第 1 章　介绍

从 Drools 统一行为建模平台的视野看，Drools Fusion 是负责启用事件处理行为的一个模块。

1.1　复杂事件处理

尽管做了多次尝试，有关复杂事件处理的术语并没有一个最新的广泛接受的定义。事件术语本身被习惯地频繁用来指不同事情，取决于它使用的上下文。定义术语并不是本指南的目标，因此，让我们采用一种宽松的定义，虽然非正式，它却会允许我们在一个共同理解基础上继续进行。

所以，在本指南的范围内：

重要：事件，指在应用程序域中的状态的一个有意义改变的一条记录。

例如，在 Stock Broker 应用程序中，当执行一个销售操作时，它导致在该域中一个状态的改变。这个状态的改变可以在该项域中的几个实例上被观察到，如证券价格改变为来匹配操作的值，个人交易资产的业主从卖方改变为买方，来自买卖双方的帐目余额被记入贷方和借方，等等。取决于域如何被建模，这种状态的改变可以用单事件、多原子事件，或者甚至相关层次事件来表示。在任何情况下，在这个指南的上下文中，事件是在这个域中的有关一个特定数据改变的一个记录。

从事件被发明开始，事件由计算机系统处理，纵观历史，系统负责给它不同的名字和使用不同的方法。虽然直到 90 年代未，一个更引人关注的工作，在 EDA（Event Driven Architecture 事件驱动体系结构）上开始了，事件处理的需求和目的使用了一个更正式的定义。旧的消息系统开始改变来解决这种需求，而且新的系统开始开发使用单目标事件处理。两种潮流，在事件流处理和复杂事件处理的名字下诞生了。

在前期，事件流处理关注在（接近）实时中的事件流处理能力，而复杂事件处理关注的是相关性，以及原子事件组合成的复杂（复合）事件。一个重要的（可能是最重要的）里程碑是 2002 年发行的 Dr. David Luckham 的书"The Power of Events"。在该书中，Dr. David Luckham 引入了复杂事件处理的概念，以及它如何可以被用来增强处理事件的系统。多年来，两种潮流汇集成了一个共识，而今天这些系统都是指 CEP 系统。

这是对一个相当复杂和丰富的研究领域的过分简单化的解释，但是却为本指南引入的概念确立了一个高级共识。

对复杂事件处理是什么的目前理解，可以引用维基百科的简明描述，如下所示：

重要：“复杂事件处理（CEP），本质上是一个事件处理概念，涉及处理多个事件的任务与在事件云内部识别有意义事件的目标。CEP 使用了技术，例如，多事件的复杂模式检测、事件关联和抽象、事件层级，以及事件之间的关系，例如，因果关系、成员、同步、事件驱动处理。”

换言之，CEP 是关于从一个事件云中检测和选择感兴趣的事件（不只这些），找出它们之间的关系，根据它们和它们之间的关系推断新的数据。

注意：对该指南的其余部分，我们会使用复杂事件处理（CEP）术语，作为用于所有相关技术和工艺的泛指，一般包含但不限于 CEP、杂事件处理、ESP、事件流处理，以及事件处理。

1.2 Drools Fusion

总体来说，事件处理用例，共享业务规则用例的几个需求和目标。这些重叠发生在业务方面和技术方面。

在业务方面：

●·业务规则常常被定义在出现了事件触发情况的基础上。可能的例子有：

- ●·在一个算法交易应用程序中：如果证券价格比当日开盘价上涨了 x%，则采取一个动作，在股票交易应用程序中，通常利用事件来指示价格上涨的位置。
- ●·在一个测控应用程序中，如果服务器的房间的温度在 Y 秒中升高了 X 度，则采取一个动作，通常利用事件来指示传感器读取的位置。

●·业务规则和事件处理查询经常变动，需要业务立即响应，使自己适应新的市场条件，新的规则和新企业策略。

从技术角度看：

●·二者需要无缝与企业基础和应用集成，特别是自主管理上，包括，但不只限于，生命周期管理，审计，安全等等。

●·二者都有象模式匹配这样的功能需求，以及象响应时间和查询/规则解释这样的非功能需求。

纵观历史，即便二者共享需求和目标，它们分别诞生在两个领域，尽管行业发展，可以在市场上找到一个好的产品，它们要么关注事件处理，要么关注规则管理。这不仅归就于历史原因，也归就于即使重叠部分，用例也确实有一些不同需求。

重要：几年前，Drools 也是作为一个规则引擎诞生的，但是随后的版本成为了一个行为建模的单一平台，它不久意识到要达到这个目标，只能把重点归到三个互补的业务建模技术：

●·业务规则管理

●·业务流程管理

●·复杂事件处理

在本上下文中，Drools Fusion 是负责添加事件处理能力到该本平台中的一个模块。

不过，支持复杂事件处理，是比简单的理解事件是什么要更多得多。CEP 场景具有几个共同而明显的特点：

●·通常需要处理巨量的事件，但是只有少部分事件是真正关心的。

●·事件通常是不变的，因为它们是状态改变的一条记录。

●·通常有关事件的规则和查询必须是运行在被动模式（reactive modes），即，对事件模式（patterns）的检测作出反应。

●·通常在相关的事件之间有强烈的时间关系。

●·个别事件通常是不重要的。系统关心相关事件的模式（patterns）和它们的关系。

●·通常，要求系统执行组合和聚合的事件。

基于这个一般共同特点，Drools Fusion 为恰当地支持复杂事件处理，定义了一组实现的目标：

●·使用正确的语义，支持事件作为一流类公民。

●·允许检测、关联、聚合和组合事件。

●·为了模拟事件之间的时间关系，支持时间约束。

●·支持有趣事件的滑动窗口。

●·支持一个会话域统一时钟。

●·支持 CEP 用例必需的事件容量。

●·支持主动（被动）规则。

●·支持用于事件输入到引擎（管道）内的适配器。

Drools Expert 本身并没有包含上述基于需求的目标列表，因为在一个统一平台中，一个模块的功能可以被其他模块利用。这样，Drools Fusion 与生俱有企业级的功能，如模式匹配（Pattern Matching），它是极为重要的一个 CEP 产品，然而它已经由 Drools Expert 提供。

同样，Drools Fusion 提供的所有功能也被 Drools Flow 利用（反之也然），使流程管理知道事件处理，反之也然。

对本指南的其余部分，我们将遍历增加到系统的每一个 Drools Fusion 功能。所有这些功能可用于支持在 CEP 世界中的不同用例，并且用户可以自由选择和使用一个，用于帮助他建模他的业务用例。

第 2 章　Drools Fusion 的功能

2.1　事件

从 Drools 的视角看，事件只是一个特殊类型的事实。那么，我们可以说所有事件都是事实，但是不是所有事实都是事件。在接下来的一些章节中，会显示事件特性的具体差异。

2.1.1　事件语义

一个事件是呈现出许多明显特性的事实：

- **通常不可变**：因为，依据前面讨论的定义，事件是在应用程序域中的状态变化的一条记录，即，已经发生了某事的一条记录，过去的不能被“改变”，所以，事件是不可变的。这种约束对一些优化的开发和事件生命周期的规范是一个重要的必要条件。这并不意味着 java 对象表示的对象必须是不可变的。恰恰相反，引擎并不强迫对象模型的不变性，因为规则的最普遍用例之一是事件的数据丰富。

技巧：作为最佳实践，允许应用程序填充空的事件属性（用推断的数据丰富事件），但是已经填充的属性决不应该被更改。

●·**强烈的时间约束**：规则涉及的事件通常需要多个事件的相关性，尤其是时间相关性，表示相对于其他事件的事件及时在某点发生。

●·**被管理的生命周期**：由于事件的不可变的本性和时间约束，事件在一个有限的时间窗口期间通常只会匹配另外的事件和事实，这让引擎自动管理事件的生命周期成为可能。换言之，一个事件被插入到工作内存，当一个事件不再匹配其他事件时，引擎可以找出它，并且自动回收它，释放相关联的资源。

●·**使用滑动窗口**：因为所有的事件都有与它们关联的时间戳，所以，对它们可以定义和使用滑动窗口，允许在一个时间段上根据值的聚合创建规则，例如，整个 60 分钟的一个事件的值的平均值。

Drools 使用两种语义支持事件的声明和用法：**point-in-time** 事件和 **interval-based** 事件。

技巧：理解统一的语义的一个最简方法，是把 **point-in-time** 事件认作为是期限为 0 的 **interval-based** 事件。

2.1.2 事件声明

要把一个事实类型声明为一个事件，只需要把@role 元数据标签指派给该事实类型。@role 元数据标签接受两种可能的值：

●·fact : 这是默认值，声明该类型作为一个正规事实（regular fact.）被处理。

●·event :声明该类型作为一个事件被处理。

例如，在下面的例子中，在一个股票经济人应用程序中，声明的事实类型 StockTick 会被作为一个事件被处理。

例子 2.1 声明一个事实作为一个事件

```
import some.package.StockTick

declare StockTick
    @role( event )
end
```

同样适用于内联事实的声明。那么，如果 StockTick 是一个在 DRL 本身中声明的一个事实类型，而不是前面存在的类，则代码应该如下：

例子 2.2 声明一个事实类型，并指派它为事件角色。

```
declare StockTick
    @role( event )

    datetime : java.util.Date
    symbol : String
    price : double
end
```

有关类型声明的更多信息，请看 Drools Expert 文档中的规则语言部分。

2.1.3 事件元数据

所有事件都有一组与之关联的元数据。大部分元数据者有默认值，当它们插入到工作内存时，默认值会自动分配给每个事件，但是可以在事件类型的基础上，使用列在下面的元数据标签改变它们。

例如，我们假设用户在应用程序域模式中有如下的类：

例子 2.3 VoiceCall 事实类

```
/**
 * A class that represents a voice call in
 * a Telecom domain model
 */
public class VoiceCall {
    private String  originNumber;
    private String  destinationNumber;
    private Date    callDateTime;
    private long    callDuration;          // in milliseconds

    // constructors, getters and setters
}
```

2.1.3.1 @role

@role 元数据已在前面章节讨论了，为了完整性显示在这里：

```
@role( <fact|event> )
```

它注释一个给定的事实作为一个正规事实或一个事件。它接"fact"或"event"作为参数，默认为"fact"。

例子 2.4 声明 VoiceCall 作为一个事件类型

```
declare VoiceCall
    @role( event )
end
```

2.1.3.2 @timestamp

每一个事件都有一个关联的时间戳指派给它。默认时，一个给定事件的时间戳是在事件被插入到工作内存时，从 Session Clock 读取，并且分配给该事件。尽管如此，有些时候，事件用时间戳作为它自己的一个属性。在这情况下，用户可能告诉引擎使用来自事件属性的时间戳，而不是从 Session Clock 读取。

```
@timestamp( <attributeName> )
```

要告诉引擎用什么属性来作为该事件的时间戳的源头，只需列出属性名作为 @timestamp 的参数即可：

例子 2.5 声明 VoiceCall 时间戳属性

```
declare VoiceCall
    @role( event )
    @timestamp( callDateTime )
end
```

2.1.3.3 @duration

Drools 支持两种事件语义：point-in-time 事件和 interval-based 事件。一个 **point-in-time** 事件可以用期限为 0 的 **interval-based** 事件表示。默认时，所有事件的期限为 0。用户可以通过在包含了事件期限的事件类型中声明那个属性使用一个不同的期限。

```
@duration( <attributeName> )
```

那么，对我们的 VoiceCall 事实类型，声明会是这样的：

例子 2.6 声明 VoiceCall 的期限（duration）属性

```
declare VoiceCall
    @role( event )
    @timestamp( callDateTime )
    @duration( callDuration )
end
```

2.1.3.4 @expires

重要：这个标签只有引擎运行在流（STREAM）模式之下才会被考虑。此外，在内存管理章节对使用这个标签的效果做了进一步的讨论。在这里包含它只是为了完整性。

在工作内存中的某个时间之后，事件可以被自动到期。它通常发生在，基于知识库中的现行规则，事件可能不再匹配和激活任何规则时。尽管，可以显示定义一个事件在什么时候应该到期。

```
@expires( <timeOffset> )
```

timeOffset 的值是以下格式的一个时间间隔：

```
[#d][#h][#m][#s][#[ms]]
```

在这里，[]是一个可选参数，#指是一个数值。

那么，要声明 VoiceCall 事实在插入工作内存后，在 1 小时 35 分钟之后，VoiceCall 事实会被到期，应该这样编写：

例子 2.7 声明 VoiceCall 事件的到期偏移量

```
declare VoiceCall
```

```
    @role( event )
    @timestamp( callDateTime )
    @duration( callDuration )
    @expires( 1h35m )
end
```

2.2 会话时钟（Session Clock）

随着时间的推移进行推理需要一个参考时钟。仅举一个例子，如果一个规则推断在过去 60 分钟中的一个特定股票的整个平均价格，为了计算平均值，引擎如何知道过去 60 分钟中股票价格发生变化呢？明显的反应是：通过用“当前的时间”与事件的时间戳进行比较。引擎如何知道当前的时间呢？显然，通过查询会话时钟。

会话时钟实现了一个策略模式，允许引擎插入和使用不同类型的时钟。这非常重要，因为引擎可能运行在一个不同的场景阵列中，这些场景需要不同的时钟实现，仅举几个例子：

- 规则测试：测试总是需要一个可控的环境，并且当测试包含了带有时间约束的规则时，不仅需要控制输入规则和事实，而且也需要时间流。
- 定期（regular）执行：通常，在运行生产规则时，应用程序需要一个实时时钟，允许引擎对时间的行进立即作出反应。
- 特殊环境：特殊环境可以对时间的控制有特殊的要求。群集环境可能需要在心跳中的时钟同步，或 JEE 环境可能需要使用一个应用服务器提供的时钟，等等。
- 规则重演或模拟：要重演场景或模拟场景也需要应用程序控制时间流。

2.2.1 可用的时钟实现

Drools 5.0 提供了两个开箱即用的时钟实现。默认的，基于系统时钟的实时时钟，和一个可选的，由应用程序控制的伪时钟。

●·实时时钟

默认时，Drools 使用一个内部使用系统时钟的实时时钟来确定当前时间戳。

要显示配置引擎使用实时时钟，只需设置会话参数为实时即可：

```
KnowledgeSessionConfiguration config =
KnowledgeBaseFactory.newKnowledgeSessionConfiguration();
config.setOption( ClockTypeOption.get("realtime") );
```

●·伪时钟

Drools 也提供了一个开箱即用的由应用程序控制的一个所谓的伪时钟。这个时钟对测试时间规则的单元特别有用，因为它可以被应用程序控制，所以结果成为确定的。

要配置伪时钟，做：

```
KnowledgeSessionConfiguration config =
KnowledgeBaseFactory.newKnowledgeSessionConfiguration();
config.setOption( ClockTypeOption.get("pseudo") );
```

下面是如何控制伪会话时钟的一个例子：

```
KnowledgeSessionConfiguration conf =
KnowledgeBaseFactory.newKnowledgeSessionConfiguration();
conf.setOption( ClockTypeOption.get( "pseudo" ) );
StatefulKnowledgeSession session =
kbase.newStatefulKnowledgeSession( conf, null );

SessionPseudoClock clock = session.getSessionClock();

// then, while inserting facts, advance the clock as necessary:
```

```
FactHandle handle1 = session.insert( tick1 );
clock.advanceTime( 10, TimeUnit.SECONDS );
FactHandle handle2 = session.insert( tick2 );
clock.advanceTime( 30, TimeUnit.SECONDS );
FactHandle handle3 = session.insert( tick3 );
```

2.3 流（stream）支持

大部分 CEP 用例必须处理事件流（stream）。可以为应用程序提供各种格式的流，从 JMS 队列到纯文本文件，从数据库表到原始套接字，或者甚至通过网页服务调用。无论什么情况，流共有一组共同的特性：

- 在流中的事件通过时间戳被排序。对不同的流时间戳有不同的语义，但是它们总是内在地被排序。
- 事件的数量（volumes）总是很高的。
- 原子事件自己是很少有用的。通常根据多个事件之间的相关性或流或其他来源提取含义。
- 流可以是相似的，即包含单一类型的事件；或者是异类的，即包含多种类型的事件。

Drools 归纳流概念作为一个“入口点”进入到引擎内。入口点是 Drools 的一个大门，事实从它进来。事实可以是正规事实或者象事件这样的特殊事实。

在 Drools 中，来自一个入口点（流）的事实可以与来自其他的入口点的事实联合，或者事件与来自工作内存的事实联合。尽管这样，它们决不能混淆，即它们不能丢失对它们进入到引擎的那个入口点的引用。这是非常重要的，因为它可能包含有通过几个入口点进入引擎内的相同类型的事实，例如，一个通过入口点 A 插入到引擎的事实决不会匹配来自入口点 B 的模式。

2.3.1 声明和使用入口点

在 Drools 中，通过在规则中直接使用入口点来显式地声明它们。即，在一个规则中引用一个入口点，会使引擎在编译时识别且创建适当的结构来支持那个入口点。

那么，例如，我们假设一个银行业应用程序，来自流的事件被送入到该系统中。一个流包含了所有在 ATM 机上执行的事务。所以，如果一条规则说：当且仅当存款余额超过了请求的取款金额，取款才被授权，规则应该象下面这样：

例子 2.8 流用法的例子

```
rule "authorize withdraw"
when
    WithdrawRequest( $ai : accountId, $am : amount ) from entry-point "ATM
Stream"
    CheckingAccount( accountId == $ai, balance > $am )
then
    // authorize withdraw
end
```

在上面的例子中，引擎编译器会识别联结到入口点"ATM Stream"的模式，并且会创建规则库需要的所有结构用于支持"ATM Stream"，而且只会匹配来自"ATM Stream"的 WithdrawRequests。在上面的例子中，规则也把来自流的事件与来自主工作内存的一个事实（CheckingAccount）联合在一起。

现在，我们假设第二个规则，它规定被安排在银行分行的取款请求，会增加 2 元的费用。

例子 2.9 使用了一个不同的流。

```
rule "apply fee on withdraws on branches"
when
    WithdrawRequest( $ai : accountId, processed == true ) from
entry-point "Branch Stream"
    CheckingAccount( accountId == $ai )
```

```
then
    // apply a $2 fee on the account
end
```

上面的规则会匹配与第一个规则有完全相同类型的事件（WithdrawRequest），但是它来自不同的流，所以，被插入到"ATM Stream"中的事件决不会根据第二个规则中的模式进行运算，因为该规则规定它只对来自"Branch Stream"的模式感兴趣。

所以，入口点，除了是流的一个适当抽象外，也是在工作内存中扩大事实的一种方法，同是也是减少交叉产品爆炸（cross products explosions）的一个宝贵工具。不过，这是其他时间讨论的一个主题。

插入事件到一个入口点内是同样简单的。而不是直接把事件插入到工作内存，如下所示把它们插入到入口点：

例子 2.10 插事实到一个入口点内

```
// create your rulebase and your session as usual
StatefulKnowledgeSession session = ...

// get a reference to the entry point
WorkingMemoryEntryPoint atmStream =
session.getWorkingMemoryEntryPoint( "ATM Stream" );

// and start inserting your facts into the entry point
atmStream.insert( aWithdrawRequest );
```

上面的例子，显示了如何手动地把事实插入到一个特定的入口点内。虽然，通常应用程序会使用众多适匹器中的一个，用于插接一个流终点（to plug a stream end point），如一个 JMS 队列，直接进入到引擎入口点内，不必编码手动插入。Drools 管道 API，有几个适匹器和助手用于做这个，以及有关如何做它的例子。

2.4 时间（Temporal）推理

时间推理是任何 CEP 系统的又一个必要条件。如上所讨论，事件的一个显著特性是它们强烈的时间关系。

时间推理是一个广泛研究的领域，从有关时间形式逻辑（Temporal Modal Logic）的根源到业务系统中更实际的应用。

对一些应用描述的方法，有数以百计的文章和编写的论文。Drools 在几个资源的基础上，再次采取了务实和简单的方法，下面的文章特别值得注意：

[ALLEN81]Allen, J.F..An Interval-based Representation of Temporal Knowledge. 1981.

[ALLEN83]Allen, J.F..Maintaining knowledge about temporal intervals. 1983.

[BENNE00] by Bennet, Brandon and Galton, Antony P.. A Unifying Semantics for Time and Events. 2005.

[YONEK05] by Yoneki, Eiko and Bacon, Jean. Unified Semantics for Event Correlation Over Time and Space in Hybrid Network Environments. 2005.

Drools 实现了由 Allen 描述的 Interval-based Time Event（基于区间的时间事件）语义，并且用期限为 0 的 Interval-based 事件表示 Point-in-Time 事件。

2.4.1 时间运算符

Drools 实现了由 Allen 定义的所有 13 种运算符，以及它们的逻辑补（非）。这部分详细描述每个运算符和它们的参数。

2.4.1.1 After

After 运算符与两个事件相关， 并且当从当前事件到相关联事件的时间距离属于操作符声明的距离范围时进行匹配。

让我们看一个例子：

```
$eventA : EventA( this after[ 3m30s, 4m ] $eventB )
```

当且仅当在$eventB 完成时的时间与当$eventA 开始时的时间的时间距离是在（3 分 30 秒）和（4 分钟）之间时，上面的模式才会匹配。换言之：

```
3m30s <= $eventA.startTimestamp - $eventB.endTimeStamp <= 4m
```

After 运算符的时间距离区间是可选的：

- 如果定义了两个值（如下所示），区间从第一个值开始到第二个值结束。
- 如果只定义了一个值，区间从该值开始到正无穷大结束。
- 如果没有定义值，它假设开始值为 1 毫秒，终值为正无穷大。

技巧：这个运行运算符可以定义负的距离，例如：

```
$eventA : EventA( this after[ -3m30s, -2m ] $eventB )
```

注意：如果第一个值大于第二个值，引擎自动倒转它们，因为没有理由第一个值大于第二个值。例如，下面的两个模式被认为有相同的语义：

```
$eventA : EventA( this after[ -3m30s, -2m ] $eventB )
$eventA : EventA( this after[ -2m, -3m30s ] $eventB )
```

2.4.1.2 Before

Before 运算符与两个事件相关， 并且当从相关联事件到当前事件的时间距离属于操作符声明的距离范围时进行匹配。

让我们看一个例子：

```
$eventA : EventA( this before[ 3m30s, 4m ] $eventB )
```

当且仅当在$eventB 开始时的时间与当$eventA 完成时的时间的时间距离是在（3 分 30 秒）和（4 分钟）之间时，上面的模式才会匹配。换言之：

```
3m30s <= $eventB.startTimestamp - $eventA.endTimeStamp <= 4m
```

Before 运算符的时间距离区间是可选的：

- 如果定义了两个值（如下所示），区间从第一个值开始到第二个值结束。
- 如果只定义了一个值，区间从该值开始到正无穷大结束。
- 如果没有定义值，它假设开始值为 1 毫秒，终值为正无穷大。

技巧：这个运行运算符可以定义负的距离，例如：

```
$eventA : EventA( this before[ -3m30s, -2m ] $eventB )
```

注意：如果第一个值大于第二个值，引擎自动倒转它们，因为没有理由第一个值大于第二个值。例如，下面的两个模式被认为有相同的语义：

```
$eventA : EventA( this before[ -3m30s, -2m ] $eventB )
```

```
$eventA : EventA( this before[ -2m, -3m30s ] $eventB )
```

2.4.1.3 Coincides

Coincidesf 运算符关联两个事件，并且当两个事件在相同的时间发生时进行匹配。可选的，运算符接受事件的开始时间戳和结束时间戳之间的距离阈值。

让我们看一个例子：

```
$eventA : EventA( this coincides $eventB )
```

当且仅当$eventA 和$eventB 两个的开始时间戳相同，且$eventA 和$eventB 结束时间戳也相同时，上面的模式才会匹配。

可选的，这个运算符接受一个或两个操作符。这些参数是匹配时间戳之间的距离阈值。

- 如果只给了一个参数，其用于开始时间戳和结束时间戳两个。
- 如果给了两个参数，那么第一个用来作为开始时间戳的阈值，第二用来作为结束时间戳的阈值。

换言之：

```
$eventA : EventA( this coincides[15s, 10s] $eventB )
```

上面的模式会匹配，当且仅当：

```
abs( $eventA.startTimestamp - $eventB.startTimestamp ) <= 15s &&
abs( $eventA.endTimestamp - $eventB.endTimestamp ) <= 10s
```

警告：该参数使用负的区间值是没有意义，并且如果那样，引擎会引发一个错误。

2.4.1.4 During

During 运算符关联两个事件，并且在关联的事件出现期间当前的事件发生时进行匹配。

让我们看一个例子：

```
$eventA : EventA( this during $eventB )
```

当且仅当在$eventA 开始在$eventB 开始后，且结束在$eventB 结束前，上面的模式会匹配。

换言之：

```
$eventB.startTimestamp < $eventA.startTimestamp <= $eventA.endTimestamp
< $eventB.endTimestamp
```

During 运算符接受 1、2 或 4 个参数，如下所示：

- ‥如果定义一个值，它会是用于运算符匹配的两个事件的开始时间戳之间的最大距离，和两个事件的结束时间戳的最大距离，如：

```
$eventA : EventA( this during[ 5s ] $eventB )
```

模式会匹配，当且仅当：

```
0 < $eventA.startTimestamp - $eventB.startTimestamp <= 5s &&
```

```
0 < $eventB.endTimestamp - $eventA.endTimestamp <= 5s
```

●·如果定义两个值，第一个值是两个事件的开始时间戳之间的最小距离，第二个值是两个事件的结束时间戳的最大距离，如：

```
$eventA : EventA( this during[ 5s, 10s ] $eventB )
```

模式会匹配，当且仅当：

```
5s <= $eventA.startTimestamp - $eventB.startTimestamp <= 10s &&
5s <= $eventB.endTimestamp - $eventA.endTimestamp <= 10s
```

●·如果定义四个值，前面两个值是两个事件的开始时间戳之间的最小距离和最大距离，后面两个值是两个事件的结束时间戳之间的最小距离和最大距离：

```
$eventA : EventA( this during[ 2s, 6s, 4s, 10s ] $eventB )
```

模式会匹配，当且仅当：

```
2s <= $eventA.startTimestamp - $eventB.startTimestamp <= 6s &&
4s <= $eventB.endTimestamp - $eventA.endTimestamp <= 10s
```

2.4.1.5 Finishes

Finishes 运算符关联两个事件，在当前事件的开始时间戳发生在相关联的事件开始时间戳之后，但两个的结束时间戳发生在相同时间时进行匹配。

让我们看一个例子：

```
$eventA : EventA( this finishes $eventB )
```

当且仅当 $eventA 开始在$eventB 开始之后，并且结束在$eventB 结束的相同时间，上面的模式会匹配。

换言之：

```
$eventB.startTimestamp < $eventA.startTimestamp &&
$eventA.endTimestamp == $eventB.endTimestamp
```

finishes 运算符接受一个可选参数。如果定义了它，它确定两个事件的结束时间戳之间的最大距离，用于运算符匹配。如：

```
$eventA : EventA( this finishes[ 5s ] $eventB )
```

会匹配，当且仅当：

```
$eventB.startTimestamp < $eventA.startTimestamp &&
abs( $eventA.endTimestamp - $eventB.endTimestamp ) <= 5s
```

警告：该参数使用负的区间值是没有意义，并且如果那样，引擎会引发一个异常。

2.4.1.6 Finished by

Finishedby 运算符关联两个事件，在当前事件的开始时间戳发生在相关联的事件开始时间戳之前，但两个的结束时间戳发生在相同时间时进行匹配。它是 Finishes 运行符的对称面。

让我们看一个例子：

```
$eventA : EventA( this finishedby $eventB )
```

当且仅当 $eventA 开始在$eventB 开始之前，并且结束在$eventB 结束的相同时间，上面的模式会匹配。

换言之：

```
$eventA.startTimestamp < $eventB.startTimestamp &&
$eventA.endTimestamp == $eventB.endTimestamp
```

finishedby 运算符接受一个可选参数。如果定义了它，它确定两个事件的结束时间戳之间的最大距离，用于运算符匹配。如：

```
$eventA : EventA( this finishedby[ 5s ] $eventB )
```

会匹配，当且仅当：

```
$eventA.startTimestamp < $eventB.startTimestamp &&
abs( $eventA.endTimestamp - $eventB.endTimestamp ) <= 5s
```

警告：该参数使用负的区间值是没有意义，并且如果那样，引擎会引发一个异常。

2.4.1.7 Includes

Includes 运算符关联两个事件，并且关联的事件发生在当前事件期间时进行匹配。它是 during 运算符的对称面。

让我们看一个例子：

```
$eventA : EventA( this finishes $eventB )
```

当且仅当在$eventB 开始在$eventA 开始后，且结束在$eventA 结束前，上面的模式会匹配。

换言之：

```
$eventA.startTimestamp < $eventB.startTimestamp <=
$eventB.endTimestamp < $eventA.endTimestamp
```

Includes 运算符接受 1、2 或 4 个参数，如下所示：

- 如果定义一个值，它会是用于运算符匹配的两个事件的开始时间戳之间的最大距离，和两个事件的结束时间戳的最大距离，如：

```
$eventA : EventA( this includes[ 5s ] $eventB )
```

模式会匹配，当且仅当：

```
0 < $eventB.startTimestamp - $eventA.startTimestamp <= 5s &&
0 < $eventA.endTimestamp - $eventB.endTimestamp <= 5s
```

- 如果定义两个值，第一个值是两个事件的开始时间戳之间的最小距离，第二个值是两个事件的结束时间戳的最大距离，如：

```
$eventA : EventA( this includes[ 5s, 10s ] $eventB )
```

模式会匹配，当且仅当：

```
5s <= $eventB.startTimestamp - $eventA.startTimestamp <= 10s
&&
5s <= $eventA.endTimestamp - $eventB.endTimestamp <= 10s
```

- 如果定义四个值，前面两个值是两个事件的开始时间戳之间的最小距离和最大距离，后面两个值是两个事件的结束时间戳之间的最小距离和最大距离：

```
$eventA : EventA( this includes[ 2s, 6s, 4s, 10s ] $eventB )
```

模式会匹配，当且仅当：

```
2s <= $eventB.startTimestamp - $eventA.startTimestamp <= 6s
&&
4s <= $eventA.endTimestamp - $eventB.endTimestamp <= 10s
```

2.4.1.8 Meets

Meets 运算符关联两个事件，并且当前事件的结束时间戳发生与关联事件的开始时间戳相同时进行匹配。

让我们看一个例子：

```
$eventA : EventA( this meets $eventB )
```

当且仅当$eventA 结束在$eventB 开始的相同时间，上面的模式会匹配。

换言之：

```
abs( $eventB.startTimestamp - $eventA.endTimestamp ) == 0
```

meets 运算符接受一个可选参数。如果定义了它，它确定当前事件的结束时间戳与关联事件的开始时间戳的最大距离，用于该运算符的匹配，例如：

```
$eventA : EventA( this meets[ 5s ] $eventB )
```

会匹配，当且仅当：

```
abs( $eventB.startTimestamp - $eventA.endTimestamp) <= 5s
```

警告：该参数使用负的区间值是没有意义，并且如果那样，引擎会引发一个异常。

2.4.1.9 Met by

Metby 运算符关联两个事件，并且当前事件的开始时间戳发生与关联事件的结束时间戳相同时进行匹配。

让我们看一个例子：

```
$eventA : EventA( this metby $eventB )
```

当且仅当$eventA 开始在$eventB 结束的相同时间，上面的模式会匹配。

换言之：

```
abs( $eventA.startTimestamp - $eventB.endTimestamp ) == 0
```

metby 运算符接受一个可选参数。如果定义了它，它确定当前事件的结束时间戳与关联事件的开始时间戳的最大距离，用于该运算符的匹配，例如：

```
$eventA : EventA( this metby[ 5s ] $eventB )
```

会匹配，当且仅当：

```
abs( $eventA.startTimestamp - $eventB.endTimestamp) <= 5s
```

警告：该参数使用负的区间值是没有意义，并且如果那样，引擎会引发一个异常。

2.4.1.10 Overlaps

Overlaps 运算符关联两个事件，并且在当前事件开始在关联事件开始前，结束在关联事件开始后，但在关联事件结束前时进行匹配。换言之，两个事件有一个重叠期。

让我们看一个例子：

```
$eventA : EventA( this overlaps $eventB )
```

上面的例子会匹配，当且仅当：

```
$eventA.startTimestamp < $eventB.startTimestamp <
$eventA.endTimestamp < $eventB.endTimestamp
```

overlaps 运算符接受 1 或 2 个参数，如下所示：

- 如果定义了一个参数，它将是关联事件的开始时间戳与当前事件的结束时间戳之间的最大距离。如：

```
$eventA : EventA( this overlaps[ 5s ] $eventB )
```

会匹配，当且仅当：

```
$eventA.startTimestamp < $eventB.startTimestamp <
$eventA.endTimestamp < $eventB.endTimestamp &&
0 <= $eventA.endTimestamp - $eventB.startTimestamp <= 5s
```

- 如果定义了两个值，它们将是关联事件的开始时间戳与当前事件的结束时间戳之间的最小距离和最大距离。如：

```
$eventA : EventA( this overlaps[ 5s, 10s ] $eventB )
```

会匹配，当且仅当：

```
$eventA.startTimestamp < $eventB.startTimestamp <
$eventA.endTimestamp < $eventB.endTimestamp &&
5s <= $eventA.endTimestamp - $eventB.startTimestamp <= 10s
```

2.4.1.11 Overlappd by

Overlappd 运算符关联两个事件，并且在关联事件开始在当前事件开始前，结束在当前事件开始后，但在当前事件结束前时进行匹配。换言之，两个事件有一个重叠期。

让我们看一个例子：

```
$eventA : EventA( this overlappedby $eventB )
```

上面的例子会匹配，当且仅当：

```
$eventB.startTimestamp < $eventA.startTimestamp <
$eventB.endTimestamp < $eventA.endTimestamp
```

overlappd 运算符接受 1 或 2 个参数，如下所示：

- 如果定义了一个参数，它将是当前事件的开始时间戳与关联事件的结束时间戳之间的最大距离。如：

```
$eventA : EventA( this overlappedby[ 5s ] $eventB )
```

会匹配，当且仅当：

```
$eventB.startTimestamp < $eventA.startTimestamp <
$eventB.endTimestamp < $eventA.endTimestamp &&
0 <= $eventB.endTimestamp - $eventA.startTimestamp <= 5s
```

- 如果定义了两个值，它们将是当前事件的开始时间戳与关联事件的结束时间戳之间的最小距离和最大距离。如：

```
$eventA : EventA( this overlappedby[ 5s, 10s ] $eventB )
```

会匹配，当且仅当：

```
$eventB.startTimestamp < $eventA.startTimestamp <
$eventB.endTimestamp < $eventA.endTimestamp &&
5s <= $eventB.endTimestamp - $eventA.startTimestamp <= 10s
```

2.4.1.12 Starts

Starts 运算符关联两个事件，并且在当前事件结束时间戳发生在关联事件的结束时间戳之前，但两个的开始时间戳发生在相同的时间进匹配。让我们看一个例子：

```
$eventA : EventA( this starts $eventB )
```

当且仅当 $eventA 结束在$eventB 结束之前，且开始在$eventB 开始的相同时间时，上面的模式会匹配。

换言之：

```
$eventA.startTimestamp == $eventB.startTimestamp &&
$eventA.endTimestamp < $eventB.endTimestamp
```

starts 运算符接受一个可选参数。如果定义了它，它确定两个事件开始时间戳的最大距离，用于该运算符的匹配，例如：

```
$eventA : EventA( this starts[ 5s ] $eventB )
```

会匹配，当且仅当：

```
abs( $eventA.startTimestamp - $eventB.startTimestamp ) <= 5s
&&
$eventA.endTimestamp < $eventB.endTimestamp
```

警告：该参数使用负的区间值是没有意义，并且如果那样，引擎会引发一个异常。

2.4.1.13 Started by

Startedby 运算符关联两个事件，并且在关联事件结束时间戳发生在当前事件的结束时间戳之前，但两个的开始时间戳发生在相同的时间进匹配。让我们看一个例子：

```
$eventA : EventA( this startedby $eventB )
```

当且仅当 $eventB 结束在$eventA 结束之前，且开始在$eventB 开始的相同时间时，上面的模式会匹配。

换言之：

```
$eventA.startTimestamp == $eventB.startTimestamp &&
$eventA.endTimestamp > $eventB.endTimestamp
```

startedby 运算符接受一个可选参数。如果定义了它，它确定两个事件开始时间戳的最大距离，用于该运算符的匹配，例如：

```
$eventA : EventA( this startedby[ 5s ] $eventB )
```

会匹配，当且仅当：

```
abs( $eventA.startTimestamp - $eventB.startTimestamp ) <= 5s
&&
$eventA.endTimestamp > $eventB.endTimestamp
```

警告：该参数使用负的区间值是没有意义，并且如果那样，引擎会引发一个异常。

2.5 事件处理模式

通常，规则引擎具有一个众所周知的处理数据和规则的方法，并且把结果提供给应用程序。此外，有关事实应该如何被提交给规则引擎并没有太多要求，尤其因为通常情况下，处理本身是时间无关的。对多数场景这是一个好的假定，但并不是所有都是。在要求包含实时或接近实时事件时，时间就成为了推理过程的一个重要变量。

下面的章节会解释在规则推理中的时间影响，以及 Drools 提供的用于推理过程的两个模式。

2.5.1 云（Cloud）模式

Cloud 处理模式是默认处理模式。规则引擎的用户是熟悉这种模式的，因为它的行为是与包含在前面的 Drools 版本中的任何纯正向链接规则引擎的行为完全相同。

当运行在 Cloud 模式时，作为一个整体，引擎了解工作内存中的所有事实，无论是正规事实还是事件。没有时间流的概念，尽管事件象平时一样有一个时间戳。换言之，尽管引擎知道一个给定的事件被创建了，例如，在 2009 年 1 月 1 日，09:35:40.767，引擎不可能确定事件是多“老”了，因为没有“现在”的概念。

在这种模式中，引擎通常会使用多对多模式匹配算法，使用规则约束找到匹配的元组，如平时一样激活并引发规则。

这种模式并不会对事实强加任何种类的其他要求，因此，例如：

●‥没有时间概念，不要求时钟同步。

●‥不要求事件顺序，引擎看事件象一个无序的云，引擎根据它们尝试匹配规则。

另一方面，因为没有要求，某些好处也是不可用。例如，在 Cloud 模式中，不可能使用滑动窗口，因为滑动窗口是基于“现在”概念的，而在在 Cloud 模式中没有“现在”的概念。

因为不要求事件顺序，所以引擎不可能确定什么时候事件不再匹配，以及没有事件的自动生命周期管理，即当事件不再匹配时，应用程序必须显式的收回它们，而且应用程序用相同的方法处理正规事实。

Drools 的 Cloud 模式是默认执行模式，但是在任何情况下，象在 Drools 中的其他配置一样，通过设置一个系统属性，使用配置属性文件或使用 API，也可以改变这种行为。相应的属性是：

```
KnowledgeBaseConfiguration config =
KnowledgeBaseFactory.newKnowledgeBaseConfiguration();
config.setOption( EventProcessingOption.CLOUD );
```

等价的属性是：

```
drools.eventProcessingMode = cloud
```

2.5.2 流（Stream）模式

流模式是当应用程序需要处理事件流时选择的模式。它为常规处理增加一些共同要求，但是却启用了一整套让事件流处理更简单的功能。

使用流模式的主要要求有：

- 在每个流中的事件是有时序的，即在一个给定的流内部，第一个发生的事件必须是第一个被插入到引擎中的。
- 引擎会通过使用会话时钟在流之间强制执行同步，所以，虽然应用程序不需要在流之间强制执行时序，但是非时间同步流（non-time-synchronized streams）的使用可能导致一些意想不到的结果。

上面给出的要求满足了，应用程序可以使用下面的 API 启用流模式：

```
KnowledgeBaseConfiguration config =
KnowledgeBaseFactory.newKnowledgeBaseConfiguration();
config.setOption( EventProcessingOption.STREAM );
```

或，等价的属性：

```
drools.eventProcessingMode = stream
```

在使用流模式时，引擎知道时间流的概念和“现在”的概念，即引擎根据从会话时钟读取的时间戳知道事件是多老了。这个特性允许为应用程序提供以下额外的功能：

- 滑动窗口支持。
- 自动事件生命周期管理。
- 在使用非模式（Negative Patterns）时，自动规则延迟。

所有这些功能在下面的章节中解释。

2.5.2.1 在流模式中的会话时钟角色

当引擎运行在云模式时，会话时钟仅被用于到达事件的时间戳，到达事件并没有一个先前定义的时间戳属性。但是，在流模式中，会话时钟承担了一个更为重要的角色。

在流模式中，会话时钟负责保持当前的时间戳，并且根据它，在事件的老化时、在多源同步流时、在时间表远期任务时，等等，引擎进行所有的时间计算。

要了解如何配置和使用会话时钟的实现，请查看文档有关会话时钟的部分。

2.5.2.2 在流模式中的非模式（Negative Patterns）

与云模式相比较，云模式的非模式（Negative Patterns）行为不同于流模式中的。在云模式中，引擎假定所有的事实和事件是提前知道的（没有时间流的概念）。因此，非模式（Negative Patterns）是立即被运算的。

当运行在流模式时，具有时间约束的非模式（Negative Patterns）可能要求引擎在激活一个规则前等待一个时间段。时间段由引擎用一种方法自动计算出来，用户不需要使用任何技巧来达到预期的效果。

例如：

例子 2.11 在匹配后，马上激活规则。

```
rule "Sound the alarm"
when
```

```
    $f : FireDetected( )
    not( SprinklerActivated( ) )
then
    // sound the alarm
end
```

上面的规则，没有要求延迟规则的时间约束，因此，规则立即激活。另一方面，下面的规则，必须在激活前等待 10 秒，因为它需要花 10 秒用于激活洒水装置（sprinklers）。

例子 2.12 一条规则，由于时间约束，自动延迟激活。

```
rule "Sound the alarm"
when
    $f : FireDetected( )
    not( SprinklerActivated( this after[0s,10s] $f ) )
then
    // sound the alarm
end
```

这种行为允许引擎在同时处理非模式（Negative Patterns）和时间约束时保持一致性。上面的与下面编写的规则是一样的，但不承担用户计算和显式编写适当的限期参数：

例子 2.13 使用显式期限（duration）参数的相同规则。

```
rule "Sound the alarm"
    duration( 10s )
when
    $f : FireDetected( )
    not( SprinklerActivated( this after[0s,10s] $f ) )
then
    // sound the alarm
end
```

2.6 滑动窗口

滑动窗口是一种方法，用于限定有趣事件范围作为一个属于一个不断移动窗口的整体。最常见的两种滑动窗口实现是基于时间的窗口和基于长度的窗口。

下面的章节会详细说明它们。

重要：滑动窗口只当引擎运行在流模式时可用。有关流模式如何工作的详情，请查看事件处理模式章节。

2.6.1 滑动时间窗口

滑动时间窗口允许用户编写规则，其将仅匹配在最近的 X 时间单元内发生的事件。

例如，如果用户希望只考虑发生在最近 2 分钟内发生的股票记号（Stock ticks），该模式是这样的：

```
StockTick() over window:time( 2m )
```

Drools 使用 "over"关键字关联窗口到模式。

一个更精细的例子，如果用户希望在过去最近的 10 分钟从一个传感器中读取的平均温度高于阀值时用声音报警，该规则是这样的：

例子 2.14 整个时间窗口的累积值

```
rule "Sound the alarm in case temperature rises above threshold"
when
    TemperatureThreshold( $max : max )
    Number( doubleValue > $max ) from accumulate(
```

```
        SensorReading( $temp : temperature ) over window:time( 10m ),
        average( $temp ) )
then
    // sound the alarm
end
```

引擎会自动丢弃任何超过 10 分钟的传感器阅读（SensorReading），并保持计算的平均一致。

2.6.2 滑动长度窗口

滑动长度窗口工作的方式与时间窗口相同，但丢弃事件是根据到达的新事件而不是时间流。

例如，如果用户希望只考虑发生在最近的 10 个 IBM 股票记号（Stock ticks），它们与多老是无关的，该模式是这样的：

```
StockTick( company == "IBM" ) over window:length( 10 )
```

正如你所见，模式是与上面章节的表示是相似的，然而不是使用了 window:time 定义滑动窗口，而是使用了 window:length。

使用上面章节中一个相似的例子，如果用户希望在过去最近的 100 阅读从一个传感器中读取的平均温度高于阀值时用声音报警，该规则是这样的：

例子 2.15 整个长度窗口的累积值

```
rule "Sound the alarm in case temperature rises above threshold"
when
    TemperatureThreshold( $max : max )
    Number( doubleValue > $max ) from accumulate(
        SensorReading( $temp : temperature ) over window:length( 100 ),
```

```
        average( $temp ) )
then
    // sound the alarm
end
```

引擎只保持最近的 100 个阅读。

2.7 知识库分割

警告：这是一个试验功能，在将来会有所变动。

典型的 Rete 算法通常使用单线程被执行。如 Dr. Forgy 在几次机会中证实一样，该算法本身是可以并行的。Drools 实现的 ReteOO 算法通过规则库分割支持粗粒度并行。

当应用这个选项时，规则库会被分割成几个独立的分割，并且通过分割一个工作者线程池会被用来传播事实。该实现保证，对一个给定的分割至多有一个工作者线程被用来执行任务，但是在一个时间可能有多个分割是“活动”的。

这一切对用户都是透明的，但除了异步执行的所有工作内存的动作（插入/撤销/修改）。

重要：该功能启用了并行的 LHS 计算，但是没有改变规则引发的行为，即，根据冲突解决方案策略，规则将按顺序连续引发。

2.7.1 什么时候分割是有用的

知识库分割对特殊情况是一个非常强大的功能，但它不是一个普通情况的解决方案。要清楚是否这个功能对一个给定的情况有用，用户可以按照以下的清单：

1. 你的硬件包含多个处理器吗？

2. 你的知识库会话处理高容量的事实吗？

3. 你的规则的 LHS 计算开销高吗？（例如：开销高的“from”表达式）

4. 你的知识库包含成百或更多的规则吗？

如果上面的答案都是“yes”，那么这个功能将可能全面提高你的规则库的计算性能。

2.7.2 如何配置分割

要启用知识库分割，设置下面的选项：

例子 2.16 启用多线程计算（分割）

```
KnowledgeBaseConfiguration config =
KnowledgeBaseFactory.newKnowledgeBaseConfiguration();
config.setOption( MultithreadEvaluationOption.YES );
```

等价的属性是：

```
drools.multithreadEvaluation = <true|false>
```

这个选项的默认值为"false"(禁用)。

2.7.3 多线程管理

Drools 为用户提供了一个简单的配置选项，用于控制工作者线程池的大小。

要定义线程池的最大值，用户可以使用下面的配置选项：

例子 2.17 设置用于规则计算的最大线程数为 5

```
KnowledgeBaseConfiguration config =
KnowledgeBaseFactory.newKnowledgeBaseConfiguration();
config.setOption( MaxThreadsOption.get(5) );
```

等价的属性为：

```
drools.maxThreads = <-1|1..n>
```

这个配置的默认值为 3，而一个负数意味着引擎会按规则库中存在的分割尽可能繁殖线程。

警告：使用一个负数设置这个选项通常是危险的。所以，始终用一个合理的线程正数设置它。

2.8 事件的内存管理

重要：事件的自动内存管理，只当引擎运行在流模式时才被执行。有关流模式如何工作的详情，请查看事件处理模式章节。

引擎运行在流模式的好处之一是：当一个事件的由于时间约束可能不再匹配任何规则时，引擎可以侦测到它。当发生这种情况时，引擎可以无副作用从会话中安全地撤销事件，并释放该事件使用的资源。

有两种基本方法，用于引擎计算一个给定事件的匹配窗口：

●·显式地，使用到期策略。

●·隐式地，分析有关事件的时间约束。

2.8.1 显式到期偏移量

第一种方法，允许引擎计算一个给定事件类型感兴趣的窗口是通过显式设置它。要做它，只使用声明语句，并为事实类型定义一个到期：

例子 2.18 显式为 StockTick 事件定义一个 30 分钟的到期偏移量。

```
declare StockTick
    @expires( 30m )
end
```

上面的例子显式为 StockTick 事件定义一个 30 分钟的到期偏移量。在这个时间之后，假定没有规则仍然需要该事件，引擎会终止并自动从会话中删除该事件。

2.8.2 推理到期偏移量

另外一个方法，引擎为一个给定事件计算到期偏移量是隐式，通过在该规则中的时间约束。例如，假设以下规则：

例子 2.19 使用时间约束的例子规则

```
rule "correlate orders"
when
    $bo : BuyOrderEvent( $id : id )
    $ae : AckEvent( id == $id, this after[0,10s] $bo )
then
    // do something
end
```

分析上面的例子，只要一个 BuyOrderEvent 匹配，引擎自动计算它，需要储存它长达 10 秒，等待匹配 AckEvent。所以， BuyOrderEvent 的隐式到期偏移量将会是 10 秒。另一方面，AckEvent，只能匹配现有的 BuyOrderEvent，因此它的到期偏移量将会是 0 秒。

引擎将会对整个规则库做这种分析，并且为每个事件类型找出偏移量。只要一个隐式的到期偏移量与显式的到期偏移量相冲突，那么引擎会使用两个中的大者。