Engineer

Reguljärt uttryck

ett mönster som beskriver vilka textsträngar som ger träff

Vad det är

Ett reguljärt uttryck är ett kort mönster som beskriver en mängd textsträngar, så att ett program kan välja ut dem som hör dit. POSIX-standarden, som ges ut av The Open Group och IEEE, inleder sitt kapitel i ämnet med att beskriva reguljära uttryck som ett sätt att välja ut specifika strängar ur en mängd teckensträngar. I vardagligt tal och i namnen på många funktioner och bibliotek krymper begreppet till regex, eller till regexp, som är stavningen i JavaScripts RegExp-objekt och i regexp-paketet i Go.

De flesta tecken i ett mönster står för sig själva: ett `a` i mönstret matchar ett `a` i texten. Några få är speciella. En punkt matchar vilket tecken som helst, `*` upprepar det som står före noll eller fler gånger, ett uttryck inom hakparenteser som `[0-9]` matchar ett tecken ur en mängd, `^` och `$` binder mönstret till början och slutet, och ett intervall som `{4}` kräver ett bestämt antal upprepningar. Därmed beskriver `^[0-9]{4}$` varje sträng som består av exakt fyra siffror, utan andra tecken före eller efter. Standardens eget exempel på ett intervall är `c{3}` tillämpat på `abababccccccd` : det matchar det sjunde till nionde tecknet, de tre första av de sex c:na.

Det finns mer än en dialekt. POSIX definierar själv två: grundläggande reguljära uttryck (BRE), som de flesta av standardens verktyg använder, och utökade reguljära uttryck (ERE), som några använder i stället. C-funktionerna `regcomp()` och `regexec()` erbjuder båda. Skillnaden är inte kosmetisk. I ett ERE betyder `a+` ett eller flera `a` . I ett BRE är `+` ett vanligt tecken, och standarden överlåter åt varje implementation att avgöra om `\+` betyder ett eller flera eller ett bokstavligt plustecken.

Den andra stora familjen härstammar från Perl. ECMAScript-specifikationen slår fast att formen och funktionaliteten hos reguljära uttryck i JavaScript är modellerade efter funktionaliteten för reguljära uttryck i Perl 5. Regexp-paketet i Go beskriver sin syntax som den allmänna syntax som Perl, Python och andra språk använder, och `re` -modulen i Python är en tredje medlem. Dessa motorer delar det mesta av notationen men inte varje detalj. Förkortningsklasserna är ett bra exempel: `\d` (en siffra) och `\w` (ett ordtecken) ser likadana ut överallt, men vad de omfattar beror på motorn och flaggorna, vilket avsnittet om praktiken nedan visar.

Använd begreppet i ett beslut

Använd ”Vad är ett reguljärt uttryck?” i ett konkret beslut. Vi kan knyta begreppet till systemet ni bygger eller köper. Ett team som tar en idé från en skiss till något dina kunder kan logga in i, i stället för att lämna den vidare mellan tre leverantörer.

Se våra tjänster Hör av er →

Granska antagandena bakom ordet

Använd förklaringen och det länkade arbetet för att formulera frågor till projektet. Vi kan undersöka antagandena tillsammans innan ett verktyg väljs. Ledde GIS-kartappen till 10 gånger högre intäkter — 10x intäktstillväxt, med kartprodukten i spetsen

Ledde GIS-kartappen till 10 gånger högre intäkter Hör av er →

Varför det spelar roll

Reguljära uttryck sitter i mycket mjukvara som inte ser ut som textbehandling: validering av formulärfält, filter som plockar ut värden ur loggrader, routningsregler i webbservrar, sök och ersätt i textredigerare. Varje mönster är kort, och därför är det lätt att förbise vad det faktiskt godtar. Två frågor avgör resultatet, och svaret på båda beror på motorn: vilken träff som väljs när flera är möjliga, och hur lång tid det kan ta att hitta en.

POSIX väljer den träff som står längst till vänster och bland dem den längsta: av alla träffar som börjar tidigast i texten vinner den längsta, på engelska leftmost-longest. Standardens exempel är BRE:t `bb*` , som matchar andra till fjärde tecknet i `abbbc` , och ERE:t `(wee|week)(knights|night)` , som matchar alla tio tecknen i `weeknights` . Perl-familjen arbetar annorlunda. Den prövar alternativen i den ordning de är skrivna och tar det första som leder till en träff, det som dokumentationen för Go kallar leftmost-first. Samma mönster på samma text kan därför ge två svar:

```
$ echo ab | grep -oE 'a|ab'
ab
>>> re.search("a|ab", "ab").group()
```

```
'a'
> "ab".match(/a|ab/)[0]
'a'
```

Go erbjuder båda. Funktionen `Compile` använder leftmost-first, medan `CompilePOSIX` begränsar syntaxen till POSIX ERE och byter till leftmost-longest. Även då behåller paketet sin egen regel för vilka delträffar som väljs, och dokumentationen förklarar varför: POSIX-regeln är enligt den oöverkomligt dyr att beräkna och inte ens väldefinierad.

Den andra frågan är tid. `re` i Python backtrackar, liksom de flesta motorer i Perl-familjen. När en del av mönstret inte kommer vidare går motorn ett steg tillbaka och prövar ett annat sätt att matcha det som kom före. Dokumentationen för Python visar idén med `a*a` mot `aaaa` : `a*` tar först alla fyra bokstäverna och lämnar sedan tillbaka en, så att det sista `a` kan matcha. För de flesta mönster är kostnaden osynlig. För vissa växer antalet sätt att pröva exponentiellt med längden på indata, och en sträng på några dussin tecken kan hålla en processor sysselsatt i minuter. MITRE katalogiserar detta som CWE-1333, Inefficient Regular Expression Complexity, och anger ReDoS (regular expression denial of service) och catastrophic backtracking som andra namn på det. Regexp-paketet i Go utformades kring det motsatta löftet. Dokumentationen slår fast att matchningen garanterat körs i tid som är linjär i förhållande till storleken på indata, och tillägger att de flesta implementationer med öppen källkod inte ger någon sådan garanti.

Så ser det ut i praktiken

Ta ett registreringsformulär för kunder i Danmark och Ukraina, och ett förnamnsfält som bara ska godta bokstäver. Det självklara mönstret är `[A-Za-z]+` , och i Python 3.15 beter det sig så här:

```
>>> import re
>>> re.fullmatch(r"[A-Za-z]+", "Mia")
<re.Match object; span=(0, 3), match='Mia'>
>>> re.fullmatch(r"[A-Za-z]+", "Søren")
>>> re.fullmatch(r"[A-Za-z]+", "Олена")
>>> re.fullmatch(r"\w+", "Søren_42")
<re.Match object; span=(0, 8), match='Søren_42'>
>>> re.fullmatch(r"[^\W\d_]+", "Олена")
```

```
<re.Match object; span=(0, 5), match='Олена'>
>>> re.fullmatch(r"[^\W\d_]+", "Søren_42")
>>>
```

Ett anrop som returnerar `None` skriver inte ut en enda rad i det interaktiva skalet, så ASCII-klassen avvisar Søren och Олена (Olena), två helt vanliga namn i Köpenhamn och Kyiv. Byter man till `\w` släpps de igenom, men i Python omfattar `\w` alla tecken som `str.isalnum()` är sant för, plus understrecket, så klassen släpper också in `Søren_42`. Klassen `[^\W\d_]` utesluter tecken som inte är ordtecken, decimala siffror och understrecket. Den tillåter också vissa numeriska tecken som inte är decimala siffror och är därför inget strikt test för enbart bokstäver. För namn på mer än ett ord godtar `[^\W\d_]+( [^\W\d_]+)*` både `Anne Sofie Østergaard` och `Олена Коваль`. Mönstret avvisar fortfarande en `Anne-Marie` med bindestreck, och det är ett skäl att hålla ett namnfält så generöst som verksamheten tål. POSIX-verktyg har samma problem i en annan form: standarden lämnar betydelsen av ett teckenintervall som `A-Z` ospecificerad utanför POSIX-localen och erbjuder klassen `[[:alpha:]]`, där den aktuella localen avgör vilka tecken som hör dit.

Att göra rätt här betyder mer än ett formulär. En av tjänsterna på den här webbplatsen, Internationalisering och lokalisering, beskriver målet som mjukvara som känns naturlig på varje marknad kunden säljer på, inklusive hur namn sorteras. Ett namnfält som avvisar Søren brister långt innan någon sortering sker.

Siffror väcker den motsatta frågan. Ett danskt postnummer består av fyra siffror, så `\d{4}` ser ut som en säker kontroll av `2200`. I Python matchar `\d` i ett textmönster ändå varje decimalsiffra i Unicode, inte bara 0 till 9, och godtar samma postnummer skrivet med arabisk-indiska siffror:

```
>>> s = "\u0662\u0662\u0660\u0660"
>>> print(s, int(s))
٢٢٠٠ 2200
>>> re.fullmatch(r"\d{4}", s)
<re.Match object; span=(0, 4), match='٢٢٠٠'>
>>> re.fullmatch(r"\d{4}", s, re.ASCII)
>>> re.fullmatch(r"[0-9]{4}", s)
>>>
```

JavaScript står på andra sidan. ECMAScript-specifikationen definierar `\d` som exakt de tio tecknen 0 till 9, så `/^\d{4}$/.test(s)` returnerar `false` i Node.js för samma sträng, medan `/^\p{Nd}{4}$/u` returnerar `true`. Ett formulär som kontrolleras i webbläsaren med den ena regeln och på servern med den andra kommer att vara oense med sig självt om samma indata. `int()` i Python läser strängen som 2200, men en databaskolumn, en uppslagstabell eller en annan tjänst gör det kanske inte. Bestäm vilka siffror som avses: `[0-9]` eller flaggan `re.ASCII` när bara ASCII-siffror räcker, och `\p{Nd}` med flaggan `u` i JavaScript när varje decimalsiffra är välkommen.

Var det går fel

Ett mönster som fördubblar tiden för varje bokstav. Ett namnmönster som ska tillåta ord åtskilda av enkla mellanslag skrivs lätt som `([^\W\d_]+ ?)+`. På ett giltigt namn är det snabbt. På en följd av bokstäver med ett tecken det inte kan godta på slutet prövar motorn alla sätt att dela in bokstäverna i ord innan den ger upp, och antalet sätt fördubblas för varje bokstav. På en maskin med Python 3.15 gav den snabbaste av fem körningar för varje längd de här tiderna för `fullmatch` på en sträng av bokstäver följd av `!`:

```
letters   seconds
     18     0.017
     20     0.067
     22     0.27
     24     1.0
     26     4.1
```

Varje bokstav fördubblar tiden, så två bokstäver till fyrdubblar den. Räknar man vidare med den faktorn skulle trettio bokstäver ta ungefär en minut och trettiotvå mer än fyra minuter. Det här är fallet som CWE-1333 beskriver: backtracking-försöken växer exponentiellt med indata, indata behöver inte matcha, och indata kan vara tillräckligt lång. Omskrivningen `[^\W\d_]+([^\W\d_]+)*` beskriver samma namn utan en nästlad kvantifierare, avvisar strängen på 24 bokstäver på ungefär en mikrosekund och en miljon bokstäver följda av `!` på ungefär trettio millisekunder. Ta bort nästlade kvantifierare, begränsa längden på indata innan den matchas, och kör mönster som tar emot opålitlig indata i en motor med garanterat linjär tid, som den i Go.

Sökning där hela värdet ska matcha. I Python godtar `re.search` en träff var som helst i strängen, och `re.match` förankrar bara i början; Python 3.15 ger den det tydligare namnet `prefixmatch` och markerar det gamla namnet som soft deprecated, med en anmärkning om att andra språk använder match för det som Python alltid har kallat search. Därmed lyckas `re.prefixmatch(r"\d{4}", "22000")` på de fyra första siffrorna, och `re.search(r"\d{4}", "DK-2200")` hittar postnumret inne i strängen. Till och med `re.search(r"^\d{4}$", "2200\n")` lyckas, eftersom `$` också matchar precis före en radbrytning i slutet. `re.fullmatch`, som har funnits sedan Python 3.4, avvisar alla tre. Använd `fullmatch` när ett mönster beskriver hela värdet, eller förankra med `\A` och `\Z`, som Python 3.14 också stavar `\z`.

En tillåtelselista med en lös punkt. En kontroll som bara släpper igenom omdirigeringar till företagets egen webbplats kan skrivas som `^https://www.example.com`. Den godtar `https://www.example.com.evil.example/`, eftersom mönstret tar slut innan värdnamnet gör det, och varje punkt utan escape matchar vilket tecken som helst, så en värd som skiljer sig från den riktiga med ett enda tecken i stället för en punkt släpps också igenom. Att escapa punkterna, som `re.escape("www.example.com")` gör, och avsluta mönstret med `/` täpper till båda hålen. Ännu bättre: parsa URL:en och jämför värdnamnet med det förväntade som en hel sträng.

Ett mönster som flyttas mellan motorer. Ett mönster som har testats med `grep -E` och klistrats in i Python kan välja en annan träff, som exemplet med `a|ab` visar, och ett som fungerar i ett BRE-verktyg med `\+` kan betyda ett bokstavligt plustecken i ett annat. Samma `\d` som godtar arabisk-indiska siffror i Python avvisar dem i JavaScript. Testa varje mönster i den motor som ska köra det, med de flaggor det ska köras med.

En global flagga som minns. I JavaScript sparar ett reguljärt uttryck med flaggan `g` eller `y` positionen där den senaste träffen slutade i `lastIndex` och börjar nästa sökning där; efter ett misslyckat försök nollställs `lastIndex` till 0. Ett mönster som ligger i en konstant och återanvänds för validering svarar då olika varannan gång:

```
> const re = /^\d{4}$/g;
undefined
> re.test("2200")
true
> re.test("2200")
```

```
false
> re.test("2200")
true
```

Utelämna flaggan `g` på mönster som används med `test()`, eller sätt `lastIndex` till 0 före varje anrop.

Text som ser likadan ut men inte är det. Namnet André kan lagras med é som ett sammansatt tecken eller med ett vanligt e följt av en kombinerande accent. De två ser identiska ut på skärmen, men POSIX slår fast att matchning följer det bitmönster som används för att koda ett tecken, inte den grafiska återgivningen, och `re.search("André", "Andre\u0301")` i Python returnerar `None`. Efter `unicodedata.normalize("NFC", ...)` på texten hittar anropet namnet. Normalisera mönstret och texten till samma form före matchning, med `unicodedata.normalize` i Python eller `String.prototype.normalize` i JavaScript.

Läs mer

- The Open Group Base Specifications Issue 8: Regular Expressions — POSIX-definitionen: BRE och ERE, uttryck inom hakparenteser, teckenklasser och locales, samt regeln om leftmost-longest.
- regexp package (Go) — garantin om linjär tid, leftmost-first mot leftmost-longest, och varför CompilePOSIX avviker från POSIX för delträffar.
- re — Regular expression operations (Python) — syntaxen, Unicode- och ASCII-klasser, backtracking och possessiva kvantifierare, samt search, prefixmatch och fullmatch.
- ECMAScript: Text Processing (living draft, ECMAScript 2027) — reguljära uttryck i JavaScript: arvet från Perl 5, definitionen av `\d`, Unicode-egenskapsescapes och hur lastIndex används.
- CWE-1333: Inefficient Regular Expression Complexity — catastrophic backtracking och ReDoS: villkoren som gör ett mönster utnyttjbart, observerade exempel och åtgärder.

Ordlista

Välj utifrån era faktiska förutsättningar

Beskriv arbetsflödet, befintliga gränssnitt och användarna som behöver förändringen. Vi kan definiera en testbar leverans och en överlämning som gör att teamet kan underhålla lösningen.

Se vad det kostar Hör av er →

Gör begreppen praktiska

Fråga hur ”Vad är ett reguljärt uttryck?” gäller i er organisation. Ett konkret exempel från ert nuvarande system räcker för att börja samtalet.

Se våra tjänster

welcome@engineer.company +45 50 10 61 84 Köpenhamn, Danmark

Ta med dig den här sidan

Visitkort

Omsorgsfullt skapad med kärlek av D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Upphovsrätt © 2025 – nu · Engineer ApS · Alla rättigheter förbehållna.

Utskrivet från https://engineer.company/sv/glossary/regular-expression/