含まれる。レスポンスメッセージの形式はリクエストメッセージの形式と似ている。

次の表は単一結果値のレスポンスメッセージの例である。

[表 88 単一結果値のレスポンスメッセージ例]

|  |
| --- |
| {  "result": {  "res1": "aaaa",  "res2": 1.0,  "res3": false  }  } |

次の表は多重結果値のレスポンスメッセージの例である。

[表 89 多重結果値のレスポンスメッセージ例]

|  |
| --- |
| {  "result": {  "res4":[ “BBBBB”,"CCCCC", “111” ],  "res5":[ 2.0,-3.0,4.11 ],  "res6":[ true,false,true ]  }  } |

レスポンスメッセージにはresultオブジェクトが必ず返却される。resultオブジェクトはルールリターン項目を「key:value」形式で出力する。もし、呼出コードタイプに該当するルールリターン項目の名称が存在しない(例:ルールリターン項目のエイリアスが指定されていない)場合は、resultオブジェクトのkeyにルールリターン項目の登録順序を表す$0、$1、…、$nを出力する。

### エラーメッセージ

RESTルールサービスが正常に実行されない、RESTルールサービスまたはルールサーバにて実行エラーが発生した場合、RESTルールサービスはエラーメッセージを返却する。

次はエラーメッセージの例である。

[表 90 エラーメッセージ例]

|  |
| --- |
| {  "rulesException": {  "errorCode": "10001",  "message": "定義されていないルールです。 エイリアス: mainTestAA"  }  } |

エラーメッセージにはrulesExceptionオブジェクトが必ず返却される。rulesExceptionオブジェクトにはerrorCodeとmessageが存在する。errorCodeはルールシステムが返却するエラーコードであり、messageはエラーコードに対するエラーメッセージである。

### 特殊文字

JSONメッセージには一部の特殊文字を使用できる。次はJSONメッセージで使用できる特殊文字とその意味である。

* \b:Backspace (ascii code 08)
* \f:Form feed (ascii code 0C)
* \n:New line
* \r:Carriage return
* \t:Tab
* \":Double quote
* \\:Backslash character

# ルールアプリケーション開発イシュー

本章では、ルールアプリケーション開発中に発生するイシューと解決方法について説明する。

## 数値型イシュー

アプリケーションは、数値データを処理するために様々なデータ型を使用する。Cアプリケーションを例としてあげると、整数形式のデータ処理のためにchar、short、int、longなどを使用し、浮動小数点演算のためにはfloatやdoubleなどを使用する。または、プラットフォームでサポートされるDECIMAL64やDECIMAL128などのデータ型を使用することもある。Javaアプリケーションの場合は整数処理のためにbyte、short、int、longなどのデータ型を、浮動小数点処理のためにfloatやdoubleなどのプリミティブ型データ 型(Primitive Data Type)を使用し、BigIntegerやBigDecimalなどのオブジェクト形式も使用されることがある。どのようなデータ型を使用するかは、業務の要件によって決定される。

反面、Javaで作成されているルールエンジンは、数値データ処理のためにJavaのdouble型のみを使用する。単一形式を使用する事由は、主に業務の専門家たちによりルールが作成されるが、その専門家たちはIT専門ではないためである。彼らは、あるデータを保存するためにどのような形式を使用するかより、そのデータをどのように使用すべきかを考えることである。そのため、ルールエンジンは性能と処理可能な範囲を考慮して、IEEE754の倍精度実数形式であるdoubleを採用している。

IEEE754の倍精度実数は高い処理性能と広い処理範囲をサポートするが、特性上、小数部分で正確な値を表現できない場合もある。本章では、このような特性を考慮して、ルールエンジンから返される数値データをアプリケーションの業務用件に合わせて適切に変換する方法について提案する。

### hello-double.c

doubleの特性を理解するために、Cプログラムの例を説明する。ルールエンジンはJavaで作成されてはいるが、JavaとC/C++を使用するdoubleはすべて同一の形式なので\*5、ルールエンジンの特性理解には問題がない。

1. C/C++のdouble実装は、機器の仕様と関連がある。一部のメインフレーム段階の機器は基本的にIEEE754形式ではなく、hexadecimal形式の倍精度実数のサポートおよび、または2つの形式を選択できるようになっている。しかし、殆どのUNIXまたはPCの機器は、IEEE754形式の倍精度実数をサポートしている。

次の表は、hello-double.cのソースコードの例である。

[表 91 hello-double.c]

|  |
| --- |
| 1 #include <stdio.h>  2  3 int main( int argc, char\* argv[ ] )  4 {  5 double a = 0.1;  6  7 printf( "%lf\n", a );  8 return 0;  9 } |

変数aに0.1を割り当て、それを標準出力として出力している。このプログラムを実行した場合、次の値が出力される。

|  |
| --- |
| 0.100000 |

printf関数は、最初のパラメータとして出力するデータの種類と出力形式を受け取る。例では、double形式の値を出力するために%lf形式が指定されている。%lfはデフォルトで、小数点以下6桁まで出力するようになっている。この出力結果は、想定どおりの結果である。この例を修正し、小数点以下20桁まで出力するようにする。

[表 92 小数点以下20桁まで出力]

|  |
| --- |
| 1 #include <stdio.h>  2  3 int main( int argc, char\* argv[ ] )  4 {  5 double a = 0.1;  6  7 printf( "%.20lf\n", a );  8 return 0;  9 } |

このソースコードをコンパイルして実行後、次の値が出力される。

|  |
| --- |
| 0.10000000000000000555 |

0.1という値が出力されたが、小数点以下18桁から予測していない555が出力されている。その事由を理解するためには、10進法の表記を2進法の表記に変換するプロセスについて理解する必要がある。

### 2進法での表記

10進法で表記された整数を2進法の表記に変換するためには、商が0になるまで2で割り、それぞれの割り算の余りを逆に表記する。例えば、10進数11を2進法に変換する方法は、次のとおりである。

[表 93 10進数11を2進数に変換]

|  |
| --- |
| 11 ÷ 2 = 5 ...... 1(余り)  5 ÷ 2 = 2 …… 1  2 ÷ 2 = 1 …… 0  1 ÷ 2 = 0 …… 1  余りの逆順：1011(2) |

反対に、小数点以下の部分を2進法に変換するためには小数点以下がなくなるまで2を掛け、各掛け算の結果の整数を順に表記する。例えば、10進数0.125を2進法の表記に変換する方法は、次のとおりである。

[表 94 10進数0.125の2進法での表記]

|  |
| --- |
| 0.125 × 2 = 0.25 …… 0(整数部分)  0.25 × 2 = 0.5 …… 0  0.5 × 2 = 0.0 …… 1  整数部分の順番：.001(2) |

同様の方法で、10進数0.1を2進法の表記に変換すると次の表のようになる。

[表 95 10進数0.1の2進法での表記]

|  |
| --- |
| 0.1 × 2 = 0.2 …… 0 (1)  0.2 × 2 = 0.4 …… 0  0.4 × 2 = 0.8 …… 0  0.8 × 2 = 0.6 …… 1  0.6 × 2 = 0.2 …… 1  (1)から繰り返し  0.1(10) = …(2) |

10進法で有限小数である0.1は2進法では無限小数になるので、有限なビット内では2進法に表現できない。

### IEEE754倍精度実数型

IEEE754倍精度実数型は8バイトのデータで、次のような構造になっている。

[表 96 IEEE754倍精度実数型]

|  |  |  |
| --- | --- | --- |
| 1 bit sign(a) | 11 bit exponent(b) | 52 bit mantissa(c) |

各ビットは10進数に変換する時に、次のように使われる。

(a)は符号ビットで0は正の数、1は負の数を表し、(b)は1023のみバイアスされた指数\*6である。(c)は仮数で、前の1と共に有効数字を示すために使用される。

1. IEEE754の仕様に関する詳しい内容については、IEEEが公開しているIEEE754文書を参照のこと。

先に説明した0.1をdoubleが表現できる52の有効数字で表現した場合、次のとおりになる。

|  |
| --- |
| 0.1(10) = 0.00011001100110011001100110011001100110011001100110011010…(2)  ≒ 1.1001100110011001100110011001100110011001100110011010(2) \* 10(2) ^ 11111111100(2) |

これを再び10進数に変換すると、

|  |
| --- |
| 1.1001100110011001100110011001100110011001100110011010(2) \* 10(2) ^ 11111111100(2)  = 0.1000000000000000055511151231257827021181583404541015625(10) |

この数値が52の有効数字を利用して表現できる、0.1に最も近い近似値である。太く表示された部分と[表 92 小数点以下20桁まで出力]の結果を比較してみよう。

doubleは-1.79×10308~1.79×10308までの非常に広い範囲の数を表現できる。しかし、この範囲のすべての実数が表現できるという意味ではない。有効数字に使用されたビットの数が52なので、この範囲にわたって有効数字52に該当する精密度のみを保持し、その精密度を超える数を表現することはできない。

倍精度実数の精密度に関してだが、仮数部分によって有効数字が決定され、仮数部分の最下位の1ビットの増減による値の変化が倍精度実数の精密度になる。0.1の例では。0.1より大きいが最も小さい倍精度実数と0.1より小さいが最も大きい倍精度実数は、次のとおりである。太く表示されたビットの変化を確認する。

|  |
| --- |
| 1.1001100110011001100110011001100110011001100110011010(2) \* 10(2) ^ 11111111100(2)  = 0.1000000000000000055511151231257827021181583404541015625(10) ….(a)  1.1001100110011001100110011001100110011001100110011001(2) \* 10(2) ^ 11111111100(2)  = 0.09999999999999999167332731531132594682276248931884765625(10) ….(b) |

(a)と(b)は0.0000000000000000000000000000000000000000000000000001(2) \* 10(2) ^ 11111111100(2) = 0.00000000000000001387778780781445675529539585113525390625(10)のみの差がある。仮数部の最下位の1ビットが表す大きさは、そのdoubleの指数部の大きさによって変わる。ある倍精度実数の最下位の1ビットが表す大きさを、そのdoubleのULP(Unit in the Last PlaceまたはUnit of Least Precision)という。

### 演算の誤差・累積

すべてのdoubleの代数計算(足し算、引き算、掛け算、割り算)と平方根の結果は、真値と0.5ULP以内の誤差を持つ。真値との誤差が0.5ULP以内というのは、その値がdoubleで表現できる最も近い数であることを意味する。しかし、数回の演算により誤差が累積して、それ以上の誤差が発生する可能性もある。

次の例で、実際の演算の結果値を示す。

[表 97 浮動小数点数の代数計算と誤差の累積]

|  |
| --- |
| 1 public class Test  2 {  3 public static void main( String[ ] args ) throws Exception  4 {  5 double result = 9986.21 \* 100.0;  6 System.out.println( result );  7 }  8 } |

ユーザは、998621.0またはそれと同一値を意味する他のフォーマットの出力を想定するが、実際にこのサンプルを実行した結果998620.9999999999が出力されている。その原因としては次のとおりである。

まず9986.21は倍精度実数で、正確な値が表現できない。最も近い値9986.209999999999126885086297988891

6015625は、真値と約0.48ULP(<0.5ULP)の誤差がある。100.0はdoubleで正確な値が表現できる。2つの数を掛けると998620.999999999883584678173065185546875になり、998621.0と1ULPの誤差が発生して、この値は真値998621.0の1京分の1(1億分の1の1億分の1)である。

double演算で、特に固定小数点の表現をdoubleに変換して実行した演算では、計算の誤差を避けることはできない。実際、この誤差は無視および、補正可能なレベルである。但し、doubleの概念を正確に理解せず使用方法を誤った場合は、この誤差により問題が発生する可能性もある。

### 文字列への変換

doubleは、パソコンのメモリで8バイトを占めている2進データである。100、1、0.1、3.14のように、すべての規定値(ただし、doubleで表現可能な範囲)は直観的に理解できるテキストの形式ではなく、8バイトの2進データの形式でメモリに存在する。その2進データを出力するためには、それを文字列に変換するプロセスが必要になる。

次は、Cプログラムを例として説明する。

[表 98 double値を文字列で出力するCソースコード]

|  |
| --- |
| 1 #include <stdio.h>  2  3 int main( int argc, char\* argv[ ] )  4 {  5 double num = 1.0 - 0.667;  6 printf( "%lf\n", num );  7 printf( "%.20lf\n", num );  8 return 0;  9 } |

1.0から0.667の減算結果をprintf関数で出力している(0.667をdoubleに変換した値が実際は0.667ではないことについては、説明を省略する)。上のプログラムをコンパイルして実行した場合、次のような出力が得られる。

|  |
| --- |
| 0.333000  0.33299999999999996270 |

同一処理を実行するが、出力形式の指定によって異なる結果が得られる。最初のprintfでは特に出力形式を指定していないが、実際にはprintfのdoubleに対する基本出力形式で、小数点以下6桁までが文字列に変換されて出力されている。

次は、Javaプログラムの例である。

[表 99 double値を文字列で出力するJavaソースコード]

|  |
| --- |
| 1 public class Test  2 {  3 public static void main( String[ ] args ) throws Exception  4 {  5 System.out.println( 1.0 - 0.667 );  6 }  7 } |

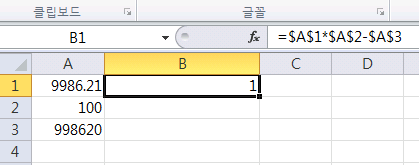
Cの例と同様に、1.0 – 0.667の結果を出力している。しかし、このJavaプログラムの出力結果には次のような差異がある。

|  |
| --- |
| 0.33299999999999996 |

System.out.println( double )の場合、sun.misc.FloatingDecimal.toJavaFormatString()というメソッドでdouble値を文字列に変換し、それをPrintStream.printlnで標準出力するようになっている(JDK 6.0基準)。つまり、doubleデータはメモリに“0.33299999999999996”という文字列の形式で存在するのではなく、FloatingDecimal.toJavaFormatString()により生成されたという意味である。基本形式ではなく他の形式に変換出力するためには、java.text.DecimalFormatなど異なる形式化メソッドを利用できる。

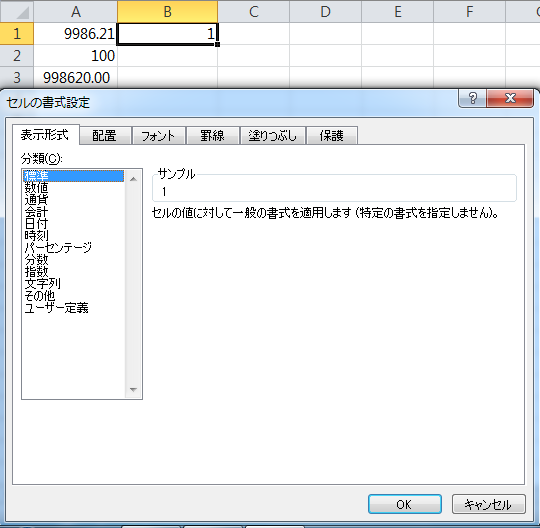
ルールエンジンから返された数値を、別の形式を指定せずに画面に出力することがある。その場合、上で説明したような原因によって意図しない桁数の数値が出力されることが多い。doubleはその特性上、近似値(恐らく真値とは殆ど差がない)を表現する。どの言語であれdoubleを出力(出力形式がコンソール、Web画面、メモリの場合でも)するためには文字列の形式に変換するロジックを通過するようになっており、変換される文字列の形式はユーザが別途指定しない場合、そのロジックの基本形式が適用される。したがって、出力形式の指定は最終的に出力される形式に詳しいプログラマの役割である。したがって、ルールエンジンから返された数値型の結果、またはルールエンジンとの関連がない場合、すべてのdouble値を計算のためではなく出力のために使用する場合は、必ず形式化を行う。

形式化に関して、Excelの例を示す。



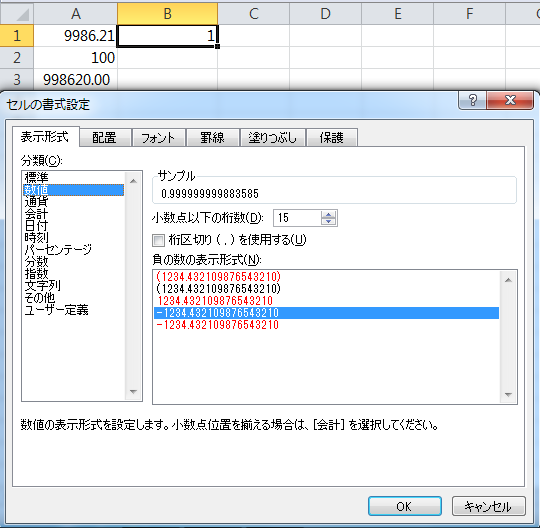
[図 12 9986.21 \* 100 – 998620]

[図 12 9986.21 \* 100 – 998620]のB1セルは99860.21 \* 100 - 998620の結果が1であることを示す。ところで、Excelの各セルはそのセルの値をどのような形式で表現するのかという書式を指定できる。次の図はB1セルの書式を示す。



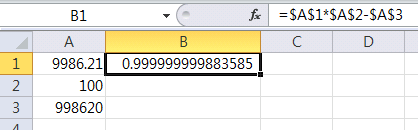
[図 13 書式 - 標準]

B1セルに対し編集者が特別な形式を指定しておらず、Excelが基本的に提供する形式でデータをセルに表示していることを表す。次の図は、このセルを小数点以下15桁まで表示するように設定することを示す。



[図 14 書式 - 小数点以下15桁]

その結果は次の図のとおりである。



[図 15 小数点以下15桁までの結果]

[図 15 小数点以下15桁までの結果]でB1セルは、「5.1.4 演算の誤差・累積」での結果と類似した値である。実際にExcelはセルに数値データを保存するためにInnoRulesと同様にIEEE754倍精度実数を使用し\*7、このデータを指定された(または基本)形式に合わせて変換してセルに表示する。\*8

1. http://support.microsoft.com/kb/78113
2. 単純に形式変換のみするのではなく、補正アルゴリズムを利用してある程度補正しているように見える。上記であげた例は、その方法で補正できない例外ケースである。DEV

### 整数変換および整数演算

CまたはJavaでは、倍精度実数と整数間の暗黙的な型変換(Implicit Type Casting)をサポートする。整数を実数に変換するのは、非常に大きい数でなければあまり問題ないかもしれない。しかし、倍精度実数を単純に整数に型変換しては、問題が起きる可能性がある。

[表 100 暗黙的な整数型変換の問題]

|  |
| --- |
| 1 import java.math.BigDecimal;  2  3 public class Test  4 {  5 public static void main( String[ ] args ) throws Exception  6 {  7 double result = 9986.21 \* 100.0;  8 int num = (int)result;  9  10 System.out.println( new BigDecimal( result ).toPlainString() );  11 System.out.println( num );  12 }  13 }  Output:  998620.999999999883584678173065185546875  998620 |

double型変数resultに保存された値と真値998621.0との誤差は1億分の1億分の1であるが、整数に型変換する際に小数点以下がなくなり誤差は100万分の1になっている。この誤差は、業務の種類によっては問題を起こす程度の誤差である。これは、double値を整数に変換する際に、単純にキャスト演算子を使用するのは一般的には誤りを意味する。

ルールエンジンから返された数値型の値、つまりdoubleをintに単純型変換して、その値の誤差が非常に大きくなり、それについて技術的なサポートの問い合わせをするケースがよくある。これはルールエンジンで誤った値を返したわけではなく、暗黙的な型変換で発生するエラーである。これに対する解決策は事例別に異なる。doubleを整数に型変換する場合は、通常次のうち1つに該当する。

* 計算の結果、小数部が0である。
* 計算の結果、小数部があるが、小数部は切り捨てられる。

前者の場合、小数点以下1桁で四捨五入(rounding)すれば良い。後者の場合、業務に影響を与えない桁(例えば、小数点以下10桁)で四捨五入した後、小数点以下1桁で切り捨て、または単純にint型変換をする方法もあり得る。

### 誤差の補正

4.1.6で、誤差の補正のために業務に影響を与えない小数点以下の桁で四捨五入する方法について説明した。小数点以下何桁で四捨五入するかは、業務によりそれぞれである。場合によっては四捨五入する小数点以下の桁が可変的で、特定の位置で四捨五入できないことがある。ここでは、それに対する解決方法について説明する。

一連の10進数計算がある場合に、計算過程の中で生じる誤差が発生し、最終誤差は8番目の有効数字の±0.5を超えないとする。計算で扱う数字が9桁以上の有効数字を持つ場合、計算の誤差により真値の相当部分が失われる。しかし、この計算で扱う数字の有効数字が8桁以下の場合、計算結果から有効数字の次の桁で四捨五入した時に正確な結果が得られる。次の表は、このような計算の例である。

[表 101 有効数字の桁数を利用した補正例]

|  |
| --- |
| 9986.21 × 100.0 = 998620.998 (有効数字9桁、誤差は8桁目の位置の0.2 (< 0.5) )  998620.998 ≒ 998621.00 (有効数字の9桁目の位置で、つまり8を四捨五入して真値を取得) |

この方式を、2進数表現である倍精度実数にも同様に適用してみる。倍精度実数は2進数53の有効桁を表現できる。2進数の有効桁53で約9 × 1015までの有効数字が表現できる。計算過程の誤差が8ULP以内の場合(相当大きい誤差である)、51番目の桁で四捨五入し50の有効数字(ビット)のみをサポートすることで誤差が補正できる。2進数50は0から約9 × 1015までの数字、つまり10進数の有効桁数15が処理できる。つまり、10進数の概念から考えると演算でサポートする有効桁数を15に制限すると、最終結果で16番目の有効数字で四捨五入して8ULP以内の誤差を補正できる。

次は、補正する関数のサンプルである。

[表 102 有効数字15桁で補正]

|  |
| --- |
| 1 private static final java.text.DecimalFormat FMT  2 = new DecimalFormat( ".###############E0" );  3  4 public static double norm( double num )  5 {  6 return Double.parseDouble( FMT.format( num ) );  7 } |

この補正を利用して「5.1.6 整数変換および整数演算」の例を修正し、その結果を出力すると次のとおりである。

[表 103 5.1.6の例を有効数字15桁で補正したJavaソースコードと結果]

|  |
| --- |
| 1 import java.math.BigDecimal;  2  3 public class Test  4 {  5 public static void main( String[ ] args ) throws Exception  6 {  7 double result = 9986.21 \* 100.0;  8 int num = norm( result );  9  10 System.out.println( new BigDecimal( result ).toPlainString() );  11 System.out.println( num );  12 }  13  14 public static double norm( double num )  15 …  16 }  Output:  998620.999999999883584678173065185546875  998621 |

補正した場合でも、最終出力のためには、必要に応じて適切な形式を指定する必要がある。補正は計算過程で発生する誤差の軽減のみであり、倍精度実数で表現できない値を表現することではないからである。例えば、どのような補正アルゴリズムも0.1を正確には表現できない。

アプリケーション開発者はルール呼び出しの結果の数値を単純な整数型への変換、または他の計算の入力で使用する前に上記のロジックを適用して補正できる。上記のロジックを適用したユーザ定義関数を作成し、ルール内で複雑な計算をする際に誤差を減らす目的で使用できる。この補正方法は数値を文字列に変換した後、再び数値に変換するため、頻繁に呼び出される場合は性能に影響を与える可能性がある。そのため、計算の重要なところでのみ使用することを推奨する。

### BigDecimal関連イシュー

分野によって、特に金融分野では固定小数点演算が有用な場合がある。JavaでもBigDecimalというタイプで固定小数点演算をサポートする。しかしInnoRulesでは、次のような事由から数値型処理のための基本データ型としてBigDecimalを使わない。

* 性能イシュー

ルールエンジンは高性能を要する場合が多い。double演算はハードウェアによって行われるため非常に高速であるが、BigDecimal演算はソフトウェア的に実行され低速である。また、BigDecimal演算は演算のたびに1つのBigDecimalオブジェクトを生成する。それにより、頻繁なGCを発生させる可能性がある。頻繁なGCは、ルールエンジンのようなJavaVM内で動作する他のアプリケーションの性能にも影響を与える可能性がある。

* 桁数の問題

BigDecimalは固定小数点を表現するため、それが持っている数字が小数点以下何桁まで表現しているかが重要である。特に割り切れない割り算では、小数点以下何桁まで計算するのか、その次の桁に対してはどう丸めるのかを指定する必要がある。単純に‘/’演算子を利用してn1/n2と表現していた計算式をDIV( n1, n2, scale, “ROUNDUP” )のように表現すると、ルールの表現が非常に複雑になる。

すべての演算で桁数を管理するため、最初の入力から桁数を明確にして、最終結果が計算されるまで桁数を保持する必要がある。したがって、計算の一部でのみBigDecimalを使用し、残りの部分ではdoubleを使用できない。

上記の事由から、Excelを含む殆どの一般的な目的で使用されるツールは内部的に倍精度実数を使用し、出力の際に適切な形式に変換する。

### 結論

InnoRulesは数値型表現のためにIEEE754標準の倍精度実数、つまりdoubleを使用する。doubleはその目的上広い範囲の値が表現できるが、場合によっては近似値しか表現できないケースもある。このような特徴のため、ルールの結果の数値を使用する開発者はその特徴を正確に理解し、出力の際に適切な文字列形式に変換する責任がある。

次は、数値型イシューに関するルール作成の注意事項である。

* ルールの結果の数値を出力に使用する場合、開発者はその数値を業務要件に合わせて適切にフォーマッティングする必要がある。
* ルールの結果を整数への型変換、四捨五入、切り上げ、切り捨てなどの関数のパラメータとして使用する場合は、パラメータとして引き渡す前に適切に補正する必要がある。

## ルールインタフェース

データベースアプリケーションからデータベースにアクセスしてクエリを実行するためには、コネクションプールからコネクションを取得し、使用終了後はコネクションプールに必ず返す。アプリケーションフレームワークを使用する構成では、大抵の開発者が明示的にコネクションの借用または返却処理はしないが、アプリケーションフレームワークがトランザクションの開始時点と終了時点に借用、返却処理を実行する。コネクションを借りた後、正常に返さない場合は、メモリリークやコネクション枯渇(Exhaustion)などの問題が発生する。

ルールインタフェースもデータベースコネクションと同一である。ルールインタフェースクラスタからインタフェースを借用し、ルールを呼び出した後にインタフェースを返さない場合、メモリリーク、コネクション枯渇などの問題が発生する。データベースアプリケーションとの差異は、一般的な常用アプリケーションフレームワークが自動でルールインタフェースの借用や返却を管理してないことのみである。ルールインタフェースの借用や返却は、開発者(アプリケーション開発者または共通モジュール開発者など)の責任下で行う必要がある。

次の表のようなパターンでルールを呼び出すことを推奨する。性能に影響を与えず、ルールインタフェースのリークを防ぐ。

[表 104 ルールインタフェースを借りるtry-finallyパターン]

|  |
| --- |
| 1 // Set input parameters  2  3 RuleInterface intf = ClusterManager.getInterface();  4 try  5 {  6 intf.execute(…); // invoke rule services  7 }  8 finally  9 {  10 intf.close();  11 } |

1つのルールインタフェースを使用して複数のメソッドでそのインタフェースを使用する場合、次の表のようにルールインタフェースをメンバ変数として共有するよりメソッドのパラメータとして引き渡すことを推奨する。その場合、ルールインタフェースを借りたところに返すことができ、インタフェースのリークを防ぐ。

[表 105 メソッド間のルールインタフェース共有パターン]

|  |
| --- |
| 1 public void method1() …  2 {  3 RuleInterface intf = ClusterManager.getInterface();  4 try  5 {  6 method2( intf, … );  7 method3( intf, … );  8 }  9 finally  10 {  11 intf.close();  12 }  13 }  14  15 private void method2( RuleInterface intf, … ) …  16 {  17 …  18 // Set input parameters  19 intf.execute( … ); // Invoke rule services  20 …  21 }  22  23 private void method3( RuleInterface intf, … ) …  24 {  25 …  26 // Set input parameters  27 intf.execute( … ); // Invoke rule services  28 …  29 } |

借りたルールインタフェースを複数のメソッドで使用するために、やむを得ずルールインタフェースオブジェクトをメンバ変数にした場合は、借用と返却に対するアプリケーション開発者の注意が必要である。

## ルール呼び出し回数のイシュー

データベースアプリケーションで、データベーステーブルに位置している同一データを複数のメソッドにより取得するために、各メソッドでクエリを実行しない。その代わり、一度クエリを実行したデータをアプリケーションのメモリ領域に位置させて、それを共有するように設計する。データベースサーバを利用したクエリより、アプリケーションメモリでの検索と参照が速いためである。

ルールアプリケーションでも複数のメソッドで同一ルールサービスの結果を使用するために各メソッドでルールを呼び出すより、ルールサービスの結果をアプリケーションのメモリで管理し、それを各メソッドで参照したほうが速いため、そのような方式を推奨する。例えば、商品の情報を取得するルールがあり、多様なメソッドでその情報を使用する必要がある場合は、商品情報を取得する共通メソッドの作成を推奨する。その共通メソッドはメモリに保存された該当の商品情報があれば、その情報を返し、そうでない場合はルールを呼び出して情報を取得する。ビジネスアプリケーションは、この共通メソッドを呼び出して商品サービスを受ける。

## アプリケーションフレームワークとの連動に関するイシュー

殆どの常用アプリケーションフレームワークまたはインハウスアプリケーションのフレームワークは、データ保存のため固有データ型を持っている。そのデータはJDKに含まれたMapの場合もあれば、フレームワークの固有のクラスの場合もある。

ルールエンジンは、アプリケーションフレームワークで使用するデータ型が把握できないため、ルール呼び出しのためには、アプリケーションフレームワークが使用するデータとルールエンジンが使用するデータ間の形式変換が必要である。そのためInnoRulesでは、com.innoexpert.rulesclient.ItemProviderとcom.innoexpert.-rulesclient.Result

Consumerインタフェースが用意されている(「1.2.6 I/Oオブジェクト方式」参照)。共通モジュール開発チームでこのインタフェースを利用し、フレームワークとルールエンジンのデータ変換モジュールを作成すれば、各開発者が直接データを変換する手間が省ける。

## 性能イシュー

ルールの実行はアプリケーショントランザクションの一部の区間で行われ、アプリケーショントランザクションの性能低下はルール実行時間が長いことが原因である場合もあれば、ルールの実行とはまったく関係がない場合もある。アプリケーショントランザクションの実行時間が長くかかる場合は、まずルールの性能に疑問を持ち、これに対する技術サポートを要求する前にアプリケーショントランザクションの区間別性能を確認することを推奨する。その時、ルール呼び出しのための入力値を構成する時間はルール実行時間に含まれない。

たまにアプリケーション全体に影響を与える要因がルールの実行性能に影響する場合がある。代表的な例がJavaのヒープ不足による問題である。アプリケーションまたはルールに特別な変化がないのに性能が低下した場合、アプリケーションのアーキテクトはJavaのヒープまたは外部的な要因を先に確認する必要がある。

## 業務実装におけるルールカバレッジのイシュー

ロジック管理の容易性、迅速性、可読性がInnoRulesルールエンジンのメリットである。反面、ルールエンジンはプログラミング言語と比較し、ロジック実装の柔軟性および性能が低い。アプリケーションアーキテクトは、それぞれのメリットを活かせるようにビジネスルールとプログラミング言語を適切に組み合わせる必要がある。

多くのプロジェクトで業務分野と開発ツールを同一視して該当の業務全体をルールで実装し、性能や保守作業の問題がよく発生する。開発段階で、ルール単位、業務単位、トランザクション単位で性能テストを行い、適切な性能および業務を分配してプロジェクトの遅延を防ぐことができる。

1. ユーザ定義関数API

組み込み関数のようなルール関数が作成できるAPIが提供される。このように作成されたルール関数をユーザ定義関数と言い、このAPIをユーザ定義関数APIと言う。ユーザ定義関数には名前が付与され、この名前を利用して組み込み関数のようにルール表現式の中で使用できる。ユーザ定義関数APIは関数の実行コードを作成する方法のみでなく、関数に名前を付与して関数に引き渡されたパラメータが関数の仕様(specification)に準拠しているかどうかをチェックする方法も提供する。

次のロジックをユーザ定義関数で実装すると効果的である。

* 変更されない複雑なロジック:ルールの表現を単純にし、可読性を向上させることができる。
* 既に作成されたプログラムモジュールの呼び出し:既に作成されたアプリケーションリソースをルールで活用できる。異種のプログラムモジュールも呼び出すことができる。ただし、JNI(Java Naming Interface)などを利用してJavaアプリケーションで呼び出す必要がある。
* 組み込み関数オーバーライド:組み込み関数の動作方式を変更する時に、組み込み関数と同一名称のユーザ定義関数を作成して、これがオーバーライドできる。

作成されたユーザ定義関数のコードは、ルールエンジンまたはルールビルダーサーバと同一JVMで実行されるため、JavaAPIのみ提供される。他のプログラミング言語で作成されたユーザコードを実行するためには、JNIを利用してユーザ定義関数の作成または、コールバックAPIを利用する必要がある。コールバックAPIはルールサービスAPIに含まれている。

# 必要なライブラリ

## ビルドライブラリ

ユーザ定義関数APIはinnorulesj.jarに含まれている。したがって、ユーザ定義関数を作成するためにはinnorulesj.jarをビルドパスに含む必要がある。作成される関数がサードパーティライブラリを使用する場合、該当のライブラリもビルダーパスに含む。

## ランタイムライブラリ

ユーザ定義関数はルールエンジンまたはルールビルダーサーバを含むルールランタイムシステム内で動作する。このランタイムには既にユーザ定義関数が動作できるイノルールズライブラリがすべて含まれているためユーザ定義関数を使用するために別のランタイムライブラリを追加する必要はない。ただし、ユーザ定義関数がサードパーティライブラリを使用する場合、ルールランタイムシステムのクラスパスにこのライブラリを追加する必要がある。

# 簡単な例

本章では階乗計算を行うユーザ定義関数を作成し、ルールランタイムシステムに反映させた後、テストするプロセスを簡単に説明する。

## 関数の原型(Prototype)

作成する関数の名前をFACTにし、2つのバリエーションを(variation)を設定する。最初のバリエーションは1つの数値をパラメータとして受け取り、1からその数まで整数の掛け算を行う関数である。2番目のバリエーションは2つの数値をパラメータとして受け取り、最初の値から2番目の値まで整数の掛け算を行う関数である。次の表は、関数のこの2つの原型を示す。

[表 106 FACT 関数の原型]

|  |
| --- |
| FACT( number )  FACT( start, end ) |

ここでnumber、start、endは数値型にする。[表 109 com.innoexpert.innorulesj.suite.udf.UserDefinedFunction]の文法は、ルール表現式の中で使用される。

## 関数のコード

次の表はFACT関数のコードである。

[表 107 com.innorules.apiguide.udf.FuncFACT]

|  |
| --- |
| 1 package com.innorules.apiguide.udf;  2  3 import java.util.\*;  4  5 import com.innoexpert.innorulesj.abstractcompiler.\*;  6 import com.innoexpert.innorulesj.exceptions.\*;  7 import com.innoexpert.innorulesj.suite.udf.\*;  8  9 public class FuncFACT implements UserDefinedFunction  10 {  11 public String getName()  12 {  13 return "FACT";  14 }  15  16 public Type check( Type[ ] params, List rules, List items ) throws CompileException  17 {  18 if( params.length == 1 )  19 {  20 if( !params[0].isNumber() )  21 throw new CompileException( "Usage: FACT( number ) or FACT( start, end )" );  22 }  23 else if( params.length == 2 )  24 {  25 if( !params[0].isNumber() || !params[1].isNumber() )  26 throw new CompileException( "Usage: FACT( number ) or FACT( start, end )" );  27 }  28 else  29 throw new CompileException( "Usage: FACT( number ) or FACT( start, end )" );  30  31 return TypeImpl.NUMBER;  32 }  33  34 public Object execute( Engine engine, Object[ ] params ) throws RuleRuntimeException  35 {  36 long start;  37 long end;  38 double arg0 = ((Double)params[0]).doubleValue();  39  40 if( params.length == 1 )  41 {  42 start = 1L;  43 end = (long)arg0;  44  45 if( (double)end != arg0 )  46 throw new UserDefinedFunctionException( "Parameters must be integer numbers." );  47 }  48 else // params.length == 2  49 {  50 double arg1 = ((Double)params[1]).doubleValue();  51 start = (long)arg0;  52 end = (long)arg1;  53  54 if( (double)start != arg0 || (double)end != arg1 )  55 throw new UserDefinedFunctionException( "Parameters must be integer numbers." );  56 }  57  58 if( start < 1 || end < 1 )  59 throw new UserDefinedFunctionException( "Arguments must be positive." );  60  61 long fact = 1L;  62 for( long n = start; n <= end; n++ )  63 fact \*= n;  64  65 return Double.valueOf( (double)fact );  66 }  67 } |

ユーザ定義関数のクラスはFuncFACTで([9行目])、com.innoexpert.innorulesj.suite.udf.User-DefinedFunctionインタフェースを実装している。このクラスには3つのメソッドがあり、getName([11行目])、check([16行目])、execute([34行目])がそのメソッドである。

getNameメソッドはユーザ定義関数の名前を返し、その名前は“FACT”である。

checkメソッドは、パラメータの数とデータ型をチェックしている。このメソッドは、ルールを保存する時のようにFACT関数を含むルール表現式の文法をチェックする際に呼び出される。「7.1 関数の原型(Prototype)」で説明したとおり、FACT関数は1つの数値型パラメータを受け取るものと2つの数値型パラメータを受け取るものがあった。[18~22行目]ではパラメータが1つの場合のデータ型のチェック、[23~27行目]ではパラメータが2つの場合のデータ型のチェックを行っている。階乗の意味を踏まえ、該当パラメータは自然数である必要がある。しかし、InnoRulesは整数型のデータ型をサポートしないため、チェック時点ではこのパラメータが整数なのかどうかを把握できない。したがって、checkメソッドではパラメータが数値型なのかどうかのみをチェックしている。

checkメソッドでは、関数の返却形式に関する情報も提供している。階乗の結果は数値のため、[31行目]でcom.inno

expert.innorulesj.abstractcompiler.TypeImpl.NUMBERを返している。関数のオーバーロードのように、パラメータの数または形式によって戻り値のデータ型を変えることもできる。例えば、(FACT関数の例としては適切ではないが)パラメータが1つのFACT関数は数値型を、パラメータが2つのFACT関数は文字型を返すように指定できる。

executeメソッド([34行目])では階乗計算を行っている。このメソッドは、ルールエンジンがこのユーザ関数を含むルール表現式を実行する際に呼び出される。paramsパラメータに関数のパラメータの値が引き渡される。checkメソッドでパラメータの数が1つまたは2つであることをチェックしたため、params配列の長さは1または2になる。また、checkメソッドでパラメータのデータ型が数値型であることをチェックしたため、param配列の要素はjava.lang.Doubleである。executeはこのパラメータの値を利用して階乗して返している([65行目])。executeメソッドでは、checkメソッドでチェックした結果とおり演算を行う必要がある。FACT関数の場合、checkメソッドで数値型を返すとなっているため([31行目])、必ずjava.lang.Double形式の値を返す。checkメソッドでパラメータの数とデータ型によってそれぞれ異なるデータ型を返す場合、executeメソッドはパラメータの数と形式に一致する正しいオブジェクトを返す必要がある。

checkメソッドではパラメータの値ではなくデータ型のみをチェックしているため、パラメータが整数なのかどうかはチェックできなかったが、executeメソッドではパラメータの値をチェックして整数でない場合はエラーを発生させることができる([46行目]、[55行目]、[59行目])。

## ヘルプファイル

次の表のようなXMLを利用して、関数のヘルプを作成できる。作成されたヘルプは、ルールビルダーのユーザ定義関数リストで照会できる。

[表 108 ヘルプ XML]

|  |
| --- |
| 1 <user-defined-function>  2 <Group>Math</Group>  3 <SimpleDescription>Calculates the factorial of a number or range.</SimpleDescription>  4 <Description>  5 <![CDATA[  6 <font size=2>  7 FACT( to )  8 Param 1: an integer number  9 Returns: The result of the factorial  10 FACT( start, end )  11 Param 1: an integer number, from value  12 Param 2: an integer number, to value  13 Returns: The result of the factorial  14 Example:  15 <BLOCKQUOTE>  16 1) FACT( 5 )  17 2) FACT( 2, 4 )  18 </BLOCKQUOTE>  19 </font>  20 ]]>  21 </Description>  22 </user-defined-function> |

Group要素([2行目])には、この関数がどのカテゴリに属する関数なのかを指定できる。SimpleDescription要素([3行目])には、この関数に関する簡単な説明が記述できる。その説明は、関数リストに関数の名前と一緒に表示される。Description要素([4~21行目])には、関数に関する詳細な説明が記述できる。ルールビルダーでその内容がどのように表示されるのかは「7.5 ヘルプの照会およびルールの作成」で確認する。

ヘルプファイルの作成は任意事項であり、ヘルプがない場合でもユーザ定義関数の使用には問題はない。しかし、ルール開発者の使用上の便宜のためにヘルプファイルを作成することを推奨する。

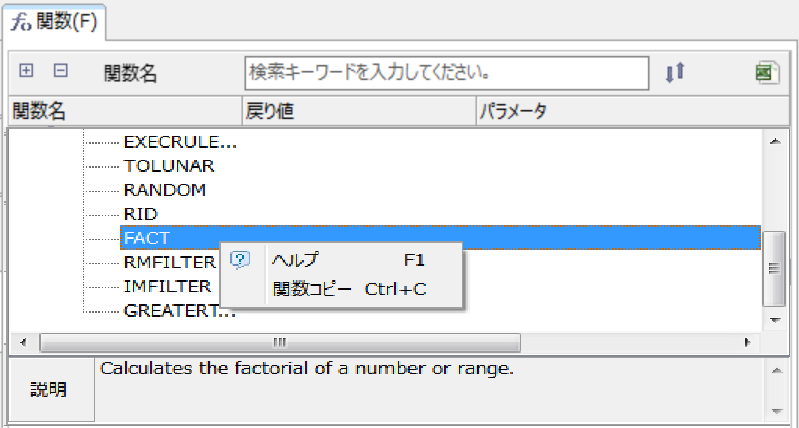
ヘルプファイルはユーザ定義関数のclassファイルがあるパッケージの位置にxmlファイルで存在する必要があり、ユーザ定義関数のクラスファイル名と同一でなければならない。ユーザ定義関数開発者はヘルプファイルとユーザ定義関数classファイルを1つのjarライブラリファイルに生成してユーザ定義関数を追加できる。

## 関数のインストールと設定

作成されたユーザ定義関数をルールシステムにインストールする方法については「InnoRules Installation and Operation Guide 2.6 アドオンウィザード」を参照のこと。

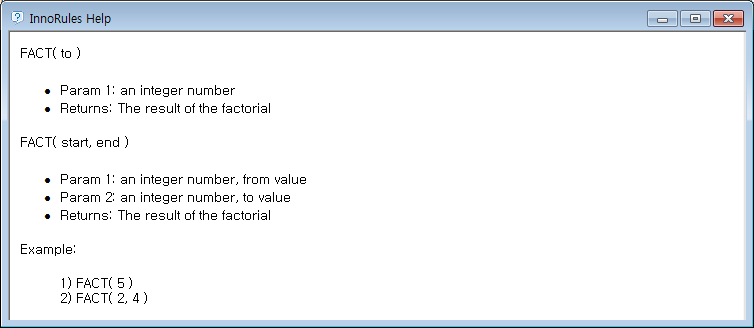
## ヘルプの照会およびルールの作成

ルールビルダーの[ユーザ定義関数]メニューで登録されたユーザ定義関数のリストを照会できる。次の図は、ユーザ定義関数リストの照会の画面である。「7.4 関数のインストールと設定」でインストールしたFACTユーザ関数がリストにあることが確認できる。



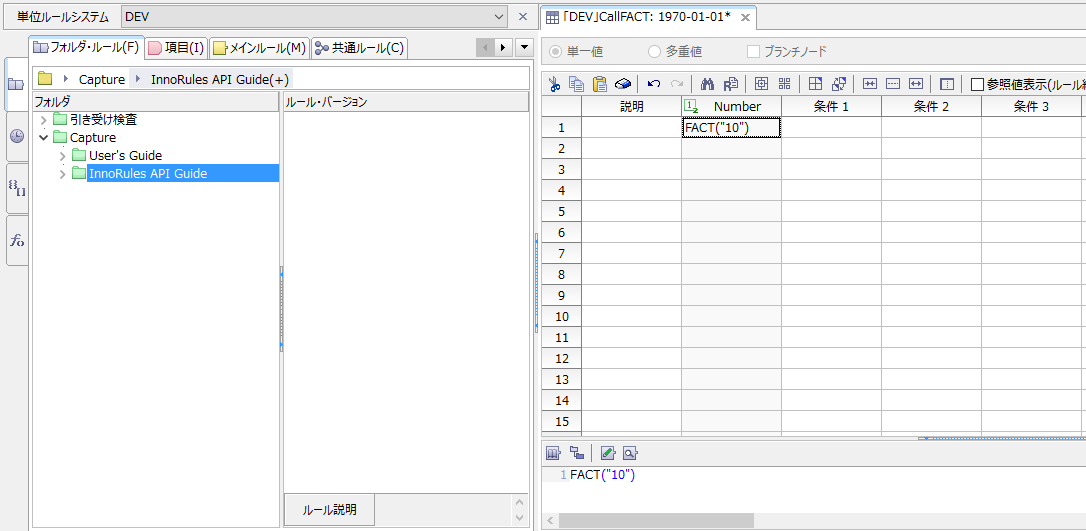
[図 16 ユーザ定義関数リスト]

区分列と説明列には、ヘルプファイルに記述した関数のグループと簡単な説明がある。[図 16 ユーザ定義関数リスト]のように、関数を選択し、マウスの右ボタンをクリックして詳細なヘルプを照会できる。次の図は、FACT関数のヘルプを照会した画面である。



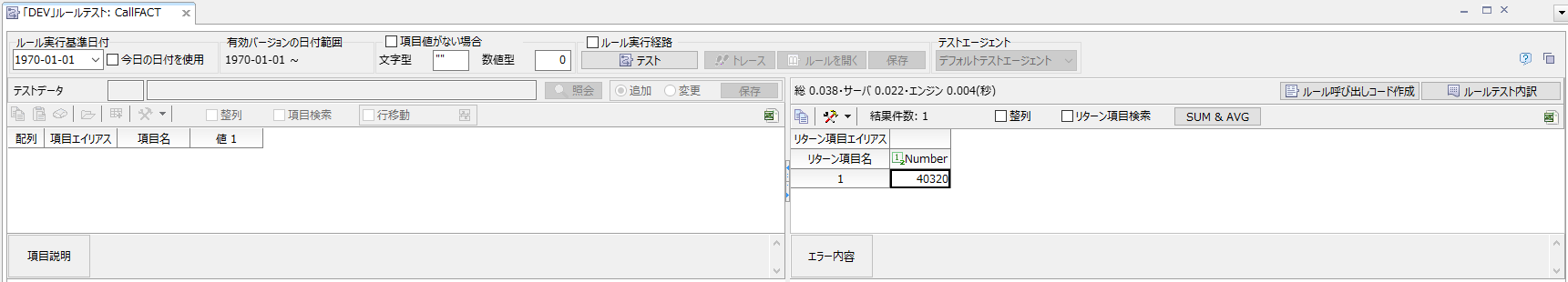
[図 17 FACT関数のヘルプ]

次の図は、FACT関数を使用するテーブルルールである。



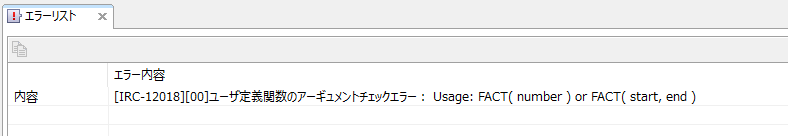
[図 18 FACT関数を使用するテーブルルール]

次の図は、[図 18 FACT関数を使用するテーブルルール]の{CallFACT}ルールをテストした結果である。



[図 19 {CallFACT}ルールのテストの結果]

[図 18 FACT関数を使用するテーブルルール]でFACT関数に正しくないパラメータを入力した場合、エラーが発生する。次の図は、[図 18 FACT関数を使用するテーブルルール]のFACT(10)をFACT( “10” )に変更して保存した時に発生するエラーメッセージである。



[図 20 正しくないFACT関数パラメータに対するエラーメッセージ]

# ユーザ定義関数APIクラス

本章では、ユーザ定義関数APIに関連するクラスについて説明する。

## com.innoexpert.innorulesj.suite.udf.UserDefinedFunction

ユーザ定義関数が実装するべきインタフェースである。次の表は、UserDefinedFunctionに定義されたメソッドを示す。

[表 109 com.innoexpert.innorulesj.suite.udf.UserDefinedFunction]

|  |
| --- |
| 1 package com.innoexpert.innorulesj.suite.udf;  2  3 import java.util.List;  4 import com.innoexpert.innorulesj.exceptions.\*;  5 import com.innoexpert.innorulesj.abstractcompiler.Type;  6  7 public interface UserDefinedFunction  8 {  9 String getName();  10 Type check( Type[ ] params, List rules, List items ) throws CompileException;  11 Object execute( Engine engine, Object[ ] params )  12 throws UserDefinedFunctionException;  13 } |

次は、各メソッドに関する説明である。

String getName()

ユーザ定義関数の名前を返す。この名前は、大小文字の区別なく、ルール表現式で使用される。例えば、このメソッドで“FACT”を返した場合、ルール表現式ではFACT、Fact、factなどの名前でこの関数にアクセスできる。関数名の最初の文字はアルファベットの大文字または小文字のみが可能で、2番目の文字以降はアルファベットの大文字、小文字、または‘\_’が使用できる。

このメソッドはルールランタイムシステムにユーザ定義関数を登録する際に1回のみ呼び出され、その後は呼び出されない。

Type check( Type[ ] params, List rules, List items ) throws CompileException

ルール表現式で使用されたユーザ定義関数の用法の正否をチェックし、関数のデータ型を返すメソッドである。ルールビルダーサーバがユーザ定義関数を含んだルール表現式の文法をチェックする際に、このメソッドを呼び出す。ルール表現式の文法チェックは、ルール保存や移管などの場合に行われる。

paramsパラメータにはルール表現式に記述されている関数パラメータ(argument)のデータ型が引き渡される。配列の各要素は関数のパラメータに対応され、配列の長さはパラメータの数と同一である。例えば、ルール表現式がFACT( 1, 2 )のように記述された場合、params配列の長さは2で、params[0]とparams[1]はそれぞれ1と2のデータ型である。データ型を表現するTypeインタフェースについては、「8.2 com.innoexpert.innorulesj.abstractcompiler.Type」で説明する。

このメソッドでパラメータのデータ型が関数の仕様に準拠しているかを確認する必要がある。準拠していない場合は、CompileExceptionを投げることがある。C++やJavaのオーバーロードのように、1つのユーザ定義関数が1つ以上のパラメータの組み合わせを持つことがある。例えばFACT関数の場合、FACT( 10 )というルール表現式を保存する際は、paramsパラメータの長さは1であり、最初の要素は数値型を表すTypeオブジェクトになる。FACT( 2, 10 )というルール表現式を保存する際は、paramsパラメータの長さは2であり、最初の要素と2番目の要素は数値型を表すTypeオブジェクトになる。checkメソッドでは、paramsをチェックしてFACT関数の明細に定義されたすべてのパラメータの組み合わせを許可できる。

このメソッドは、ユーザ定義関数が返す値のデータ型を返す。データ型は、Typeインタフェースを利用して記述する。返されたデータ型は、ルール表現式で関数のデータ型として使用される。例えば、FACT関数の場合、checkメソッドがTypeImpl.NUMBERを返すため(言い換えると、FACT関数が数値型であるため)

1 + FACT( 10 )

は数値型と数値型の+演算で処理される。反面、次の表現式

TONUMBER( FACT( 10 ) )

はTONUMBER関数がパラメータを受け取っていないため、文法エラーが発生する。関数が2つ以上のパラメータの組み合わせを許可する場合、関数のデータ型をパラメータの組み合わせによって異なるように指定することもできる。

ユーザ定義関数のexecuteでアプリケーションから引き渡された項目値の参照や他のルールを呼び出すことができる。項目値を参照して他のルールを呼び出す方法については、「com.innoexpert.innorulesj.suite.udf.Engine」を参照のこと。executeで項目値を参照する場合は、参照する項目のIDをitemsパラメータに追加する必要がある。executeで他のルールを呼び出す場合は、呼び出されるルールのIDをrulesパラメータに追加する必要がある。ルールビルダーサーバはcheckメソッドを呼び出した後、itemsとrulesパラメータを調査してユーザ定義関数がどのような項目とルールを使用するのかを確認し、その項目とルールが削除されないようにする。executeメソッドで使用された項目とルールをitemsとrulesに追加しなくてもユーザ定義関数は動作するが、ルールビルダーサーバでそれらが削除される可能性もある。

Object execute( Engine engine, Object[ ] params ) throws UserDefinedFunctionException

ユーザ定義関数の実行コードである。このメソッドは、ルールエンジンがユーザ定義関数を含んでいるルール表現式を実行する際に呼び出される。

engineはルールエンジンのプロキシ(proxy)オブジェクトとしてこれを利用し、executeメソッドの中でルールエンジンが提供するサービスを呼び出すことができる。Engineインタフェースについては、「8.3 com.innoexpert.innorulesj.suite.udf.Engine」で説明する。

paramsは、ユーザ定義関数のパラメータ(argument)が順に引き渡される。配列の要素は、次のクラスのうちの1つのオブジェクトである。

* java.lang.Double：数値型の値
* java.lang.String：文字型の値
* java.lang.Boolean：論理型の値
* com.innoexpert.innorulesj.runtime.SOValue：SO型(行数は1つだが、カラム数が2つ以上)
* com.innoexpert.innorulesj.runtime.MOValue：MO型(行数が0、または1つ以上)

params配列の要素はcheckメソッドで許可された組み合わせのみ引き渡される。FACT関数のcheckメソッドでは、関数のパラメータ(argument)が数値1つまたは数値2つではない場合、文法エラーが発生する。したがって、executeメソッドに引き渡されるparams配列は次の2パターンである。

* 長さが1で、その要素がjava.lang.Doubleオブジェクト
* 長さが2で、すべての要素がjava.lang.Doubleオブジェクト

executeメソッドは、ユーザ定義関数の結果値を返す必要がある。戻り値のデータ型は、checkメソッドで返した形式と一致する必要がある。そうでない場合は、java.lang.ClassCastExceptionが投げられる場合や想定外の動作を起こす可能性がある。返されるデータ型と返却値のクラスは、パラメータに適用される規則と同一である。FACT関数の場合、checkメソッドではFACT関数のパラメータのデータ型と関係なくTypeImpl.NUMBERを返したため、executeメソッドは必ずjava.lang.Doubleオブジェクトを返す必要がある。例えば、checkメソッドで2つ以上のパラメータの組み合わせを許可し、組み合わせによってそれぞれ異なるデータ型を返した場合、executeメソッドでも必ずパラメータの組み合わせに応じて該当する形式のオブジェクトの返却が必要である。

executeメソッドの処理中に例外を発生させる場合は、UserDefiendFunctionExceptionを投げることがある。UserDefinedFunctionExceptionについては、「8.6 例外クラス」で説明する。

ユーザ定義関数のオブジェクトは、関数の登録時に一度のみ生成される。すべてのcheckメソッドの呼び出しおよびすべてのexecuteメソッドの呼び出しは、同一オブジェクトに対して行われる。したがって、メンバ変数を使用することは推奨しない。

## com.innoexpert.innorulesj.abstractcompiler.Type

Typeインタフェースは、データ型を表現するのに使用されるインタフェースである。このインタフェースは、UserDefinedFunction.checkのパラメータおよび返却値として使用され、ユーザ定義関数のパラメータのデータ型と返却形式の定義に使用される。このインタフェースは数値型、文字型、論理型のような単純なデータ型のみでなく、SO型と呼ばれる1つまたは2つ以上のカラムで構成されるデータ型、MO型と呼ばれる複数の行を持つことができるデータ型も表現できる。

このインタフェースには、次の表のようなメソッドが定義されている。

[表 110 com.innoexpert.innorulesj.abstractcompiler.Type]

|  |
| --- |
| 1 package com.innoexpert.innorulesj.abstractcompiler;  2  3 public interface Type extends java.io.Serializable  4 {  5 byte getType();  6 boolean isNumber();  7 boolean isString();  8 boolean isBoolean();  9 boolean isSO();  10 boolean isMO();  11 int getTypeCnt();  12 byte[ ] getTypes();  13 } |

次は、各メソッドに関する説明である。

byte getType()

Typeオブジェクトが表すデータ型の定数を返す。データ型定数は、次の表のようにcom.innoexpert.inno

rulesj.runtime.Typesインタフェースに定義されている。

[表 111 データ型定数 - com.innoexpert.innorulesj.runtime.Types]

|  |
| --- |
| 1 package com.innoexpert.innorulesj.runtime;  2  3 interface Types  4 {  5 static final byte IR\_NUMBER = 1;  6 static final byte IR\_STRING = 2;  7 static final byte IR\_BOOLEAN = 3;  8 static final byte IR\_SO = 4;  9 static final byte IR\_MO = 5;  10 …  11 } |

IR\_NUMBER、IR\_STRING、IR\_BOOLEANは、それぞれ数値型、文字型、論理型を表す定数である。IR\_SOはSO型を表す定数で、IR\_MOはMO型を表す定数である。

boolean isNumber()

Typeオブジェクトの形式が数値型であるかを判定する。getType() == Types.IR\_NUMBERと同一である。

boolean isString()

Typeオブジェクトの形式が文字型であるかを判定する。getType() == Types.IR\_STRINGと同一である。

boolean isBoolean()

Typeオブジェクトの形式が論理型であるかを判定する。getType() == Types.IR\_BOOLEANと同一である。

boolean isSO()

Typeオブジェクトの形式がSO型であるかを判定する。getType() == Types.IR\_SOと同一である。

boolean isMO()

Typeオブジェクトの形式がMO型であるかを判定する。getType() == Types.IR\_MOと同一である。

int getTypeCnt()

Typeオブジェクトの形式のカラム数を返す。数値型、文字型、論理型のような単純データ型はカラム数が1つであるが、SO型またはMO型はカラム数が2つ以上の場合もある。

byte[ ] getTypes()

Typeオブジェクト形式のカラムのデータ型を返す。返される配列の長さはカラム数と同一で、配列の各要素の値は対応するカラムのデータ型定数である。カラムのデータ型は数値型、文字型、論理型のみが可能である。

使用頻度の高いデータ型のType定数を提供する。

* com.innoexpert.innorulesj.abstractcompiler.TypeImpl.NUMBER

数値型を表すType定数である。

* com.innoexpert.innorulesj.abstractcompiler.TypeImpl.STRING

文字型を表すType定数である。

* com.innoexpert.innorulesj.abstractcompiler.TypeImpl.BOOLEAN

論理型を表すType定数である。

* com.innoexpert.innorulesj.abstractcompiler.TypeImpl.NUMARR

数値型カラム1つのみで構成されるMO型を表すType定数である。

* com.innoexpert.innorulesj.abstractcompiler.TypeImpl.STRARR

文字型カラム1つのみで構成されるMO型を表すType定数である。

* com.innoexpert.innorulesj.abstractcompiler.TypeImpl.BOOLARR

論理型カラム1つのみで構成されるMO型を表すType定数である。

UserDefinedFunction.checkメソッドでユーザ定義関数のデータ型を返すために、[表 107 com.innorules.apiguide.udf.FuncFACT]の[31行目]のようにこの定数を使用できる。より複雑なデータ型は、Typeインタフェースを実装したクラスを作成して表現する。

## com.innoexpert.innorulesj.suite.udf.Engine

Engineインタフェースは、ルールエンジンが提供するサービスを呼び出すメソッドが定義されたインタフェースである。Engineオブジェクトは、UserDefinedFunction.executeにパラメータとして引き渡される。このインタフェースには、次の表のメソッドが定義されている。

[表 112 com.innoexpert.innorulesj.suite.udf.Engine]

|  |
| --- |
| 1 package com.innoexpert.innorulesj.suite.udf;  2  3 import java.util.Map;  4  5 import com.innoexpert.innorulesj.exceptions.RuleRuntimeException;  6 import com.innoexpert.innorulesj.repository.RuleMeta;  7  8 public interface Engine  9 {  10 Object executeRule( String id, com.innoexpert.rulesclient.Item... items ) throws RuleRuntimeException;  11 Object executeRuleByName( String name, com.innoexpert.rulesclient.Item... items ) throws RuleRuntimeException;  12 Object executeRuleByAlias( String alias, com.innoexpert.rulesclient.Item... items ) throws RuleRuntimeException;  13 Object execute( String expr, String codetype ) throws RuleRuntimeException;  14 String getSysProp( String name ) throws RuleRuntimeException;  15 Object queryItem( String id ) throws RuleRuntimeException;  16 Object queryItemByName( String name ) throws RuleRuntimeException;  17 Object queryItemByAlias( String alias ) throws RuleRuntimeException;  18 String queryCallParameter( String key, String option ) throws RuleRuntimeException;  19 RuleMeta queryRuleTypeByID( String id ) throws RuleRuntimeException;  20 RuleMeta queryRuleTypeByName( String name ) throws RuleRuntimeException;  21 RuleMeta queryRuleTypeByAlias( String alias ) throws RuleRuntimeException;  22 com.innoexpert.innorulesj.repository.Item queryItemTypeByID( String id ) throws RuleRuntimeException;  23 com.innoexpert.innorulesj.repository.Item queryItemTypeByName( String name ) throws RuleRuntimeException;  24 com.innoexpert.innorulesj.repository.Item queryItemTypeByAlias( String alias ) throws RuleRuntimeException;  25 ...  26 } |

次は、各メソッドに関する簡単な説明である。このメソッドはルールエンジンとルールリポジトリキャッシュに直接アクセスするため、ルールエンジンの動作やルールリポジトリキャッシュの構造などに対する理解が必要である。

Object executeRule( String id, com.innoexpert.rulesclient.Item... items ) throws RuleRuntimeException

idとして指定したルールを実行し結果を返す。このルール実行に必要な項目値をitemsパラメータに指定する。実行にあたり項目値が不要の場合、nullを指定する。idとitemsに使用するコード体系はIDである。

このメソッドを利用してあるルールを呼び出す場合は、UserDefinedFunction.checkメソッドのrulesパラメータに呼び出されるルールのIDを追加することを推奨する。詳細は、「8.1 com.innoexpert.innorulesj.suite.udf.UserDefinedFunction」を参照のこと。

Object executeRuleByName( String name, com.innoexpert.rulesclient.Item... items ) throws

RuleRuntimeException

nameとして指定したルールを実行し結果を返す。このルール実行に必要な項目値をitemsパラメータに指定する。実行にあたり項目値が不要の場合、nullを指定する。nameとitemsに使用するコード体系はNAMEである。

このメソッドを利用してあるルールを呼び出す場合は、UserDefinedFunction.checkメソッドのrulesパラメータに呼び出されるルールのIDを追加することを推奨する。詳細は、「8.1 com.innoexpert.innorulesj.suite.udf.UserDefinedFunction」を参照のこと。

Object executeRuleByAlias( String alias, com.innoexpert.rulesclient.Item... items ) throws

RuleRuntimeException

aliasとして指定したルールを実行し結果を返す。このルール実行に必要な項目値をitemsパラメータに指定する。実行にあたり項目値が不要の場合、nullを指定する。aliasとitemsに使用するコード体系はALIASである。

このメソッドを利用してあるルールを呼び出す場合は、UserDefinedFunction.checkメソッドのrulesパラメータに呼び出されるルールのIDを追加することを推奨する。詳細は、「8.1 com.innoexpert.innorulesj.suite.udf.UserDefinedFunction」を参照のこと。

Object execute( String expr, String codetype ) throws RuleRuntimeException

exprに指定された任意のルール表現式を実行し結果を返す。codetypeはルール表現式に使用されたコード体系である。ルール表現式実行時にコンパイルを行うため、場合によっては性能低下の原因となる。

String getSysProp( String name ) throws RuleRuntimeException

ルールリポジトリキャッシュに登録されているシステムプロパティを取り込む。代表的なシステムプロパティの名前は、次のとおりである。

* SYSTEM

ルールエンジンが実行されているルールシステムの名前

* TIMESTAMP

ルールリポジトリのタイムスタンプ。タイムスタンプは、ルールのデータベースが最後に修正された時刻である。

Object queryItem( String id ) throws RuleRuntimeException

入力された項目のうち、IDがidである項目値を返す。項目値が入力されていない場合は、例外が発生する。項目が数値型の場合はjava.lang.Doubleオブジェクト、文字型の場合はjava.lang.Stringオブジェクトが返される。

Object queryItemByName( String name ) throws RuleRuntimeException

入力された項目のうち、名前がnameである項目値を返す。項目値が入力されていない場合は、例外が発生する。項目が数値型の場合はjava.lang.Doubleオブジェクト、文字型の場合はjava.lang.Stringオブジェクトが返される。

Object queryItemByAlias( String alias ) throws RuleRuntimeException

入力された項目のうち、エイリアスがaliasである項目値を返す。項目値が入力されていない場合は、例外が発生する。項目が数値型の場合はjava.lang.Doubleオブジェクト、文字型の場合はjava.lang.Stringオブジェクトが返される。

String queryCallParameter( String key, String option ) throws RuleRuntimeException

ルール実行の基準日と同一ルールの呼び出しパラメータの値を返す。サポートされるkeyは、次のとおりである。

* CALLDATE

ルール呼び出し基準日であり、“-”を区切り文字にするyyyy-MM-dd形式が基本だが、optionパラメータに他の区切り文字を指定できる。例えば、optionに“/”を引き渡した場合、yyyy/MM/dd形式の日付が返される。

* CODETYPE

ルール呼び出しに使用されたコード体系であり、ID体系を利用した場合は“ID”、名前体系を利用した場合は“NAME”、エイリアス体系を利用した場合は“ALIAS”が返される。

RuleMeta queryRuleTypeByID( String id ) throws RuleRuntimeException

IDがidであるルール情報を返す。

RuleMeta queryRuleTypeByName( String name ) throws RuleRuntimeException

名前がnameであるルール情報を返す。

RuleMeta queryRuleTypeByAlias( String alias ) throws RuleRuntimeException

エイリアスがaliasであるルール情報を返す。

com.innoexpert.innorulesj.repository.Item queryItemTypeByID( String id ) throws RuleRuntimeException

IDがidである項目情報を返す。

com.innoexpert.innorulesj.repository.Item queryItemTypeByName(String name ) throws RuleRuntimeException

名前がnameである項目情報を返す。

com.innoexpert.innorulesj.repository.Item queryItemTypeByAlias( String alias ) throws RuleRuntimeException

エイリアスがaliasである項目情報を返す。

## com.innoexpert.innorulesj.runtime.SOValue

1行でカラムが2つ以上の値、つまりSO型の値を表現するためのクラスである。各カラムのデータ型は、必ず数値型、文字型、または論理型にする。それをイメージで表現すると、次のとおりである。

[表 113 SO型データ]

|  |  |  |  |
| --- | --- | --- | --- |
| カラム0 | カラム1 | … | カラムn-1 |
| カラム値 | カラム値 | カラム値 | カラム値 |

SOValueの主要メソッドは、次のとおりである。

[表 114 com.innoexpert.innorulesj.runtime.SOValue]

|  |
| --- |
| 1 package com.innoexpert.innorulesj.runtime;  2  3 import java.util.\*;  4 import com.innoexpert.innorulesj.exceptions.\*;  5  6 public final class SOValue implements java.io.Serializable  7 {  8 public SOValue();  9 public void mergeTail( double num );  10 public void mergeTail( String str );  11 public void mergeTail( boolean b );  12 public void mergeTail( SOValue so );  13 public int getNoCols();  14 public boolean isSameType( SOValue so );  15 public byte getColumnType( int colNo );  16 public double getNumber( int colNo ) throws RuleRuntimeException;  17 public String getString( int colNo ) throws RuleRuntimeException;  18 public boolean getBoolean( int colNo ) throws RuleRuntimeException;  19 } |

SOValue()

カラム数が0であるSO型のオブジェクトを生成する。

void mergeTail( doube num )

SO型の最後に数値型カラムを追加し、その値をnumで設定する。

void mergeTail( String str )

SO型の最後に文字型カラムを追加し、その値をstrで設定する。

void mergeTail( boolean b )

SO型の最後に論理型カラムを追加し、その値をbで設定する。

void mergeTail( SOValue so )

SO型の最後にSO型を追加する。追加されるカラムの順番と値は、soのカラムおよび値と同一である。

int getNoCols()

SO型のカラム数を返す。

boolean isSameType( SOValue so )

この値とsoパラメータのカラム数に対応する各カラムのデータ型を比較する。すべて一致する場合はTRUE、一致しない場合は、FALSEを返す。

byte getColumnType( int colNo )

colNo番目のカラムのデータ型を返す。戻り値は[表 111 データ型定数 - com.innoexpert.innorulesj.runtime.Types]を参照のこと。colNoは、0 basedのインデックスである。

double getNumber( int colNo ) throws RuleRuntimeException

colNo番目のカラムの数値を返す。colNoは、0 basedのインデックスである。カラムのデータ型が数値型でない場合は、RuleRuntimeExceptionが投げられる。

String getString( int colNo ) throws RuleRuntimeException

colNo番目のカラムの文字値を返す。colNoは、0 basedのインデックスである。カラムのデータ型が文字型でない場合は、RuleRuntimeExceptionが投げられる。

boolean getBoolean( int colNo ) throws RuleRuntimeException

colNo番目のカラムの論理値を返す。colNoは、0 basedのインデックスである。カラムのデータ型が論理型でない場合は、RuleRuntimeExceptionが投げられる。

次は、2つのカラムを持つSO値を生成する例である。カラムのデータ型は、数値型、文字型とし、それぞれ値に「5.0」、「“ABC”」と設定している。

|  |
| --- |
| SOValue so = new SOValue();  so.mergeTail( 5.0 );  so.mergeTail( “ABC” ); |

## com.innoexpert.innorulesj.runtime.MOValue

0行または1行で、カラムが1つ以上の値、つまりMO値を表現するためのクラスである。各カラムのデータ型は、必ず数値型、文字型、または論理型にする。それをイメージで表現すると、次のとおりである。

[表 115 MO型データ]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | カラム0 | カラム1 | … | カラムn-1 |
| 行0 | 値0,0 | 値1,0 |  | 値n-1,0 |
| … |  |  |  |  |
| 行m-1 | 値0,m-1 | 値1,m-1 |  | 値n-1,m-1 |

MOValueの主要メソッドは、次のとおりである。

[表 116 com.innoexpert.innorulesj.runtime.MOValue]

|  |
| --- |
| 1 package com.innoexpert.innorulesj.runtime;  2  3 import java.util.\*;  4 import com.innoexpert.innorulesj.exceptions.\*;  5  6 public final class MOValue implements java.io.Serializable  7 {  8 public MOValue()  9 public int getNoRows()  10 public int getNoCols()  11 public Column getColumn( int i )  12 public void mergeTail( Column col ) throws RuleRuntimeException  13 }  14  15 public class Column implements java.io.Serializable  16 {  17 public int getNoRows()  18 public byte getType()  19 }  20  21 public final class NumColumn extends Column  22 {  23 public NumColumn()  24 public NumColumn( double[ ] arr )  25 public NumColumn( double[ ] arr, int size )  26 public double getNumber( int row )  27 public void append( double num )  28 }  29  30 public final class StrColumn extends Column  31 {  32 public StrColumn()  33 public StrColumn( String[ ] arr )  34 public StrColumn( String[ ] arr, int size )  35 public String getString( int row )  36 public void append( String str )  37 }  38  39 public final class BoolColumn extends Column  40 {  41 public BoolColumn()  42 public BoolColumn( boolean[ ] arr )  43 public BoolColumn( boolean[ ] arr, int size )  44 public boolean getBoolean( int row )  45 public void append( boolean b )  46 } |

MOValue()

カラム数が0で行数が0であるMO型のオブジェクトを生成する。

int getNoRows()

MO型の行数を返す。

int getNoCols()

MO型のカラム数を返す。

Column getColumn( int colNo )

colNo番目のカラムを返す。ColumnはMO型の各カラムを表現するクラスで、後に説明する。colNoは、0ベースのインデックスである。

void mergeTail( Column col ) throws RuleRuntimeException

MO型のカラムを追加する。追加するカラムの行数がMO型の行数と一致しない場合は、RuleRuntimeExceptionが投げられる。

次は、Columnクラスの主要メソッドに関する説明である。

int getNoRows()

カラムの行数を返す。

byte getType()

カラムのデータ型を返す。

NumColumn、StrColumn、BoolColumnは数値型、文字型、論理型カラムを表現するクラスで、Columnを継承している。各クラスは類似しているため、ここではNumColumnのみを説明する。

NumColumn()

行の長さが0である数値型カラムを生成する。

NumColumn( double[ ] arr )

arrを利用して数値型カラムを生成する。カラムの行の長さはarr.lengthで、各行の値は対応するarrの値と同一である。

NumColumn( double[ ] arr, int size )

arrを利用して数値型カラムを生成する。カラム行の長さはsizeで、arr配列の最初のsizeの要素のみがカラム値として使用される。

double getNumber( int row )

row番目の行数値を返す。rowは、0ベースのインデックスである。

void append( double num )

カラムの最後の行に数値を追加する。

次は、2つのカラムで構成されたMO型を生成する例である。カラムのデータ型は数値型、文字型の順になる。

|  |
| --- |
| MOValue mo = new MOValue();  mo.mergeTail( new NumColumn( new double[ ] { 1.0, 2.0 } ) );  mo.mergeTail( new StrColumn( new String[ ] { “ABC”, “abc” } ) ); |

## 例外クラス

### com.innoexpert.innorulesj.exceptions.CompileException

CompileExceptionは、UserDefinedFunction.checkメソッドで関数パラメータの有効性を検証する際、パラメータのデータ型が関数の定義と一致しない場合に発生する。次の表はコンストラクタである。

[表 117 com.innoexpert.suite.udf.CompileException]

|  |
| --- |
| 1 package com.innoexpert.innorulesj.exceptions;  2  3 public class CompileException extends RuleRuntimeException  4 {  5 public CompileException( String message, Throwable t );  6 public CompileException( String message );  7 } |

### com.innoexpert.suite.udf.UserDefinedFunctionException

UserDefinedFunctionExceptionは、ユーザ定義関数の実行ルーチンであるUserDefinedFunction.executeの実行中にエラーが発生した際に投げられる例外クラスである。次の表はコンストラクタである。

[表 118 com.innoexpert.suite.udf.UserDefinedFunctionException]

|  |
| --- |
| 1 package com.innoexpert.innorulesj.suite.udf;  2  3 import com.innoexpert.innorulesj.exceptions.RuleRuntimeException;  4  5 public class UserDefinedFunctionException extends RuleRuntimeException  6 {  7 public UserDefinedFunctionException( String message, Throwable t );  8 public UserDefinedFunctionException( String message );  9 } |

# ユーザ定義関数の開発フレームワーク

本章では、ユーザ定義関数の開発に必要なテストおよびデバッグ環境であるユーザ定義関数の開発フレームワーク(以下、UDFフレームワーク)について説明する。

## ビルドライブラリ

UDFフレームワークAPIは、udf-unit.jarに含まれている。したがって、ユーザ定義関数の開発フレームワークを使用するためには、udf-unit.jarをビルドパスに含める必要がある。ユーザ定義関数APIに必要なビルドライブラリは「6. 必要なライブラリ」を参照のこと。

## ランタイムライブラリ

UDFフレームワークAPIは開発フレームワークで、ランタイムライブラリを含む必要はない。

## ユーザ定義関数フレームワークAPIクラス

本章では、UDFフレームワークAPIに関連するクラスについて説明する。

### com.innoexpert.innorulesj.suite.udfunit.UdfUnit

UDFフレームワークを使用するためのクラスである。次の表は、UdfUnitに定義されているメソッドを示す。

[表 119 com.innoexpert.innorulesj.suite.udfunit.UdfUnit]

|  |
| --- |
| 1 package com.innoexpert.innorulesj.suite.udf;  2  3 import java.lang.RuntimeException;  4 import java.util.\*;  5 import com.innoexpert.innorulesj.abstractcompiler.\*;  6 import com.innoexpert.innorulesj.exceptions.\*;  7 import com.innoexpert.innorulesj.runtime.\*;  8 import com.innoexpert.innorulesj.suite.udf.\*;  9 import com.innoexpert.rulesclient.RuleReq;  10  11 public class UdfUnit  12 {  13 public static void configureProxy( String address, int port,  14 String id, String password ) throws IllegalStateException  15 public static UdfUnit check( Class<? extends UserDefinedFunction> clazz,  16 Object... args ) throws CompileException, IllegalArgumentException  17 public Type getReturnType()  18 public Object execute( RuleReq req, int codetype, Object... args )  19 throws RuleRuntimeException  20 public Object execute( Object... args ) throws RuleRuntimeException  21 public void close()  22 } |

次は、各メソッドに関する説明である。

public static void configureProxy( String address, int port, String id, String password ) throws

IllegalStateException

UDFフレームワークでルールエンジンのプロキシ(proxy)オブジェクトのサービスを使用するために、リモートのTCPコネクタサーバへの接続設定をする。

addressパラメータにはTCPコネクタサーバのアドレス、portパラメータにはTCPコネクタサーバのサービスポートを引き渡す。idとpasswordパラメータはUDFフレームワークの認証手続きのために予約されたパラメータで、現在のバージョンでは使用されない。

public static UdfUnit check( Class<? extends UserDefinedFunction> clazz, Object... args ) throws CompileException, IllegalArgumentException

ユーザ定義関数のcheckメソッドを呼び出し、UdfUnitを返すメソッドである。

clazzパラメータにはデバッグするユーザ定義関数のクラスを引き渡す。

argsパラメータは、ユーザ定義関数のパラメータ(argument)を順に引き渡す。例えば、「7. 簡単な例」であるFACT関数を使用する場合、clazzパラメータにcom.innorules.apiguide.udf.FuncFACTとargsパラメータにnew Object[ ] { new Double( 1.0 )とnew Double( 2.0 ) }を使用してcheckメソッドを呼び出す。

public Type getReturnType()

ユーザ定義関数のリターン値のデータ型を返す。

public Object execute( RuleReq req, int codetype, Object... args ) throws RuleRuntimeException

ユーザ定義関数のexecuteメソッドを呼び出す。ルールエンジンのプロキシオブジェクトのサービスを使用する場合のメソッドである。

reqパラメータは、ルールエンジンのプロキシ(proxy)オブジェクトのサービスを使用するための入力項目を引き渡す。

codetypeパラメータは、reqパラメータの呼び出し方式(ID、Alias、Name)を指定する。

argsパラメータは、ユーザ定義関数のパラメータ(argument)を順に引き渡す。

public Object execute( Object... args ) throws RuleRuntimeException

ユーザ定義関数のexecuteメソッドを呼び出す。ルールエンジンのプロキシオブジェクトのサービスを使用しない場合のメソッドである。

argsパラメータは、ユーザ定義関数のパラメータ(argument)を順に引き渡す。

public void close()

UDFフレームワークのリモートTCPコネクタサーバへの接続を終了する。

## 簡単な例

次の表は、UDFフレームワーク使用についての例である。

[表 120 com.innoexpert.innorulesj.suite.udfunit.UdfUnit]

|  |
| --- |
| 1 package test.innorules.udfunit;  2  3 import com.innoexpert.innorulesj.udfunit.UdfUnit;  4  5 public class UDFFuncFACT  6 {  7 public static void main( String[ ] args ) throws Exception  8 {  9 UdfUnit.configureProxy( "localhost", 25800, "", "" );  10 test();  11 }  12  13 public static void test() throws Exception  14 {  15 Object[ ] args = new Object[ ] { new Double( 1.0 ), new Double( 3.0 ) };  16 UdfUnit unit = UdfUnit.check( com.innorules.apiguide.udf.FuncFACT.class, args );  17  18 for( int i = 0; i < 10; i++ )  19 System.out.println( unit.execute( args ).toString() );  20  21 unit.close();  22 }  23 } |

UDFFuncFACTクラスでフレームワークを利用し、FACT関数を開発する。

UdfUnit.configureProxyメソッドを利用([9行目])してローカルの25800ポートのTCPコネクタサーバを設定する。

FACT関数に引き渡すパラメータ(argument)を順に指定([15行目])する。

UdfUnit.check([16行目])を利用してFACT関数のcheckメソッドを実行し、UdfUnitを返却させる。

UdfUnitを利用してFACT関数のexecuteメソッドを実行([19行目])する。

TCPコネクタサーバの連結([21行目])を終了する。

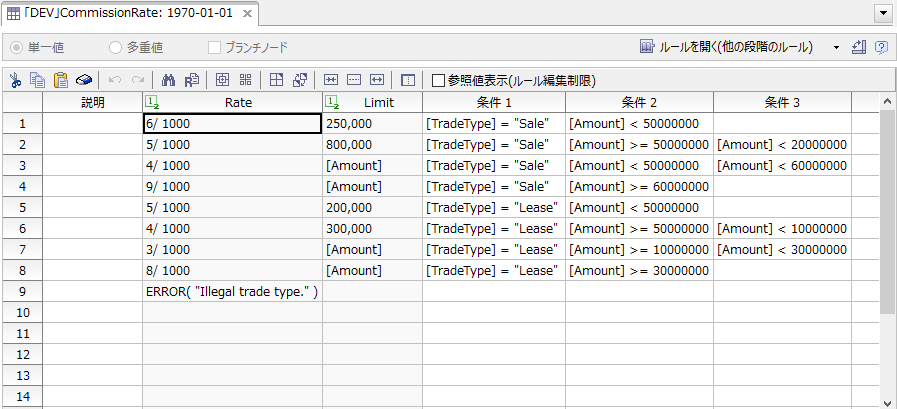
# 定数ルールとユーザ定義関数

この章では定数ルールに関してユーザ定義関数を作成する時の留意事項について説明する。

## 定数ルール

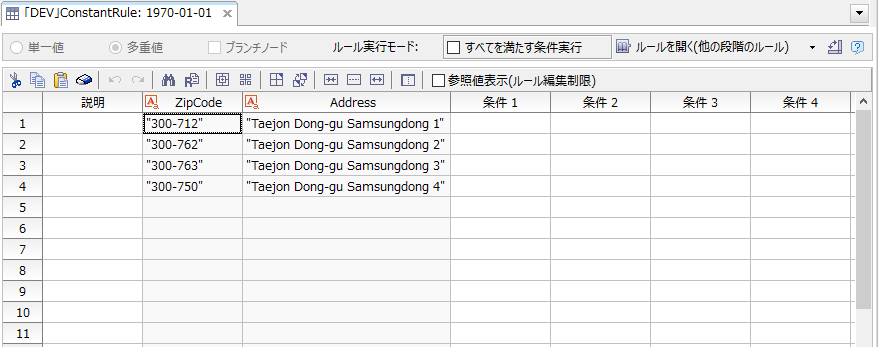
次の図に{Commission Rate}を示す。このルールは[Trade Type]と[Amount]という項目を使用する。

このルールは呼び出す度に、この2つの項目の値によって異なる結果を出力する可能性がある。



[図 21 非定数ルール]

一方、次の図の{ConstantRule}は、いずれの項目も使用しておらず、他のルールを呼び出していない。このルールは別のルールトランザクションで呼び出しても常に同一結果を出力する。



[図 22 定数ルール]

[図 22 定数ルール]のようにどのルールや項目も使用してないルールを定数ルールという。定数ルールは入力値に関係なく、常に同一結果を出すため、ルールエンジンは初回のみ定数ルールを実行し、その結果をキャッシュに保管する。以降のトランザクションで、その定数ルールが呼び出されるとルールエンジンはルールを再実行せずにキャッシュに保管された結果を使用する。

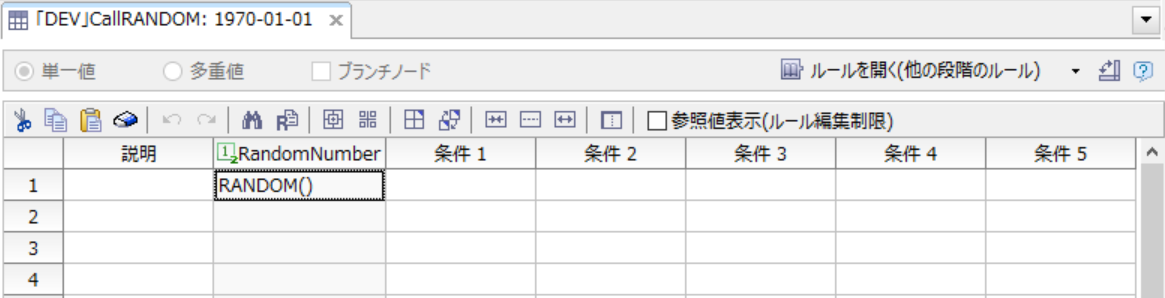
## RANDOM関数

一般的に関数は同一入力については同一結果を返す。ただし、タイムスタンプなどの乱数を発生する関数の場合、同一入力に対しても異なる結果を返す場合もある。次の表は乱数を発生させるRANDOMユーザ定義関数のコードである。

[表 121 RANDOM関数]

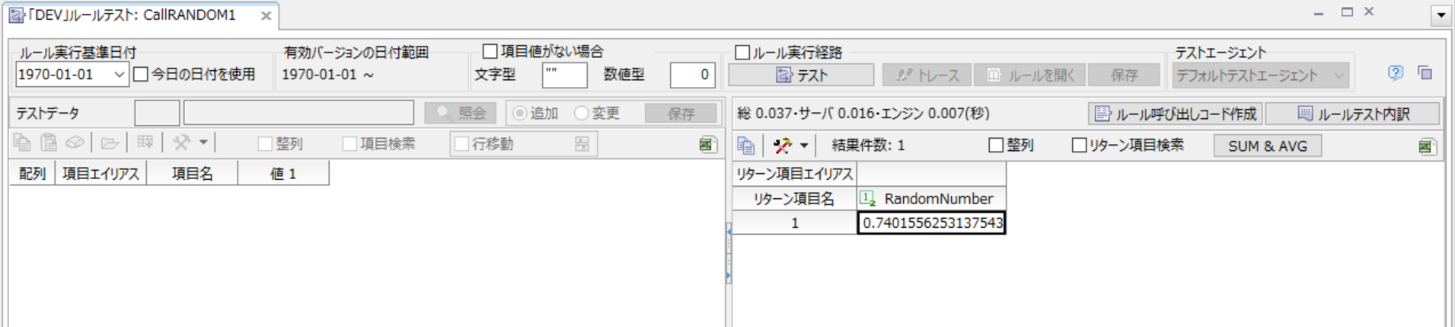
|  |
| --- |
| 1 package com.innorules.apiguide.udf;  2  3 import java.util.\*;  4  5 import com.innoexpert.innorulesj.abstractcompiler.\*;  6 import com.innoexpert.innorulesj.exceptions.\*;  7 import com.innoexpert.innorulesj.suite.udf.\*;  8  9 public class FuncRANDOM implements UserDefinedFunction  10 {  11 private static final Random RANDOM = new Random( System.currentTimeMillis() );  12  13 public String getName()  14 {  15 return "RANDOM";  16 }  17  18 public Type check( Type[] params, List rules, List items ) throws CompileException  19 {  20 if( params.length != 0 )  21 throw new CompileException( "Usage: RANDOM()" );  22 return TypeImpl.NUMBER;  23 }  24  25 public Object execute( Engine engine, Object[] params ) throws  UserDefinedFunctionException  26 {  27 return Double.valueOf( RANDOM.nextDouble() );  28 }  29 } |

FuncRANDOMクラスをクラスパスに位置させ、イノルールズシステム設定ファイルのuser-defined-functionに該当パスを追加してRANDOM関数を登録する。次の図はRANDOM関数を使用する{CallRANDOM}ルールである。



[図 23 CallRANDOMルール]

このルールを実行する場合、常に異なる乱数を返す。次の図は{CallRANDOM}ルールをテストした結果である。次の図の右側の上段のルール実行経路のチェックボックスはチェックしないこと。



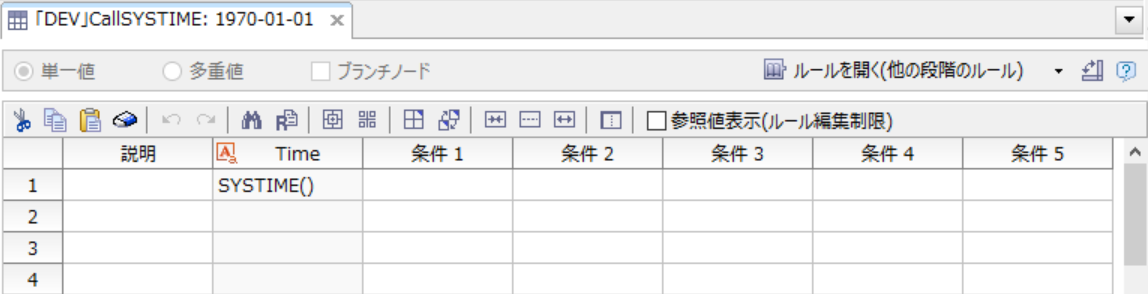
[図 24 ルール実行結果]

テストの結果はユーザごとに異なる可能性があるが、ビルダーサーバを含むルールランタイムシステムの再起動前までは結果に変化ないことが確認できる。これはこのルールが項目や他のルールを使用しないため、ルールエンジンはこのルールを定数ルールと見なして、最初に実行された結果が以降のルールトランザクションでも使用されるためである。

定数ルールの結果を再使用するには例外がある。ルールをテストする時にはルールの結果および性能よりも、実行経路が重要であるため、ルールトレースが有効化されている場合、定数ルールでもトランザクションごとに新しく実行される。[図 24 ルール実行結果]の画面でルール実行経路のチェックボックスをチェックして再実行する。

## ビルトイン関数に対するユーザ定義関数の制約事項

次の図は項目や他のルールを使用しないルールを示す。ただし、このルールは[図 23 CallRANDOMルール]と違ってビルトイン関数であるSYSTIMEを使用している。

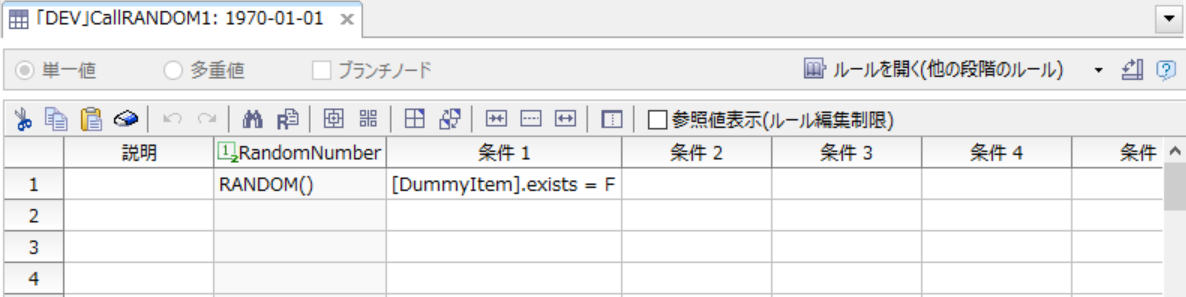


[図 25 CallSYSTIME]

このルールをテストするとトレースの有効化有無に関係なく常に同一結果を出力できない。これにより、このルールが項目や他のルールを使用していないが、ルールエンジンはこのルールを定数ルールと見なしていない。ビルトイン関数中、ルール内で使用すると項目はルールの呼び出し可否に関係なく、そのルールを非定数ルールと見なせる関数がある。そのビルトイン関数のリストは次のとおりである。

* GET
* SET
* SYSDATE
* SYSTIME

ユーザ定義関数はcom.innoexpert.innorulesj.suite.udf.Engineに定義されたサービスを使用してビルトイン関数レベルの強力な機能を実行できる。ただし、定数ルールに関して上記の制約事項があるため、ロジックをユーザ定義関数で実装する場合は注意が必要である。現在の機能で[図 23 CallRANDOMルール]のRANDOM関数を使用するルールを非定数ルールとして作成するためには次の図のようにダミー項目を使用する迂回的な方法もある。注意する点は、ダミー項目の実行経路上で1回は使用されなければならないことである。



[図 26 非定数ルールCallRANDOM]