## 运行步骤

首先将项目导入到eclipse或其他IDE中，开发环境需要安装maven工具，右键pom.xml，点击install即可，安装成功后，会生成natty的jar包，在该项目中，采用Antlr的3.2版本对词法文件进行编译。

为了便于开发，可以安装Antlr的eclipse插件，具体的安装过程可参照，<http://www.cnblogs.com/justinzhang/archive/2012/02/24/2366878.html>，该插件可以支持自动编译，语法结构的可视化以及自动化的测试。

## 程序处理过程

### 项目结构

#### src/main/java

* com.joestelmach.natty.generated：存放词法文件以及自动生成的代码。
* com.joestelmach.natty：存放核心代码，包括Parser主类、WalkerState以及监听类等。

ANTLRNoCaseInputStream：继承ANTLRInputStream，将所有字符转化为小写，在定义token时，只需要考虑小写即可，不用修改。

CalendarSource：内含ThreadLocal<Date> \_baseDate成员，该处的ThreadLocal保证每一个线程都有一个baseDate，每一次解析都会获取一个GregorianCalendar，在使用时，需要先设置baseDate。

DateGroup：返回的日期类型，在该类中，包含了该日期的语法树、解析位置以及相关的信息，不用修改。

Holiday和Season：枚举了假期和季节，在本项目中没用到，不用修改。

IcsSearcher：如果收到holiday，会调用net.fortuna.ical4j.model.Calendar对象，查询日期，不用修改。

NattyTokenSource：实现TokenSource，不用修改。

ParseListener：继承BlankDebugEventListener类，记录解析过程中的匹配规则、匹配位置等信息，最后会生成具体的ParseLocation以及对应的Rule，包含的解析信息够全面。

ParseLocation：rulename、text、line、start、end等，Start和end表示了该rule所在的区间树。

Parser：核心类，完成解析的三个步骤，见2.2节。

WalkerState：对语法树进行遍历，完成日期的解析，在该类中采用CalendarSource 中的GregorianCalendar存放日期值。

JodaWalkerState：该方法与WalkerState的作用一样，在该类中采用了JodaDateTime而不是GregorianCalendar存放日期，因此不需要CalendarSource类统一提供GregorianCalendar，JodaDateTime更加灵活，无需考虑并发，但每次parse都会新建日期对象。在后期的并发测试中，与WalkerState性能差不多。除此之外，我们还对11-08-12这种年只有两位的格式进行了进一步分析，我们首先分析成2011-08-12，然后再将日期分析2012-11-8日，我们去离现在最近的日期为解析之后的日期。

* com.joestelmach.natty.service：包含了采用jetty封装好的服务接口。

DateParserService：主函数，分发请求。

DateParserServlet：处理post请求，返回Json数据。

DateParser：执行具体的解析，将结果转化为Json数据。

* com.joestelmach.natty.tool:几个工具类。

China2Unicode产生对应的unicode值，ExtractStaticMember用来处理自动生成代码后产生的“static initializer over 65525 limits”错误，该类主要将\_transitionS结尾的字符串数组提取到IStaticMember接口文件中。

#### src/main/resources:

resource文件夹下存放着解析日期用的网页即js文件。

其他文件都是配置文件。

#### src/test/java:

com.joestelmach.natty:对日期和时间的测试代码。

com.joestelmach.natty.grammar：对具体语法的测试。

#### src/test/resources:

测试数据。

### 执行流程

Parser类中的parse函数是处理的主函数，在parse中程序的处理流程大致分为三个阶段，第一个阶段是token流的获取与分割，调用词法器分析词法，产生相应的token，第二个阶段对单个token流进行语法分析，并获得日期，第三个阶段是对解析失败的token流进行进一步分析。

#### token流的获取与分割

Parser.java文件中的parse(String value)函数是核心函数，在该函数中完成上述三个步骤的执行，首先创建一个词法对象，DateLexer lexer = new DateLexer(input);该词法对象会分析输入，产生响应的token流，然后调用collectTokenStreams(TokenStream stream)函数对该流进行分割处理。在该函数中会循环读入token。

* 如果记录当前tokenlist的currentGroup为空，且当前的token为初始token，新建currentGroup。
* 否则，
* 如果当前token为UNKNOWN或者.WORD，说明该token不是合法的token，在之前的版本中，作者会截断该token流，并将之前的的tokenlist加入到groups中，这样会造成日期的缺失，如2012-9-5 lasted updated 20:20，这俩个相近的时间本来是一体，如果遇到lasted就将其分成两段，则会解析成两个日期，为此，我们做了修改。当遇到不认识的字符时，不立刻截断，而是记录两个可识别字符之间的举例，如果举例小于4，则说明两段具有较强的关联，就不会强行截取。如果大于4，则截断token 流，并将当前的token list加入到group中。
* 如果当前token为LEFT\_PARENTHESIS或者是星期，我们就会直接忽略，因为在抓取的日期中，如果只有星期，不好判定具体时间，如果有星期也有日期，则直接解析日期即可。

以上就是对token流的分割过程，分割结束之后，会将stream list返回，一个stream表示一个可解析的日期格式。

#### 单个token流的解析

对应singleParse(TokenStream stream)函数，该函数对具体的流进行解析。首先调用dateparser对象，解析该stream，返回Tree。

ParseListener listener = new ParseListener(); //监听解析过程中具体的匹配过程

DateParser parser = new DateParser(stream, listener);

DateParser.parse\_return parseReturn = parser.parse();

Tree tree = (Tree) parseReturn.getTree(); //返回生成的语法树。

接下来，首先调用生成的TreeRewrite对象，重新规范一下树的结构。

CommonTreeNodeStream nodes = new CommonTreeNodeStream(tree);

TreeRewrite s = new TreeRewrite(nodes);

tree = (CommonTree) s.downup(tree);

然后遍历树。

nodes = new CommonTreeNodeStream(tree);

nodes.setTokenStream(stream);

DateWalker walker = new DateWalker(nodes);

walker.getState().setDefaultTimeZone(\_defaultTimeZone);

walker.getState().setDefaultDate(\_defaultDate);

walker.parse();

最后获取结果。

group = walker.getState().getDateGroup();

#### 进一步分析

对单个的stream解析完成之后，可能获取不到日期，这可能是由于token流的截取有关，也有可能是stream中包含了识别不了的结构，为了进一步分析，采取如下策略。

首先从后往前依次删除token，直到token的个数小于4，这样做主要是为了去掉多余的字符，如2012-6-5 10:20 at …，由于at也是可识别字符，但是在语法分析时不能匹配。如果执行完之后，仍不能获得日期，则我们再从前往后，找到新的开始token，继续进行解析。

## 词法文件定义

总共包含五个词法或语法文件DateLexer，NumericRules，DateParser，TreeRewrite以及DateWalker。DateLexer文件中是词法文件，定义了DateLexer词法器，基本的Token定义均在此文件中；NumericRules是语法文件，定义了NumericRules语法器，该文件主要是提供对数字语法的定义，如日期数字、时间数字等；DateParser是语法文件，所有对日期格式的语法定义均在此文件中，该语法文件DateLexer产生的Token词汇库；TreeRewrite也使用DateParser的词汇库，该文件不是在解析过程中使用，而是在生成语法树之后，对语法树进行合并修复等功能；DateWalker是对语法树进行遍历，返回该语法树所对应的日期和时间。

### DateLexer.g

定义了基本的词法，包括月份、星期、时间、时区、数字以及分隔符。每种类型都包括英文和汉语部分。

JANUARY : 'january' 's'? | 'jan' DOT? ;

FEBRUARY : 'february' 's'? | 'feb' DOT? ;

//汉语的月份采用语法的方式定义，如果采用词法的方式定义，

//如*JANUARY* : (CINT\_1 | INT\_1 ) WHITE\_SPACE? MONTH;这样的定义有可能在生成的java代码中，造成单个函数超过65526行的错误，因为为了确定具体的token，antlr会对有歧义的起始词进行超前查找，知道确定该词的token,如果有歧义的词过多，就会生成大量的判断代码。

SUNDAY:'sunday' 's'? | 'sun' DOT? | 'suns' DOT? | '\u661f\u671f\u5929'| '\u5468\u5929' | '\u5468\u65e5';//星期天、周天、周日。

MINUTE : 'minute' | 'minutes' | 'min' DOT? | 'mins' DOT?| '\u5206' WHITE\_SPACE? '\u949f'?;//分钟

AT : 'at' | '@' | '\u5728';//在

UTC : 'utc' | 'gmt' | 'z';

CINT\_1 : '\u4e00'; //一

CINT\_2 : '\u4e8c'; //二

INT\_1 : '1';

INT\_2 : '2'；

THREE : 'three';

FOUR : 'four';

FIFTH : 'fifth';

SIXTH : 'sixth';

COLON : ':' | '\uff1a';// ：

### NumericRules.g

该文件根据DateLexer.g文件中的词法，对数字进行了归类，通过重写，将具体的数字转化为生成的语法树节点。

如：

int\_00\_to\_59\_mandatory\_prefix

: (INT\_00

| int\_01\_to\_12

| int\_13\_to\_23

| int\_24\_to\_31

| int\_32\_to\_59) -> *INT*[$int\_00\_to\_59\_mandatory\_prefix.text]

;

表示将00-12,13-23,24-31和32-59范围的数字重写为INT[‘value’]类型。上述定义的数字用来表示分钟和秒。

int\_00\_to\_99\_mandatory\_prefix：用来表示年份。

int\_1\_to\_9\_prefix：表示1-9。

int\_01\_to\_12\_optional\_prefix：表示01-12以及0-9。用于表示月份。

int\_00\_to\_23\_optional\_prefix：表示00-23以及0-9，用于表示小时。

int\_01\_to\_31\_optional\_prefix：表示01-31以及0-9，用于表示天数。

int\_four\_digits：定义了四位日期格式。

spelled\_or\_int\_01\_to\_31\_optional\_prefix：包括了int\_01\_to\_31\_optional\_prefix以及拼写的one,two等。

spelled\_or\_int\_optional\_prefix：1-9999，以及one..thirty-one。

### DateParser.g

该文件定义了详细的匹配语法规则，起始入口是parse语法，

parse

: empty ((recurrence)=>recurrence | date\_time\_alternative)

;

Empty什么具体值也没有，定义该语法，主要目的是通过最后生成的FOLLOW\_empty\_in\_parse186函数，获得empty后的有效字符，即合法的起始字符。Recurrence本来是支持every类型的日期，在本系统中，由于用于解析具体的日期，暂时没有用到。

(recurrence)=>recurrence结构称为语法谓词，这种方式主要是为了防止二义性，date\_time\_alternative和recurrence可能具有相同的起始字符，采用(recurrence)=>recurrence，表示先检验是否符合recurrence，如果符合则匹配第一个，否则匹配第二个。

date\_time

: (

explicit\_time (time\_date\_separator date)? |

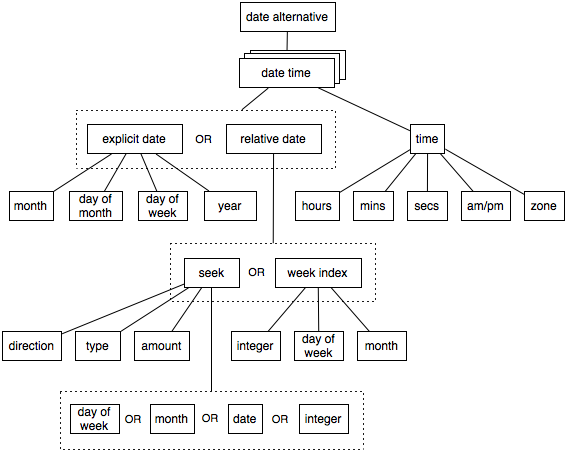
(date)=>date (date\_time\_separator explicit\_time)

) -> ^(*DATE\_TIME* date? explicit\_time?)

| relative\_time -> ^(*DATE\_TIME* relative\_time?)

;

将日期分为两大部分，一部分是date，另一部分是time，根据date和time的相对位置，定义了不同的连接符， -> ^(DATE\_TIME date? explicit\_time?)表示重写功能，即将符合本规则的语法最后转化为以DATE\_TIME为根节点，以date、explicit\_time为子树的语法树。完成的语法树如下图所示。Date对应explicit\_date或relative\_date子树，explicit\_time对应time子树。



#### 2.3.1 date

date

: formal\_date

| relaxed\_date

| relative\_date

;

主要包括三部分：formal\_date、relaxed\_date以及relative\_date。

formal\_date：只包括纯数字的日期。具体支持1979-02-28, 79-02-28，1/02/1980以及1/二1980这四种格式。其中分隔符可以是’-‘，’/’，’.’，’ ‘以及’，’。分隔符的个数可以有一个或多个。

formal\_date

// year first: 1979-02-28, 80/01/02, etc.

: formal\_year formal\_date\_separator+ formal\_month\_of\_year formal\_date\_separator+ formal\_day\_of\_month

-> ^(*EXPLICIT\_DATE* formal\_month\_of\_year formal\_day\_of\_month formal\_year)

// year last: 1/02/1980, 4 digit year is reqired

| formal\_month\_of\_year formal\_date\_separator+ formal\_day\_of\_month formal\_date\_separator+ formal\_year\_four\_digits

-> ^(*EXPLICIT\_DATE* formal\_month\_of\_year formal\_day\_of\_month formal\_year\_four\_digits)

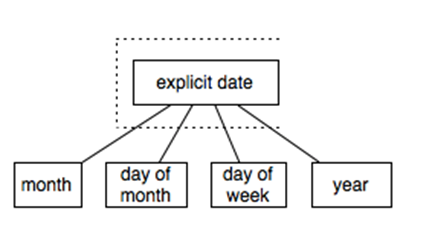
// year last: 1/er/1980 2 or 4 digit year is acceptable

| formal\_day\_of\_month formal\_date\_separator+ cformal\_month\_of\_year formal\_date\_separator+ formal\_year

-> ^(*EXPLICIT\_DATE* cformal\_month\_of\_year formal\_day\_of\_month formal\_year)

;

该类型的日期会生成类似的子树。



relaxed\_date：包括松散的日期类型，具体支持21 Nov 1997，Jan 21, 1997以及1988年4月5日，月份支持jan到december，1月-12月以及一月到12月。年份支持两位或四位的数字。

// relaxed date with a spelled-out or abbreviated month

relaxed\_date

:

(

// The 31st of April in the year 2008

// RFC822 style: Fri, 21 Nov 1997

relaxed\_day\_of\_month *DAY*? formal\_date\_separator\* (*OF* *WHITE\_SPACE*)? (relaxed\_month|crelaxed\_month) formal\_date\_separator\* relaxed\_year\_prefix? relaxed\_year *YEAR*?

// Jan 21, 1997 Sun, Nov 21 yiyue 21,1898

| (relaxed\_month|crelaxed\_month) formal\_date\_separator\* relaxed\_day\_of\_month *DAY*? formal\_date\_separator\* relaxed\_year\_prefix? relaxed\_year *YEAR*?

) -> ^(*EXPLICIT\_DATE* relaxed\_month? crelaxed\_month? relaxed\_day\_of\_month relaxed\_year)

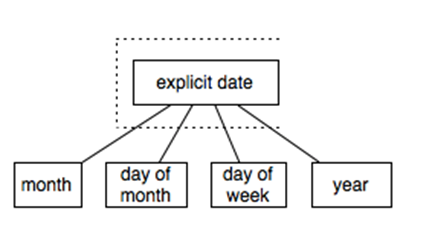
//1989nian 2yue 23

| relaxed\_year\_prefix? formal\_year *WHITE\_SPACE*\* *YEAR* (*SLASH*|*WHITE\_SPACE*)\* (relaxed\_month|crelaxed\_month) (*SLASH*|*WHITE\_SPACE*)\* relaxed\_day\_of\_month *WHITE\_SPACE*\* *DAY*?

-> ^(*EXPLICIT\_DATE* relaxed\_month? crelaxed\_month? relaxed\_day\_of\_month formal\_year)

;

该类型的日期最终也会生成类似的子树。



relative\_date:主要支持两种类型：“3天前”以及“今天”等关键字。“3天前”类型的语法具体支持天、月以及年。“今天”类型的语法具体支持今天、明天、昨天以及前天。

relative\_date

://3 天前

| spelled\_or\_int\_optional\_prefix *WHITE\_SPACE*? relative\_target *WHITE\_SPACE*? relative\_date\_suffix

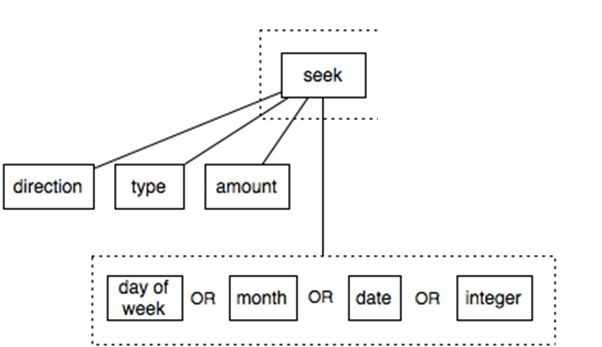
-> ^(*RELATIVE\_DATE* ^(*SEEK* relative\_date\_suffix? spelled\_or\_int\_optional\_prefix relative\_target?))

// today, tomorrow 今天，只有与时间一起存在时才有效。

| named\_relative\_date

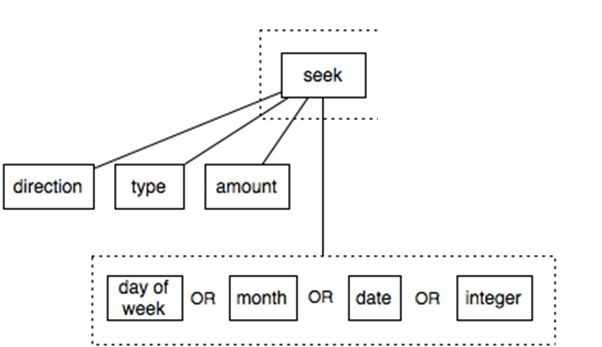
;

该类型的日期会生成以seek为根的子树。



#### 2.3.2 time

时间主要支持relative\_time以及explicit\_time两种时间类型。

relative\_time：主要解决相对时间类型，具体包括：“3小时前”、“半小时前”以及“last minute”。“3小时前”类型包括小时、分钟、秒以及对应英文的翻译。“last minute”目前只支持英文。“半小时前”支持中文和英文。该类型的时间也会生成以SEEK为根节点的子树。

relative\_time

:

//1 hours 30 minutes ago 也支持中文的

spelled\_or\_int\_optional\_prefix *WHITE\_SPACE*? *HOUR* *WHITE\_SPACE*? minutes *WHITE\_SPACE*? *MINUTE* *WHITE\_SPACE*? relative\_time\_suffix

-> ^(*RELATIVE\_TIME* ^(*SEEK* relative\_time\_suffix? spelled\_or\_int\_optional\_prefix *SPAN*["hour"]))

| //支持半小时前，half an hour

*HALF\_HOUR* relative\_time\_suffix

-> ^(*RELATIVE\_TIME* ^(*SEEK* relative\_time\_suffix? *INT*["30"] *SPAN*["minute"]))

| //30分钟前 2小时前等

spelled\_or\_int\_optional\_prefix *WHITE\_SPACE*? relative\_time\_target *WHITE\_SPACE*? relative\_time\_suffix

-> ^(*RELATIVE\_TIME* ^(*SEEK* relative\_time\_suffix? spelled\_or\_int\_optional\_prefix relative\_time\_target))

// this hour等

| prefix *WHITE\_SPACE* relative\_time\_target

-> ^(*RELATIVE\_TIME* ^(*SEEK* prefix relative\_time\_target))

;

explicit\_time：解决具体的时间定义，具体包括“10:05：05 am ”，“23h”，“上午9点5分”以及“上午5:30”。其中“上午9点5分”和“上午5:30”语法可支持上午、下午和晚上。

explicit\_time

: // 6:30:23 am utc格式的日期

hours *COLON* minutes (*COLON* seconds)? (*WHITE\_SPACE*? (meridian\_indicator | (*MILITARY\_HOUR\_SUFFIX* | *HOUR*)))? (*WHITE\_SPACE*? time\_zone)?

-> ^(*EXPLICIT\_TIME* hours minutes seconds? meridian\_indicator? time\_zone?)

//支持6am

| hours *WHITE\_SPACE*? meridian\_indicator (*WHITE\_SPACE*? time\_zone)?

-> ^(*EXPLICIT\_TIME* hours ^(*MINUTES\_OF\_HOUR* *INT*["0"]) meridian\_indicator? time\_zone?)

//今天6点1分格式

| (c\_meridian\_indicator)? hours *WHITE\_SPACE*? *CHOUR* *WHITE\_SPACE*? minutes *WHITE\_SPACE*? *MINUTE* (*WHITE\_SPACE*? seconds)?

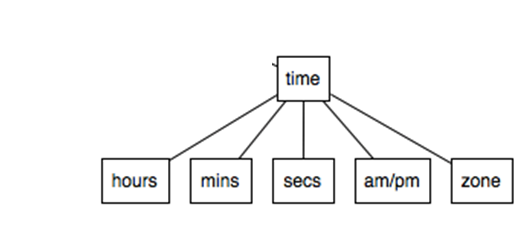
-> ^(*EXPLICIT\_TIME* hours minutes seconds? c\_meridian\_indicator?)

//支持今天6:30格式。

| (c\_meridian\_indicator)? *WHITE\_SPACE*? hours *COLON* minutes (*COLON* seconds)?

-> ^(*EXPLICIT\_TIME* hours minutes seconds? c\_meridian\_indicator?)

该类型的时间最终会生成以time为根节点。



### TreeRewrite

处理重复出现的节点进行合并。

^(*SEEK* *DIRECTION* *SEEK\_BY* *INT* ^(*DAY\_OF\_WEEK* *INT*) ^(*DAY\_OF\_WEEK* dow=*INT*))

-> ^(*SEEK* *DIRECTION* *SEEK\_BY* *INT* ^(*DAY\_OF\_WEEK* $dow))

### DateWalker.g

该词法文件主要是处理生成之后的语法树，根据具体的树结构，调用WalkerState中的相关方法计算该节点上的时间值，如

explicit\_date

: ^(*EXPLICIT\_DATE* ^(*MONTH\_OF\_YEAR* month=*INT*) ^(*DAY\_OF\_MONTH* dom=*INT*)

(^(*DAY\_OF\_WEEK* dow=*INT*))? (^(*YEAR\_OF* year=*INT*))?)

{\_walkerState.setExplicitDate($month.text, $dom.text, $dow.text, $year.text);}

;

会生成explicit\_date函数，在代码的最后会插入walkerState.setExplicitDate($month.text, $dom.text, $dow.text, $year.text);该函数会在遍历完该节点之后调用。

在本文件的@members中声明了使用到的具体walkerstate类，在本系统中，使用WalkerState或者JodaWalkerState都可以。

函数调用，完成对日期的计算，其中$month.text是获取该属性的值。在树遍历完成之后，会调用\_walkerState.captureDateTime()方法，@after标签限定了该函数的执行位置。

date\_time

@after {

\_walkerState.captureDateTime();

}

: ^(*DATE\_TIME* date? time?)

;