5.3 Maintainability-Oriented Construction Techniques

面向可维护性的构造技术

表达式部分参见 MIT6.031 17 18

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

一级标题，二级标题，三级标题，四级标题，注，拓展，不懂

目录

5.3.1 State-based construction

基于状态的编程是一种编程技术，其使用有限状态机来描述程序的行为。

将程序看作是一个有限状态自动机，侧重于对“状态”以及“状态转换”的抽象和编程，且一段程序的执行被分解为“状态转换操作”的有限操作的自动执行。

程序的控制流由状态决定，每一个状态及参数决定了下一个状态转换。

注：通常使用枚举类型 enum 或更加复杂的数据类型定义状态。

注：通常使用二维数组定义状态转换表。

State Pattern 状态模式（）

问题：

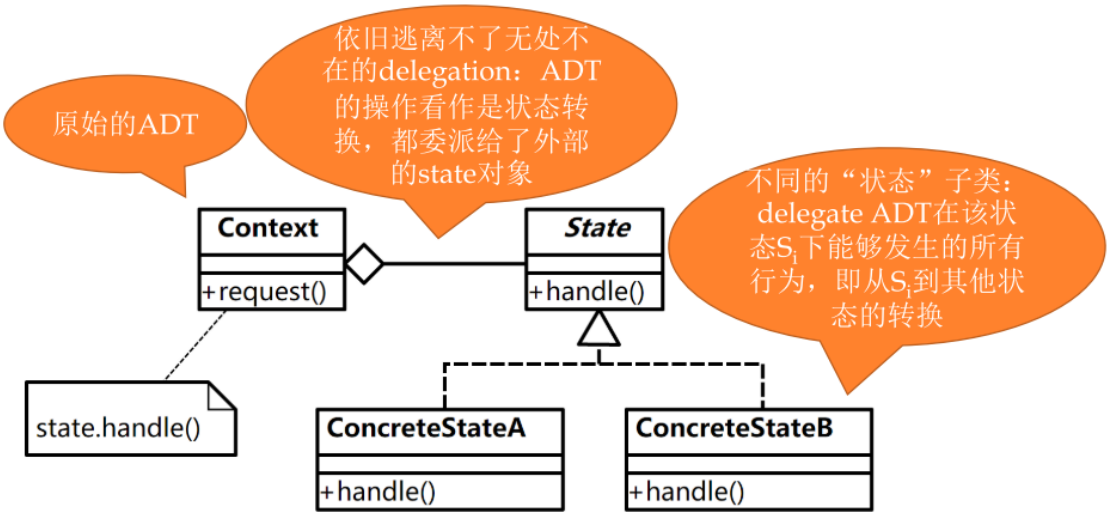
一般而言可以通过 ifelse 分支结构进行基于状态的编程。但是这存在不易于扩展状态的问题。

功能以及特点：

完成状态的转换，且易于扩充状态。

将状态和该状态下的操作合并为一个状态类。

形象图



解释：

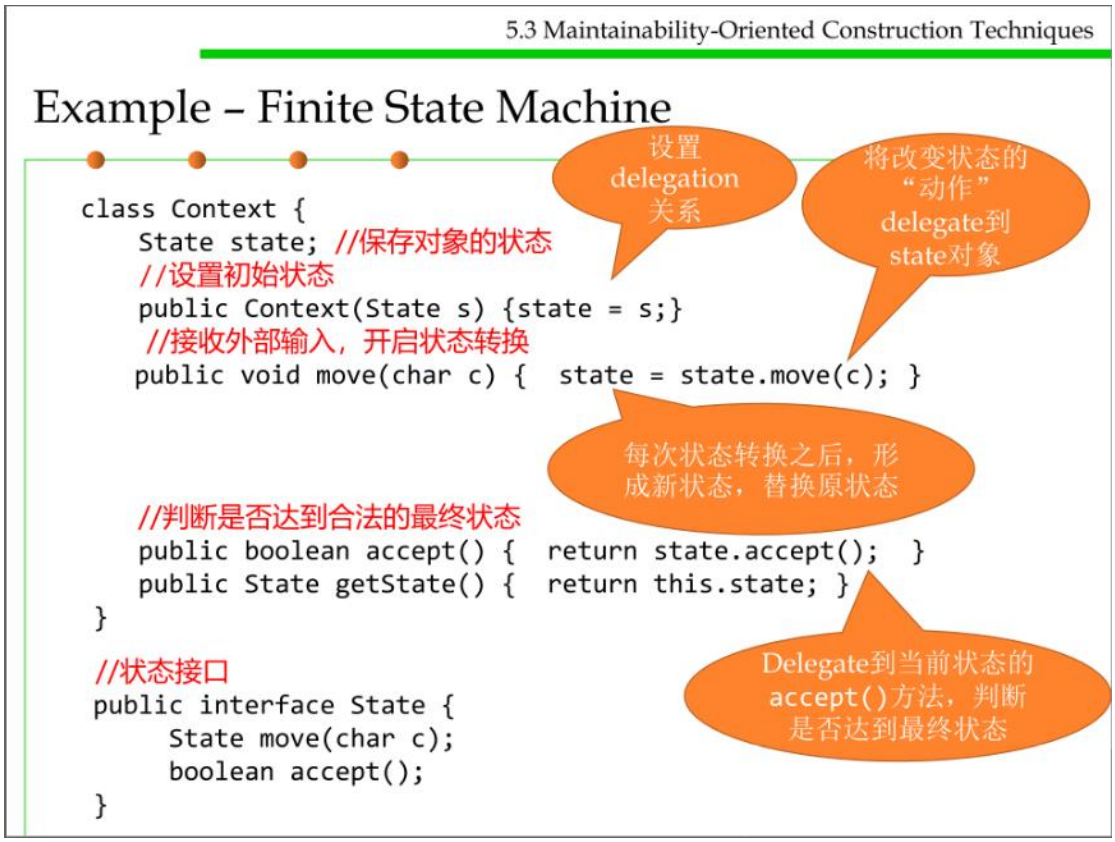
Context 进行状态的初始化，状态之间的转换，终止状态的定义等方法的定义。

接口 State 描述所有状态的共性，State 的每一个实现类 ConcreteStateA，ConcreteStateB 等都描述了一个状态，而 handle（）为其共性的方法。

这样操作使得需要扩展状态，只需要额外定义一个 State 的实现类即可。不过这仍然需要改变别的状态类或者起始状态类，否则无用。

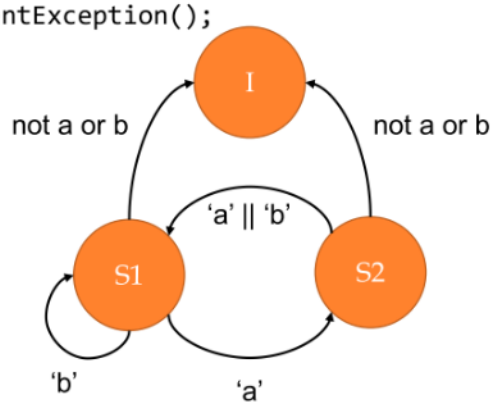
Ex：

在此例中，使用了单例模式，单例模式就是使得某个类在程序运行中只能产生一个对象，其实用的原理是：将构造函数私有化，将某个类的静态成员作为这个对象。



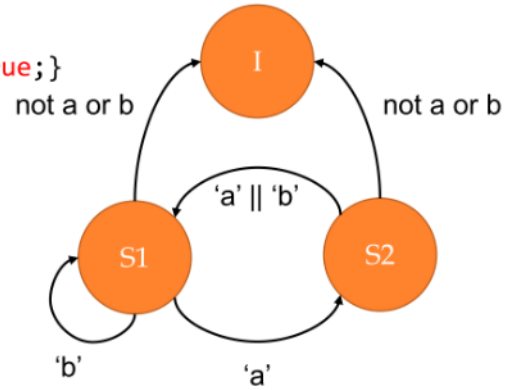
FSM Example – cont.

```
class State1 implements State {
  static State1 instance = new State1();  //singleton模式 (see 8-3)
  private State1() {} //构造器私有，不可生成新实例，单例模式
  public State move (char c) {
     switch (c) {
      case 'a': return State2.instance;    //返回新状态的singleton实例
      case 'b': return State1.instance;
      default: throw new IllegalArgumentException();
     }
  }
  public boolean accept() {
       return false;
  } //该状态非可接受状态
}
```



FSM Example – cont.

```
class State2 implements State {
   static State2 instance = new State2();
   private State2() {}
   public State move (char c) {
       switch (c) {
           case 'a': return State1.instance;
           case 'b': return State1.instance;
           default: throw new IllegalArgumentException();
          }
    }
   public boolean accept() {return true;}
}
```

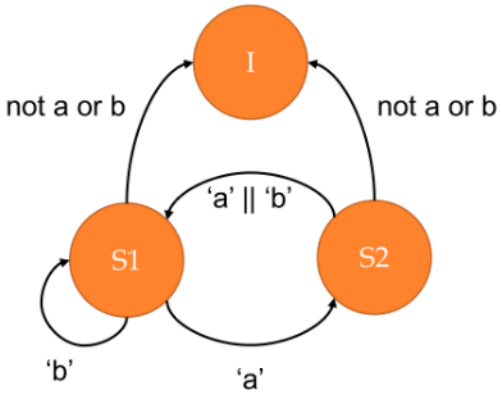


Example

```
public static void main(String[] args) {
    Context context = new Context(State1.instance);
    for (int i = 0; i < args.length; i++) {
        context.move(args[i]);
        if(context.accept()) break;
    }
}
```

给ADT初始状态

根据输入的一组字符，每次用一个字符使状态发生变迁，直到达到最终状态，程序结束。

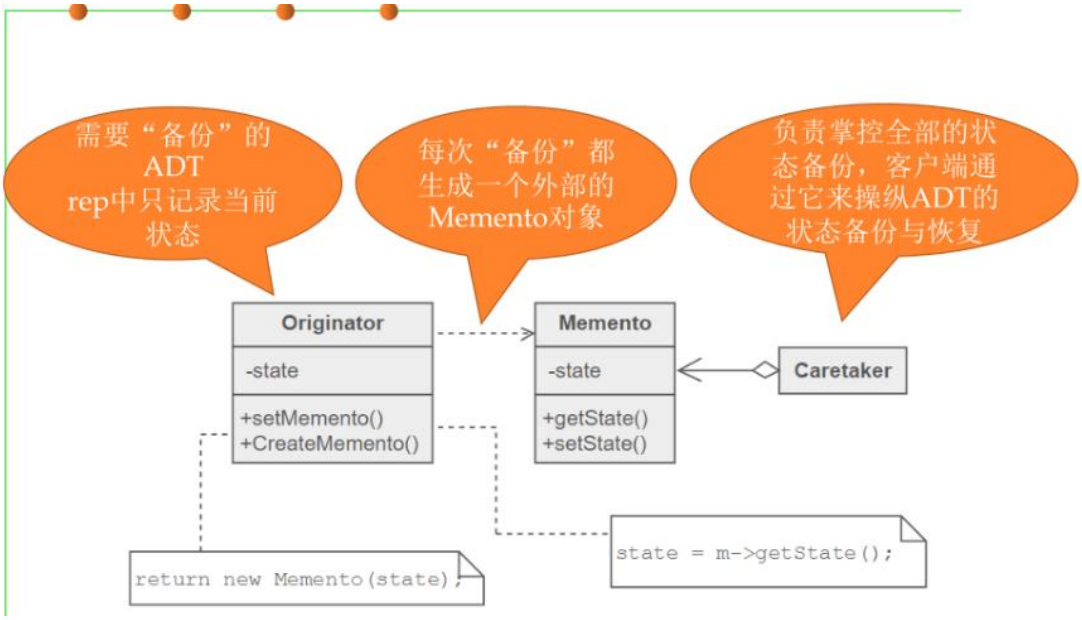


Memento Pattern（备忘录模式）

问题：

需要由当前状态回滚到之前的状态，可以为倒数第一个，倒数第二个……

形象图：



解释：

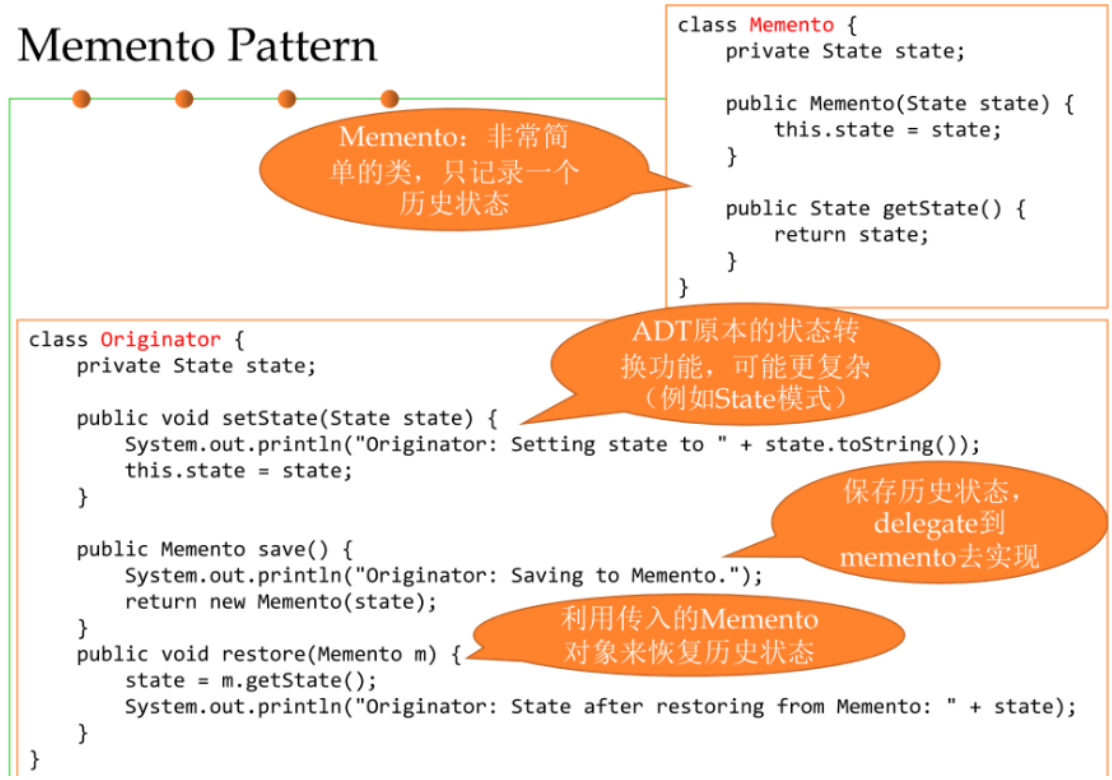
Originator 为当前状态类，也就是需要备份的类，可以将其认为是 State Pattern 状态模

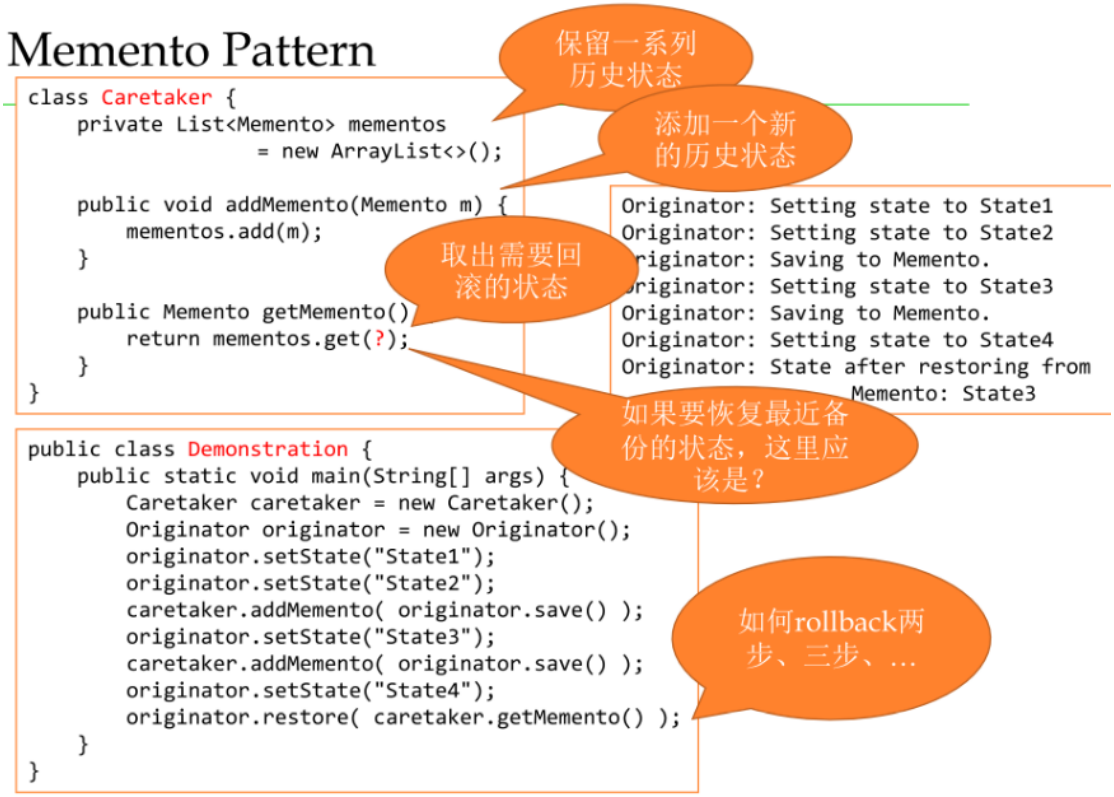
式（）中的 Context 类，其中 state 表示当前类。对于其中方法参加下文。

Memento 表示需要记录状态的类。State 表示需要记录的状态，包含两个方法 get 和 set 方法。Originator 类中的 SetMemento（Memento）方法表示为由当前状态恢复为 Memento 状态，CreatMemento（）表示由当前状态创建一个 Memento 的对象。

Caretake 表示创建一个 Memento 的一个集合。存储类。

Ex:





Memento Pattern

```
class Caretaker {
    private List<Memento> mementos = new ArrayList<>();
    public void addMemento(Memento m) { mementos.add(m); }
    public Memento getMemento(int i) {
        if(mementos.size()-i < 0)
            throw new RuntimeException("Cannot rollback so many back!");
        return mementos.get(mementos.size()-i);
    }
}
```

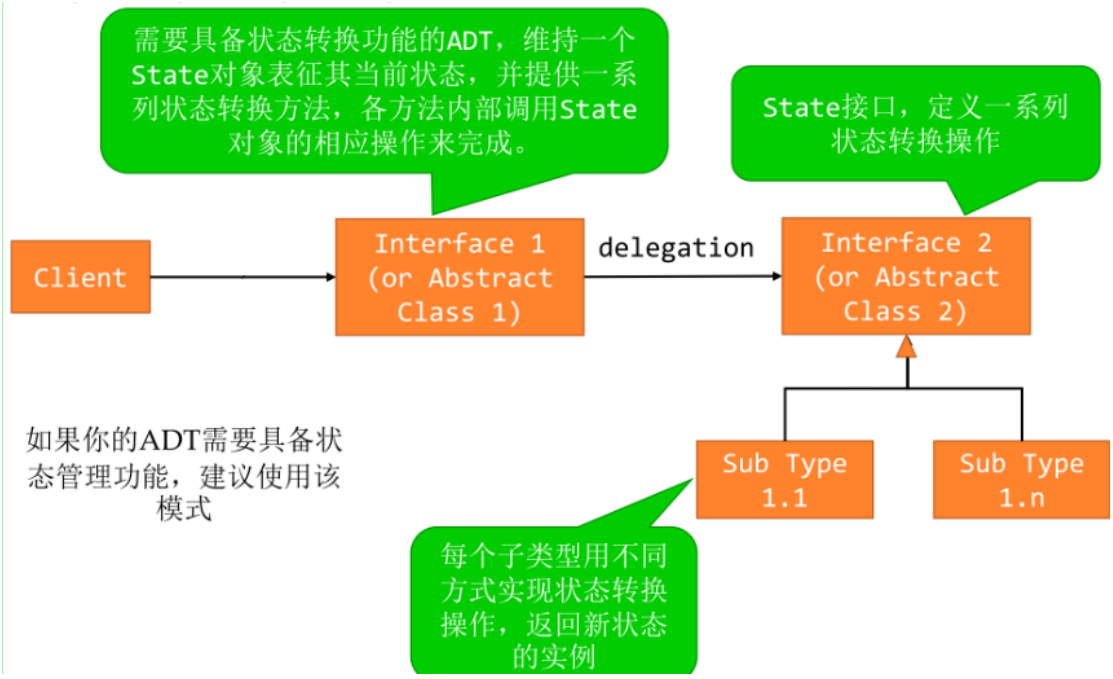
但这里有个潜在bug：每次restore之后，是否应删除某些备忘录？如何继续修改该代码？

```
public class Demonstration {
    public static void main(String[] args) {
        Caretaker caretaker = new Caretaker();
        Originator originator = new Originator();
        originator.setState("State2");
        caretaker.addMemento( originator.save() );
        originator.setState("State3");
        caretaker.addMemento( originator.save() );
        originator.setState("State4");
        originator.restore( caretaker.getMemento(2) );
    }
}
```

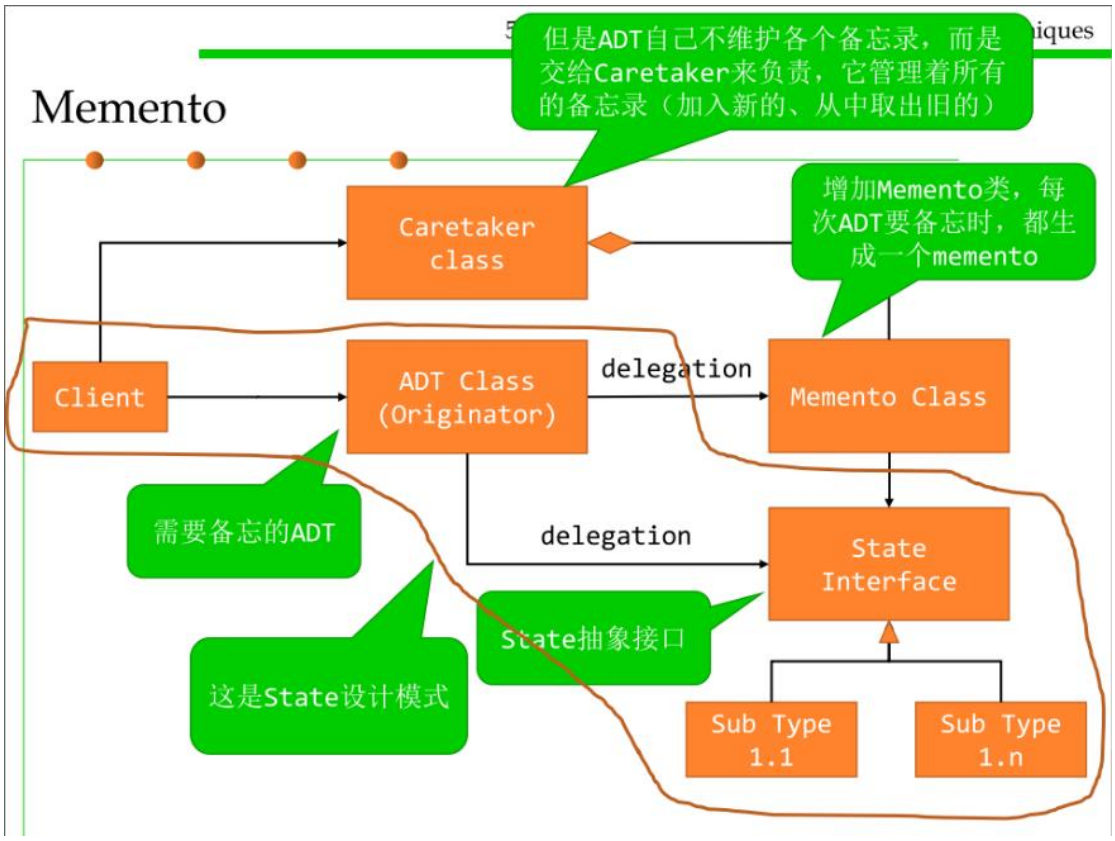
如果你需要支持repeat功能，就不要删除memento对象

此两个设计模式均属于 5.2 节中的共性样式 2.

State Pattern 状态模式（）



Memento Pattern（备忘录模式）



5.3.2 Regular Expressions & Grammars（正则表达式和语法）

产生原因：

一些程序需要读入一些字节或者字符的序列来完成其职责，我们需要设计一种规约使得程序可以处理这些序列：是否满足我们的规约，可以从满足规约的序列中提取到相应的信息，可供程序使用，一般而言是一个数据结构。这种规约一般称作语法。

这些序列包括：

A. 字符串

B. 桌面上的文件。其对应的规约一般为文件格式。

C. 互联网发送的消息，其对应的规约一般为网络流协议。

D. 由用户输入的命令行参数，其对应的规约一般为命令行接口协议。

而存在一种特殊的语法形式成为正则表达式，其除了具有辨别和提取之外，还是一种广泛使用的字符串处理工具，可以反汇编字符串，从字符串中提取信息，以及转换字符串。

Grammars（语法）

描述字符序列的一种紧凑的表示形式，就叫做语法。

Terminals（终端）

The literal strings。

Literal： A letter or symbol that stands for itself as opposed to a feature, function, or entity associated with it in a programming language.

可以认为这是一个标量，其表示的意思就是它本身的意思。与变量相对比。

其被称为终端也是因为它在解析树（一种表示字符序列的结构，下文会介绍）的叶子，它不可以进行扩展。

通常一个终端会写在单引号中来表示区分，例如：'http' 。

Nonterminal（非终端）

可以认为这是一个变量，其代表着一个或多个字符串构成的集合。

Productions（产品）

可以认为是一个表达式，其表示了一个非终端的具体定义。

其具有这样的形式：

```
nonterminal ::= expression of terminals, nonterminals, and operators
```

root（根）

根是非终端的一个特例，其在完全的一组表达式集合中，其具有这样的性质：

- ✧ 在该集合中，根有且仅有一个
- ✧ 在该集合中，根仅出现在每个表达式的“: : =”的左端

Grammar operators（语法操作符）

表达式中使用的，将终端以及非终端联合，以表达这些要素的关系的一些符号。

Repetition（重复操作符）：“*”，即星号

```
x ::= y*        // x matches zero or more y
```

Concatenation（串联操作符）：“ ”，即空格

```
x ::= y z       // x matches y followed by z
```

Union（或操作符）：“|”，即竖线

```
x ::= y | z     // x matches either y or z
```

操作符的优先级：

依照前人留下的习惯，括号>重复操作符>串联操作符>或操作符。

也可以使用括号来更加清楚地表示这些优先级。

Ex:

表示 `http://mit.edu/`, `http://google.com/`, and `http://stanford.edu/`

```
url ::= 'http://' hostname '/'

hostname ::= 'mit.edu' | 'stanford.edu' | 'google.com'
```

下列表示很好得到结果，但需要注意的是，*所代表的重复，可以是任意多个小写字符的集合，不需要是同一个小写字符，理解清楚“|”与“*”共用的含义。

还有除了通常意义的字符串，还有 word 均为 null。

```
url ::= 'http://' hostname '/'

hostname ::= word '.' word

word ::= ('a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i'
            | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q'
            | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | '
z')*
```

如果想要使得它表示一个正常的 url，可以这样定义：

```
word ::= ('a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i'
            | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q'
            | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | '
z')
        ('a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i'
            | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q'
            | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | '
z')*
```

拓展：更多的运算符

上面所讲的三种运算符为基本运算符，这里所展示的不过是一些语法符号甜头，它们都可以由上面的运算符进行表示。

0 or 1 occurrence（不出现或者只出现一次）：“?”即问号

```
x ::=  y?      // an x is a y or is the empty string
```

1 or more occurrences（出现一次或出现多次）：“+”即加号

```
x ::= y+       // an x is one or more y
               //    equivalent to  x ::= y y*
```

表示"[n]"内数字 n 或者"[n，m]"所表示范围大于等于 n 小于等于 m（缺失为无穷）的"[ ]"前缀的终端或非终端，表示方式："y[ ]"

具体用法详见下方。

```
x ::= y{3}     // an x is three y
               // equivalent to x ::= y y y


x ::= y{1,3}   // an x is between one and three y
               // equivalent to x ::= y | y y | y y y


x ::= y{,4}    // an x is at most four y
               // equivalent to x ::=   | y | y y | y y y | y y y
y
               //                    ^--- note the empty string h
ere, so this can match zero y's


x ::= y{2,}    // an x is two or more y
               // equivalent to x ::= y y y*
```

"[ ]"表示方括号内字符的某一个，可以采用"-"进行省略表示。

```
x ::= [aeiou]  // equivalent to  x ::= 'a' | 'e' | 'i' | 'o' | '
u'
```

或者

```
x ::= [a-ckx-z]    // equivalent to  x ::= 'a' | 'b' | 'c' | 'k'
 | 'x' | 'y' | 'z'
```

"[^]"表示除方括号内字符的任意一个

```
x ::= [^a-c]  // equivalent to  x ::= 'd' | 'e' | 'f' | ... | '0
' | '1' | '2' | ... | '!' | '@'
              //                       | ... (all other possibl
e characters)
```

语法中的递归

其递归的形式就是在非终端的产品表达式: : =左侧的部分出现了被定义的终端。
一些递归可以使用操作符将其变为非递归，但有些递归是不可以的

Parse trees（解析树）

依据语法匹配字符序列的过程可以被归纳为一个解析树，它展示了字符序列的每一部分如何与语法的表达式的每一部分相对应。

解析树的叶子是终端，它代表了已经被解析的那一部分，或者说不用解析，他就代表它本身。如果我们将叶子按照某种特定顺序组合，会得到一个符合语法的字符序列。

解析树内部的节点是非终端。每一个非终端，都必须具有其相应的产品（即表达式）。

注: 一个解析树不止由产品（表达式）构成，还有一个确定的符合条件的字符序列，解析树所表现的是该字符序列的每一部分如何与语法的表达式的每一部分相对应。

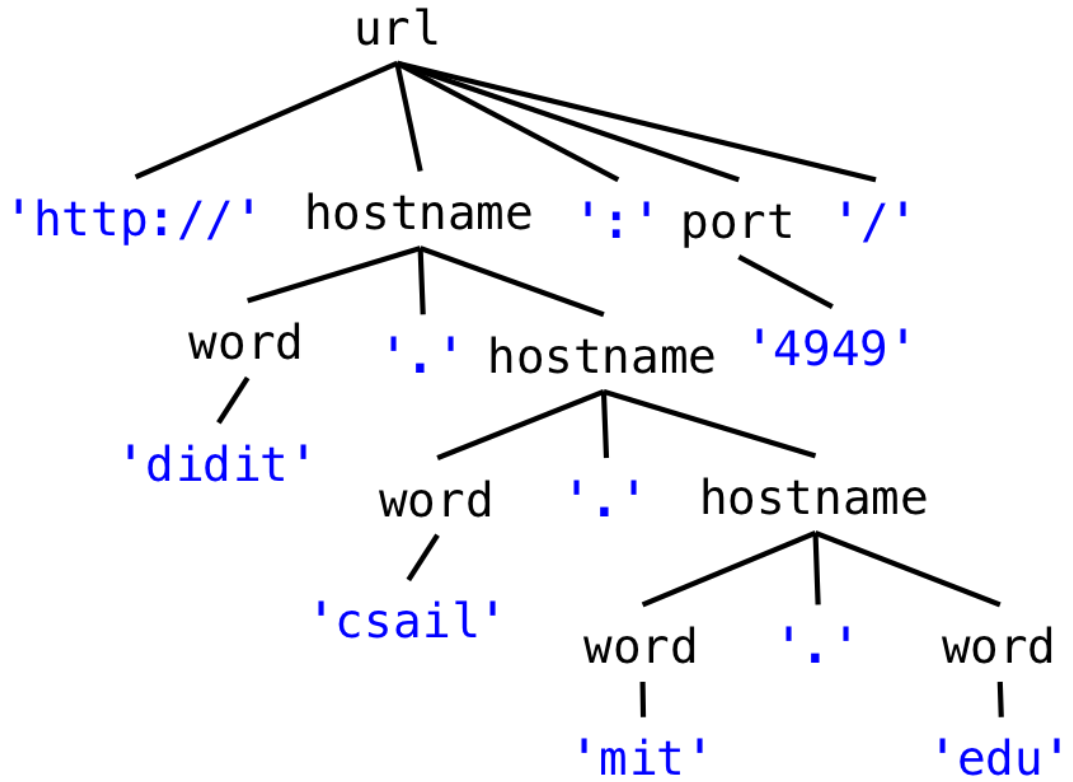
Ex:

```
url ::= 'http://' hostname (':' port)? '/'

hostname ::= word '.' hostname | word '.' word

port ::= [0-9]+

word ::= [a-z]+
```



regular gramma &Regular expressions（正则语法和正则表达式）

正则语法

正则语法就是对于一组完全的词法表达式，用终端和语法操作符替换除根节点之外的非终端节点，以达到下面的形式：

```
root ::= expression of terminals and operators
```

注：存在某些表达式，无法化简为正则语法的形式，产生无法消除的递归。例如 html。

```
Html 的最终化简形式：

html ::=  ( [^<>]* | '<i>' html '</i>' )*
```

Regular expressions（正则表达式）

一个正则表达式就是一个满足正则语法的表达式，去除终端两侧的引号以及空格。产生一个仅仅由终端特征字符，括号，操作符组成的表达式。

正则表达式以一种更加紧凑的方法进行书写，与书写的紧凑相对应，其是更加难以理解的，因为其去除了那些子非终端的解释。但是许多编程语言提供了库支持正则语言，正则语言的匹配速度也远快于一般语言。

Ex:

```
markdown ::= ([^_]* | '_' [^_]* '_' )*    //非正则表达式

markdown ::= ([^_]*|_[^_]*_)*             //正则表达式
```

拓展：正则语言在编程语言的库有其他的一些特性，下面是一些有用的：

```
.   // matches any single character (but sometimes excluding ne
wline, depending on the regex library)
```

```
\d  // matches any digit, same as [0-9]

\s  // matches any whitespace character, including space, tab, newline

\w  // matches any word character including underscore, same as [a-zA-Z_0-9]
```

拓展：反斜线“\”常用于表示一些被充当运算符的终端，下面是一些代替品：

```
\.  \(  \)  \*  \+  \|  \[  \]  \\
```

Ex: 一些编程中应用的例子：

A. 将多个空格替换为单个空格

```
String singleSpacedString = string.replaceAll(" +", " ");
```

B.

```
Pattern regex = Pattern.compile("(?<year>[0-9][0-9][0-9][0-9])-(?<month>[0-9][0-9])-(?<day>[0-9][0-9]");

Matcher m = regex.matcher(string);

if (m.matches()) {

    String year = m.group("year");

    String month = m.group("month");

    String day = m.group("day");

    // Matcher.group(name) returns the part of the string that matched (?<name>...)

}
```

这里的用于`(?<year>...)`抽取被匹配字符串中符合条件的那一部分，然后分配名字。条件为`...`

下面是一个输入示例。

`If this regex were matched against "2025-03-18", for example, then m.group("year") would return "2025", m.group("month") would return "03", and m.group("day") would return "18".`

Context-free grammars（上下文无关语法）

可以用我们的语法系统表示的语言就是上下文无关语言。我们的语法系统是指上面定义的语法。这种语法不依赖于上下文，这显而易见，只有一个根节点是非终端，终端则有自己定义。。

注：上下文无关不属于正则。

Ex：下面是 java 的语法结构。

```
statement ::=
  '{' statement* '}'
| 'if' '(' expression ')' statement ('else' statement)?
| 'for' '(' forinit? ';' expression? ';' forupdate? ')' stateme
nt
| 'while' '(' expression ')' statement
| 'do' statement 'while' '(' expression ')' ';'
| 'try' '{' statement* '}' ( catches | catches? 'finally' '{' s
tatement* '}' )
| 'switch' '(' expression ')' '{' switchgroups '}'
| 'synchronized' '(' expression ')' '{' statement* '}'
| 'return' expression? ';'
| 'throw' expression ';'
| 'break' identifier? ';'
| 'continue' identifier? ';'
| expression ';'
| identifier ':' statement
| ';'
```

MIT 总结：

Machine-processed textual languages are ubiquitous in computer science. Grammars are the most popular formalism for describing such languages, and regular expressions are an important subclass of grammars that can be expressed without recursion.

The topics of today's reading connect to our three properties of good software as follows:

- **Safe from bugs.** Grammars and regular expressions are declarative specifications for strings and streams, which can be used directly by libraries and

tools. These specifications are often simpler, more direct, and less likely to be buggy than parsing code written by hand.

- **Easy to understand.** A grammar captures the shape of a sequence in a form that is easier to understand than hand-written parsing code. Regular expressions, alas, are often not easy to understand, because they are a one-line reduced form of what might have been a more understandable regular grammar.
- **Ready for change.** A grammar can be easily edited, but regular expressions, unfortunately, are much harder to change, because a complex regular expression is cryptic and hard to understand.