Engineer

Regulært uttrykk

et mønster som beskriver hvilke tekststrenger som gir treff

Hva det er

Et regulært uttrykk er et kort mønster som beskriver en mengde tekststrenger, slik at et program kan plukke ut dem som hører til. POSIX-standarden, som The Open Group og IEEE gir ut, åpner kapitlet sitt om emnet med å beskrive regulære uttrykk som en måte å velge ut bestemte strenger fra en mengde tegnstrenger på. I dagligtale og i navnene på mange funksjoner og biblioteker krymper begrepet til regex, eller til regexp, som er skrivemåten i JavaScripts RegExp-objekt og i regexp-pakken i Go.

De fleste tegn i et mønster står for seg selv: en `a` i mønsteret matcher en `a` i teksten. Noen få er spesielle. Et punktum matcher et hvilket som helst tegn, `*` gjentar det som står foran, null eller flere ganger, et uttrykk i hakeparenteser som `[0-9]` matcher ett tegn fra en mengde, `^` og `$` binder mønsteret til starten og slutten, og et intervall som `{4}` krever et bestemt antall gjentakelser. Dermed beskriver `^[0-9]{4}$` enhver streng som består av nøyaktig fire sifre, uten andre tegn før eller etter. Standardens eget eksempel på et intervall er `c{3}` brukt på `abababccccccd` : det matcher det sjuende til niende tegnet, de tre første av de seks c-ene.

Det finnes mer enn én dialekt. POSIX definerer selv to: grunnleggende regulære uttrykk (BRE), som de fleste av verktøyene i standarden bruker, og utvidede regulære uttrykk (ERE), som noen bruker i stedet. C-funksjonene `regcomp()` og `regexec()` tilbyr begge. Forskjellen er ikke kosmetisk. I et ERE betyr `a+` én eller flere `a` . I et BRE er `+` et vanlig tegn, og standarden overlater til hver implementasjon om `\+` betyr én eller flere eller et bokstavelig plusstegn.

Den andre store familien stammer fra Perl. ECMAScript-spesifikasjonen slår fast at formen og funksjonaliteten til regulære uttrykk i JavaScript er modellert etter funksjonaliteten for regulære uttrykk i Perl 5. Regexp-pakken i Go beskriver

syntaksen sin som den generelle syntaksen som Perl, Python og andre språk bruker, og `re`-modulen i Python er et tredje medlem. Disse motorene deler det meste av notasjonen, men ikke hver detalj. Forkortelsesklassene er et godt eksempel: `\d` (et siffer) og `\w` (et ordtegn) ser like ut overalt, men hva de dekker, avhenger av motoren og flaggene, som avsnittet om praksis nedenfor viser.

Bruk begrepet i en beslutning

Bruk «Hva er et regulært uttrykk?» i en konkret beslutning. Vi kan knytte begrepet til systemet dere bygger eller kjøper. Ett team som tar en idé fra skisse til noe kundene dine kan logge inn på, i stedet for at den sendes videre mellom tre leverandører.

Se tjenestene våre Ta kontakt →

Undersøk antakelsene bak ordet

Bruk forklaringen og det tilknyttede arbeidet til å formulere spørsmål for prosjektet. Vi kan undersøke antakelsene sammen før et verktøy velges. Ledet en GIS-kartapp til 10 ganger mer omsetning — 10x inntektsvekst, ledet fra kartproduktet

Ledet en GIS-kartapp til 10 ganger mer omsetning Ta kontakt →

Hvorfor det er viktig

Regulære uttrykk sitter i mye programvare som ikke ser ut som behandling av tekst: validering av skjemafelt, filtre som henter verdier ut av logglinjer, rutingregler i webservere, søk og erstatt i tekstredigeringsprogrammer. Hvert mønster er kort, og derfor er det lett å overse hva det faktisk godtar. To spørsmål avgjør resultatet, og svaret på begge avhenger av motoren: hvilket treff som velges når flere er mulige, og hvor lang tid det kan ta å finne ett.

POSIX velger treffet som står lengst til venstre, og blant dem det lengste: av alle treffene som begynner tidligst i teksten, vinner det lengste, på engelsk leftmost-longest. Standardens eksempler er BRE-et `bb*`, som matcher andre til fjerde tegn i `abbbc`, og ERE-et `(wee|week)(knights|night)`, som matcher alle ti tegnene i `weeknights`. Perl-familien jobber annerledes. Den prøver alternativene i den rekkefølgen de er skrevet, og tar det første som fører til et treff, det dokumentasjonen til Go kaller leftmost-first. Det samme mønsteret på den samme teksten kan derfor gi to svar:

```
$ echo ab | grep -oE 'a|ab'
ab
>>> re.search("a|ab", "ab").group()
'a'
> "ab".match(/a|ab/)[0]
'a'
```

Go tilbyr begge deler. Funksjonen `Compile` bruker leftmost-first, mens `CompilePOSIX` begrenser syntaksen til POSIX ERE og bytter til leftmost-longest. Selv da beholder pakken sin egen regel for hvilke deltreff som velges, og dokumentasjonen forklarer hvorfor: POSIX-regelen er ifølge den uoverkommelig dyr å beregne og ikke engang veldefinert.

Det andre spørsmålet er tid. `re` i Python backtracker, som de fleste motorer i Perl-familien. Når en del av mønsteret ikke kommer videre, går motoren et steg tilbake og prøver en annen måte å matche det foregående på. Dokumentasjonen til Python viser ideen med `a*a` mot `aaaa` : `a*` tar først alle fire bokstavene og gir så én tilbake, slik at den siste `a` kan matche. For de fleste mønstre er kostnaden usynlig. For noen vokser antallet måter å prøve på eksponentielt med lengden på inndataene, og en streng på noen dusin tegn kan holde en prosessor opptatt i minutter. MITRE katalogiserer dette som CWE-1333, Inefficient Regular Expression Complexity, og oppgir ReDoS (regular expression denial of service) og catastrophic backtracking som andre navn på det. Regexp-pakken i Go ble utformet rundt det motsatte løftet. Dokumentasjonen slår fast at matchingen garantert kjører i tid som er lineær i størrelsen på inndataene, og legger til at de fleste implementasjoner med åpen kildekode ikke gir noen slik garanti.

Slik ser det ut i praksis

Ta et registreringsskjema for kunder i Danmark og Ukraina, og et fornavnfelt som bare skal godta bokstaver. Det opplagte mønsteret er `[A-Za-z]+` , og i Python 3.15 oppfører det seg slik:

```
>>> 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")
>>>
```

Et kall som returnerer `None` , skriver ikke ut en eneste linje i det interaktive skallet, så ASCII-klassen avviser Søren og Олена (Olena), to helt vanlige navn i København og Kyiv. Bytter man til `\w` , slipper de gjennom, men i Python dekker

`\w` alle tegn som `str.isalnum()` er sann for, pluss understreken, så klassen slipper også inn `Søren_42`. Klassen `[^\W\d_]` utelukker tegn som ikke er ordtegn, desimalsifre og understreken. Den slipper også gjennom enkelte talltegn som ikke er desimalsifre, så den er ingen streng test for bare bokstaver. For navn på mer enn ett ord godtar `[^\W\d_]+( [^\W\d_]+)*` både `Anne Sofie Østergaard` og `Олена Коваль`. Mønsteret avviser fortsatt en `Anne-Marie` med bindestrek, og det er en grunn til å holde et navnefelt så romslig som virksomheten tåler. POSIX-verktøy har det samme problemet i en annen form: standarden lar betydningen av et tegnområde som `A-Z` være uspesifisert utenfor POSIX-localet og tilbyr klassen `[[:alpha:]]`, der det gjeldende localet bestemmer hvilke tegn som hører med.

Å få dette riktig betyr mer enn ett skjema. En av tjenestene på dette nettstedet, Internasjonalisering og lokalisering, beskriver målet som programvare som leses naturlig i alle markeder kunden selger i, helt ned til måten navn sorteres på. Et navnefelt som avviser Søren, kommer til kort lenge før noen sortering finner sted.

Sifre reiser det motsatte spørsmålet. Et dansk postnummer består av fire sifre, så `\d{4}` ser ut som en trygg kontroll av `2200`. I Python matcher `\d` i et tekstmønster likevel ethvert desimalsiffer i Unicode, ikke bare 0 til 9, og godtar det samme postnummeret skrevet med arabisk-indiske sifre:

```
>>> 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 den andre siden. ECMAScript-spesifikasjonen definerer `\d` som nøyaktig de ti tegnene 0 til 9, så `/^\d{4}$/.test(s)` returnerer `false` i Node.js for den samme strengen, mens `/^\p{Nd}{4}$/u` returnerer `true`. Et skjema som kontrolleres i nettleseren med den ene regelen og på serveren med den andre, vil være uenig med seg selv om de samme inndataene. `int()` i Python leser strengen som 2200, men en databasekolonne, en oppslagstabell eller en annen tjeneste gjør kanskje ikke det. Bestem hvilke sifre som menes: `[0-9]` eller flagget `re.ASCII` når bare ASCII-sifre holder, og `\p{Nd}` med flagget `u` i JavaScript når ethvert desimalsiffer er velkomment.

Hvor det går galt

Et mønster som dobler tiden for hver bokstav. Et navnemønster som skal tillate ord skilt med enkle mellomrom, skrives lett som `([^\W\d_]+ ?)+` . På et gyldig navn er det raskt. På en rekke bokstaver fulgt av et tegn det ikke kan godta, prøver motoren alle måter å dele bokstavene inn i ord på før den gir opp, og antallet måter dobles for hver bokstav. På én maskin med Python 3.15 ga den raskeste av fem kjøringer for hver lengde disse tidene for `fullmatch` på en streng av bokstaver fulgt av `!` :

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

Hver bokstav dobler tiden, så to bokstaver til firedobler den. Regner man videre med den faktoren, ville tretti bokstaver ta omtrent ett minutt og trettito mer enn fire minutter. Dette er tilfellet CWE-1333 beskriver: backtracking-forsøkene vokser eksponentielt med inndataene, inndataene trenger ikke å matche, og de kan være lange nok. Omskrivingen `[^\W\d_]+( [^\W\d_]+)*` beskriver de samme navnene uten en nøstet kvantifikator, avviser strengen på 24 bokstaver på omtrent ett mikrosekund og en million bokstaver fulgt av `!` på omtrent tretti millisekunder. Fjern nøstede kvantifikatorer, begrens lengden på inndataene før de matches, og kjør mønstre som ser inndata man ikke kan stole på, på en motor med garantert lineær tid, som den i Go.

Søk der hele verdien skal matche. I Python godtar `re.search` et treff hvor som helst i strengen, og `re.match` forankrer bare i starten; Python 3.15 gir den det tydeligere navnet `prefixmatch` og merker det gamle navnet som soft deprecated, med en merknad om at andre språk bruker match om det Python alltid har kalt search. Dermed lykkes `re.prefixmatch(r"\d{4}", "22000")` på de fire første sifrene, og `re.search(r"\d{4}", "DK-2200")` finner postnummeret inne i strengen. Selv `re.search(r"^\d{4}$", "2200\n")` lykkes, fordi `$` også matcher rett før et linjeskift på slutten. `re.fullmatch` , som har vært med siden Python 3.4, avviser alle tre. Bruk `fullmatch` når et mønster beskriver hele verdien, eller forankre med `\A` og `\Z` , som Python 3.14 også staver `\z` .

En tillatelsesliste med et løst punktum. En kontroll som bare slipper omdirigeringer til selskapets eget nettsted, kan skrives som `^https://www.example.com`. Den godtar `https://www.example.com.evil.example/`, fordi mønsteret slutter før vertsnavnet gjør, og hvert punktum uten escape matcher et hvilket som helst tegn, så en vert som skiller seg fra den ekte med ett enkelt tegn i stedet for et punktum, slipper også gjennom. Å escape punktumene, slik `re.escape("www.example.com")` gjør, og avslutte mønsteret med `/` lukker begge hullene. Enda bedre: parse URL-en, og sammenlign vertsnavnet med det forventede som en hel streng.

Et mønster som flyttes mellom motorer. Et mønster som er testet med `grep -E` og limt inn i Python, kan velge et annet treff, slik eksemplet med `a|ab` viser, og et som virker i ett BRE-verktøy med `\+`, kan bety et bokstavelig plusstegn i et annet. Den samme `\d` som godtar arabisk-indiske sifre i Python, avviser dem i JavaScript. Test hvert mønster i motoren som skal kjøre det, med flaggene det skal kjøre med.

Et globalt flagg som husker. I JavaScript lagrer et regulært uttrykk med flagget `g` eller `y` posisjonen der det siste treffet sluttet, i `lastIndex`, og starter neste søk der; etter et mislykket forsøk nullstiller det `lastIndex` til 0. Et mønster som ligger i en konstant og gjenbrukes til validering, svarer da ulikt annenhver gang:

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

Utelat flagget `g` på mønstre som brukes med `test()`, eller sett `lastIndex` til 0 før hvert kall.

Tekst som ser lik ut, men ikke er det. Navnet André kan lagres med é som ett samlet tegn eller med en vanlig e fulgt av en kombinerende aksent. De to ser identiske ut på skjermen, men POSIX slår fast at matching følger bitmønsteret som brukes til å kode et tegn, ikke den grafiske fremstillingen, og `re.search("André", "Andre\u0301")` i Python returnerer `None`. Etter

`unicodedata.normalize("NFC", ...)` på teksten finner kallet navnet. Normaliser mønsteret og teksten til samme form før matching, med `unicodedata.normalize` i Python eller `String.prototype.normalize` i JavaScript.

Les mer

- The Open Group Base Specifications Issue 8: Regular Expressions — POSIX-definisjonen: BRE og ERE, uttrykk i hakeparenteser, tegnklasser og locales, og regelen om leftmost-longest.
- regexp package (Go) — garantien om lineær tid, leftmost-first mot leftmost-longest, og hvorfor CompilePOSIX avviker fra POSIX for deltreff.
- re — Regular expression operations (Python) — syntaksen, Unicode- og ASCII-klasser, backtracking og possessive kvantifikatorer, og search, prefixmatch og fullmatch.
- ECMAScript: Text Processing (living draft, ECMAScript 2027) — regulære uttrykk i JavaScript: arven fra Perl 5, definisjonen av `\d`, Unicode-egenskapsescapes og hvordan lastIndex brukes.
- CWE-1333: Inefficient Regular Expression Complexity — catastrophic backtracking og ReDoS: betingelsene som gjør et mønster utnyttbart, observerte eksempler og tiltak.

Ordliste

Velg ut fra deres faktiske rammer

Beskriv arbeidsflyten, eksisterende grensesnitt og brukerne som trenger endringen. Vi kan definere en testbar leveranse og en overlevering som gjør teamet i stand til å vedlikeholde løsningen.

Se hva det koster Ta kontakt →

Gjør begrepene praktiske

Spør hvordan «Hva er et regulært uttrykk?» gjelder i organisasjonen deres. Et konkret eksempel fra dagens system er nok til å starte samtalen.

Se tjenestene våre

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

Ta med deg denne siden

Visittkort

Laget med kjærlighet av D.

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

Opphavsrett © 2025 – nå · Engineer ApS · Alle rettigheter forbeholdt.

Skrevet ut fra https://engineer.company/nb/glossary/regular-expression/