Tabela de conteúdos

Introduction

Olá Mundo

```
_ <- fat.std
console.log('Olá Mundo')
```

Início Rápido

Vá diretamente para a documentação:

- Visão geral
- Sintaxe da linguagem
- Bibliotecas padrão

Executando seu código

Você pode executar o FatScript usando o interpretador `fry` ou o playground na web.

Interpretador Fry

Para execução local, utilize o interpretador `fry`. Para detalhes sobre sua instalação e uso, consulte a seção de configuração.

Playground na Web (beta)

Para testes rápidos e convenientes, execute seu código diretamente no FatScript Playground. O playground oferece um REPL e uma interface intuitiva que permite carregar scripts a partir de um arquivo.

Download de PDF

- FatScript v2.6.0 (atual)
- FatScript v1.3.5 (legado)

Tutoriais

Mergulhe em nossos tutoriais imersivos, insights por trás dos bastidores e tópicos relacionados no canal do YouTube FatScript.

Doações

Você achou o FatScript útil e gostaria de agradecer?

Compre-me um café

Licença

fatscript.org

Published on Mon Jun 10 2024 00:35:04 GMT+0100 (British Summer Time)

Visão geral

FatScript é uma linguagem de programação leve e interpretada projetada para criar aplicativos baseados em console. Ela enfatiza a simplicidade, facilidade de uso e conceitos de programação funcional.

Livre e de código aberto

`fatscript/fry` é um projeto de código aberto que incentiva a colaboração e o compartilhamento de conhecimento. Nós convidamos os desenvolvedores a contribuir para o projeto e nos ajudar a melhorá-lo com o tempo.

Conceitos chave

- Gerenciamento automático de memória por coleta de lixo (GC)
- Combinações simbólicas de caracteres para uma sintaxe minimalista
- REPL (Read-Eval-Print Loop) para testes rápidos de expressões
- Suporte para sistema de tipos, herança e subtipagem por meio de aliases
- Suporte para programação imutável e métodos passáveis (como valores)
- Manter se simples e intuitivo, sempre que possível

Conteúdo desta seção

- Configuração: como instalar o interpretador de FatScript
- Opções: como personalizar a execução
- Empacotamento: como empacotar um aplicativo FatScript
- Ferramentas: visão geral de algumas ferramentas e recursos extras

Limitações e desafios

Embora o FatScript seja projetado para ser simples e intuitivo, ele ainda é uma linguagem relativamente nova e pode não ser adequado para todos os casos de uso. Por exemplo, pode ter desempenho inferior em comparação com linguagens de programação mais maduras ao lidar com cargas de trabalho complexas ou tarefas de computação de alto desempenho.

Configuração

Para começar a "fritar" seu código "gordo", você precisará de um interpretador para a linguagem de programação FatScript.

fry, O Interpretador FatScript

fry é um interpretador e ambiente de execução gratuito para FatScript. Você pode instalá-lo em sua máquina seguindo as instruções a seguir.

Instruções

`fry` é projetado para GNU/Linux, mas também pode funcionar em outros sistemas operacionais.

Para distribuições baseadas em Arch, instale através do pacote AUR fatscript-fry.

Para outras distribuições, experimente o script de instalação automática:

```
curl -sSL https://gitlab.com/fatscript/fry/raw/main/get_fry.sh -o get_fry.sh;
bash get_fry.sh || sudo bash get_fry.sh
```

Ou, para instalar `fry` manualmente:

- Clone o repositório:

```
git clone --recursive https://gitlab.com/fatscript/fry.git
```

- Depois, execute o script de instalação:

```
cd fry
./install.sh
```

a instalação manual pode copiar o binário `fry` para a pasta $HOME/.local/bin, alternativamente, use `sudo` para instalá-lo em /usr/local/bin/

- Verifique se o `fry` foi instalado, executando:

```
fry --version
```

Dependências

Se a instalação falhar, podem estar faltando algumas dependências. `fry` requer `git`, `gcc` e `libcurl` para compilar. Por exemplo, para instalar essas dependências no Debian/Ubuntu, execute:

```
apt update
apt install git gcc libcurl4-openssl-dev
```

Back-end para entrada de texto

`linenoise` é uma dependência leve e uma alternativa ao `readline`, mantida como um submódulo. Se foi não foi incluída durante a operação inicial de `git clone`, você pode corrigir isso com os seguintes comandos:

```
git submodule init
git submodule update
```

Se você preferir "linkar" com o `readline`, apenas certifique-se de que ele esteja instalado, executando:

```
apt install libreadline-dev
```

Suporte de Sistema Operacional

`fry` é primordialmente projetado para GNU/Linux, mas também é acessível em outros sistemas operacionais:

Android

Se você estiver no Android, pode instalar o `fry` via Termux. Basta instalar as dependências necessárias da seguinte maneira:

```
pkg install git clang
```

Em seguida, você pode seguir as instruções padrão de instalação do `fry`.

ChromeOS

Se você estiver usando o ChromeOS, pode habilitar o suporte ao Linux seguindo as instruções aqui.

MacOS

Se você estiver usando o MacOS, precisará ter as Command Line Tools instaladas.

iOS

Se você estiver usando o iOS, poderá usar o `fry` via iSH. Primeiro, instale as dependências necessárias:

```
apk add bash gcc libc-dev curl-dev
```

Em seguida, de acordo com esta discussão, configure o `git` para funcionar corretamente, assim:

```
wget https://dl-cdn.alpinelinux.org/alpine/v3.11/main/x86/git-2.24.4-r0.apk
apk add ./git-2.24.4-r0.apk
git config --global pack.threads "1"
```

Windows

Se você estiver usando o Windows, poderá usar o `fry` via Windows Subsystem for Linux (WSL).

Imagem Docker

`fry` também está disponível como uma imagem docker:

```
docker run --rm -it fatscript/fry
```

Para executar um arquivo FatScript com o docker, use o seguinte comando:

```
docker run --rm -it -v ~/project:/app fatscript/fry prog.fat
```

substitua ~/project pelo caminho para o seu arquivo FatScript

Solução de problemas

Se você encontrar qualquer problema ou bug ao usar o `fry`, por favor abra uma "issue".

Opções

Com esta descrição dos modos e parâmetros disponíveis, você descobrirá que o `fry` tem várias especiarias guardadas na manga para você temperar a execução do seu código.

Argumentos de linha de comando

A interface CLI oferece alguns modos de operação:

- `fry [OPÇÕES]` read-eval-print-loop (REPL)
- `fry [OPÇÕES] ARQUIVO [ARGS]` executar um arquivo FatScript
- `fry [OPÇÕES] -b/-o ENTRADA SAÍDA` criar um bundle
- `fry [OPÇÕES] -f ARQUIVO...` formatar arquivos de código-fonte do FatScript

Aqui estão os parâmetros de opção disponíveis:

- `-a, --ast` exibir apenas a árvore sintática abstrata
- `-b, --bundle` salvar bundle em arquivo de saída (implica em -p)
- `-c, --clock` registro de tempo e estatísticas (alternar)
- `-d, --debug` habilitar logs de debug (implica em -c)
- `-e, --error` continuar em caso de erro (alternar)
- `-f, --format` indentar arquivos de código-fonte do FatScript
- `-h, --help` exibir ajuda e sair
- `-i, --interactive` habilitar REPL com execução do arquivo
- `-j, --jail` restringir sistema de arquivos, rede e chamadas de sistema
- `-k, --stack` # definir a profundidade da pilha (contagem de frames)
- `-m, --meta` exibir informações sobre a build
- `-n, --nodes` # definir limite de memória (contagem de nós)
- `-o, --obfuscate` ofuscar o bundle (implica em -b)
- `-p, --probe` realizar análise estática (teste seco)
- `-s, --save` armazenar sessão do REPL em repl.fat
- `-v, --version` exibir número da versão e sair
- `-w, --warranty` exibir isenção de responsabilidade e sair
- `-z, --minify` minificar código-fonte (implica em -p)

a opção `-e` é ativada automaticamente nos modos REPL e probe

combinar `-p` com `-f` envia o resultado formatado para stdout

Gerenciamento de memória

`fry` gerencia a memória automaticamente sem pré-reserva. Você pode limitar o uso da memória especificando o número de nós com as opções da CLI:

- `-n <count>` para uma contagem exata de nós
- `-n <count>k` para kilonós, contagem * 1000
- `-n <count>m` para meganós, contagem * 1000000

Por exemplo, `fry -n 5k meuPrograma.fat` restringe o aplicativo a 5000 nós.

O coletor de lixo (GC) é executado automaticamente quando restam 256 nós antes que o limite final de memória seja atingido (premonição do GC). Você também pode invocar o GC a qualquer momento chamando o método `runGC` da biblioteca SDK desde a thread principal.

Estimativa de bytes (x64)

Cada nó em uma plataforma de 64 bits usa aproximadamente ~200 bytes. O tamanho real do nó depende dos dados que ele contém. Por exemplo, o limite padrão é 10 milhões de nós, seu programa pode chegar a usar cerca de 2 GB de RAM ao atingir o limite padrão.

Use a opção `-c` ou `--clock` para imprimir as estatísticas de execução e ter uma melhor compreensão de como seu programa está se comportando na prática.

Verificação de tempo de execução

Existem dois comandos embutidos para verificar o uso de memória em tempo de execução:

- `$nodesUsage` nós alocados no momento (O(1))
- `$bytesUsage` bytes alocados no momento (O(n))

verificar os bytes alocados no momento é uma operação cara, pois precisa percorrer todos os nós para verificar o tamanho real de cada um

Tamanho da pilha

A profundidade máxima da pilha é definida em `parameters.h`, no entanto, você pode personalizar o tamanho da pilha até certo ponto usando opções da linha de comando (CLI):

- `-k <count>` para um número exato de frames
- `-k <count>k` para kibiframes, count * 1024

Arquivo "run commands"

Na inicialização, `fry` procura um arquivo `.fryrc` no mesmo caminho do arquivo do programa e, se não encontrado, também no diretório de atual. Se encontrado, é executado como uma fase de "pré-cozimento" para configurar o ambiente para a execução do programa.

Gerenciamento de memória com .fryrc

Você pode usar o arquivo `.fryrc` para definir o limite de memória para seu projeto sem precisar especificá-lo como argumento da CLI. Para fazer isso, você pode usar o método `setMem` fornecido pela biblioteca SDK, assim:

```
_ <- fat.system
setMem(64000)  # define 64k nós como limite de memória
```

Detalhes de inicialização

As opções da linha de comando são aplicadas primeiro, exceto pelo limite de memória. Durante a fase de pré-cozimento, o `fry` usa o limite padrão de 10 milhões de nós, independentemente da opção da linha de comando. Se você definir um limite de memória no arquivo `.fryrc`, esse limite terá efeito a partir desse ponto e substituirá a opção da linha de comando para toda a execução. Se o arquivo `.fryrc` não definir um limite de memória, a opção da linha de comando terá efeito após a fase de pré-cozimento.

O escopo de pré-cozimento é invisível por padrão. Após a execução do arquivo `.fryrc`, um escopo zerado é fornecido para o seu programa, o que permite testar seu código com um limite muito baixo de nós ao usar um arquivo `.fryrc` sem afetar a contagem de nós. Isso também impede que o namespace `.fryrc` entre em conflito com o escopo global do seu programa. No entanto, se você quiser manter as entradas declaradas no `.fryrc` no escopo global para fins de configuração, pode chamar o comando embutido `$keepDotFry` em algum lugar do arquivo `.fryrc`.

Outro uso possível, além de configurar o limite de memória, é pré-carregar as importações comuns, por exemplo, os tipos padrão:

```
$keepDotFry
_ <- fat.type._
```

Modo Sandbox

Use a opção `-j` ou `--jail` para inibir os seguintes comandos embutidos:

- `write`, `remove` e `mkDir` - Esses comandos modificam o sistema de arquivos.
- `request` - Este comando é usado para fazer requisições HTTP externas.
- `shell`, `capture`, `fork` e `kill` - Estes comandos estão envolvidos em iniciar ou parar processos arbitrários.

Veja também

- Comandos embutidos
- Biblioteca SDK

Empacotamento

O fry oferece uma ferramenta integrada de empacotamento para código FatScript.

Utilização

Para agrupar seu projeto em um único arquivo a partir do ponto de entrada, execute:

```
fry -b sweet mySweetProject.fat
```

Este processo consolida todas as importações (exceto caminhos literais) e remove espaços desnecessários, melhorando os tempos de carregamento:

- Adiciona um shebang ao código empacotado
- Recebe o atributo de execução para o modo de arquivo

A seguir, você pode executar seu programa:

```
./sweet
```

> o empacotamento substituirá quaisquer instruções `$break` (ponto de interrupção do depurador) por `()`

Ofuscação

Para uma ofuscação opcional, use `-o`:

```
fry -o sweet mySweetProject.fat  # cria o pacote ofuscado
./sweet                          # executa seu programa da mesma maneira
```

> Ao distribuir por meio de hosts públicos, considere definir uma chave personalizada com um `.fryrc` local. Apenas o cliente deve ter acesso a esta chave para proteger o fonte.

A ofuscação usa o algoritmo enigma para encriptação, garantindo uma decodificação rápida. Para um tempo de carregamento ótimo, prefira `-b` se a ofuscação não for essencial.

Considerações

As importações são deduplicadas e incluídas com base na ordem de sua primeira aparição. Como resultado, a sequência em que você importa seus arquivos desempenha um papel crítico no resultado final agrupado. Embora essas considerações geralmente sejam inconsequentes para projetos pequenos, o empacotamento de projetos maiores pode exigir uma organização adicional. Sempre valide o seu código empacotado.

Ferramentas

Aqui estão algumas dicas que podem melhorar sua experiência de programação com FatScript.

Análise estática

Use o modo de verificação para checar a sintaxe e receber dicas sobre o seu código:

```
fry -p mySweetProgram.fat
```

Depurador

Um ponto de interrupção, indicado pelo comando `$break`, atua como uma ferramenta de depuração ao interromper temporariamente a execução do programa em um local designado e carregar o console de depuração integrado. Ele oferece um ambiente interativo para examinar o estado atual do programa, inspecionando valores no escopo, avaliando expressões e rastreando o fluxo do programa.

Para ativar os pontos de interrupção, é necessário executar o programa com modo interativo habilitado:

```
fry -i mySweetProgram.fat
```

No FatScript, `$break` retorna `null`, o que pode alterar um valor de retorno se colocado no final de um bloco, devido ao recurso de auto-retorno. Tenha cuidado com o posicionamento de `$break` para evitar efeitos não intencionais na funcionalidade do programa.

Formatação do código-fonte

Suporte nativo

Você pode aplicar a indentação automática ao seu código fonte usando o seguinte comando:

```
fry -f mySweetProgram.fat
```

Extensão do Visual Studio Code

Para adicionar suporte de formatação de código ao VS Code, você pode instalar a extensão fatscript-formatter. Abra o Quick Open do VS Code (Ctrl+P), cole o seguinte comando e pressione enter:

```
ext install aprates.fatscript-formatter
```

- o `fry` precisa estar instalado em seu sistema para que essa extensão funcione

Realce de sintaxe

Extensão do Visual Studio Code

Para adicionar destaque de sintaxe do FatScript ao VS Code, você pode instalar a extensão fatscript-syntax. Abra o Quick Open do VS Code (Ctrl+P), cole o seguinte comando e pressione enter:

```
ext install aprates.fatscript-syntax
```

Você também pode encontrar e instalar essas extensões no Marketplace de Extensões do VS Code.

Plugin para Vim e Neovim

Para instalar o realce de sintaxe do FatScript para Vim e Neovim, confira o plugin vim-syntax.

Para usuários do Neovim, adicione a linha respectiva à sua configuração:

Usando packer.nvim:

```
use { 'https://gitlab.com/fatscript/vim-syntax', as = 'fatscript' }
```

Usando lazy.nvim:

```
{ 'https://gitlab.com/fatscript/vim-syntax', name = 'fatscript' }
```

Arquivo de sintaxe do Nano

Para instalar o realce de sintaxe do FatScript no `nano`, siga estes passos:

1. Baixe o arquivo `fat.nanorc` daqui.
2. Copie o arquivo `fat.nanorc` para o diretório de sistema do `nano`:

```
sudo cp fat.nanorc /usr/share/nano/
```

Se o realce de sintaxe não for habilitado automaticamente, talvez você precise habilitá-lo explicitamente em seu arquivo `.nanorc`. Consulte as instruções na Wiki do Arch Linux para mais informações.

Após a instalação do destaque de sintaxe, você também pode usar o formatador de código no `nano` com a seguinte sequência de atalhos:

- Ctrl+T Executar; e em seguida...
- Ctrl+O Formatador

Outras dicas

Navegação de arquivos no console

Para navegar pelas pastas do seu projeto a partir do terminal, você pode experimentar usar um gerenciador de arquivos do console como o ranger, combinado com o `nano`, `vim` ou `nvim`. Defina-o como o editor padrão para o `ranger` adicionando a seguinte linha ao seu arquivo `~/.bashrc`:

```
export EDITOR="nano"
```

Sintaxe

Nas seguintes páginas, você encontrará informações sobre os aspectos centrais da escrita de código FatScript, utilizando tanto os recursos básicos da linguagem quanto os recursos avançados do sistema de tipos e bibliotecas padrão.

Tópicos abordados

- Formatação: como formatar corretamente o código FatScript
- Importações: como importar bibliotecas para o seu código
- Entradas: entendendo o conceito de entradas e escopos
- Tipos: um guia para o sistema de tipos FatScript
- Controle de fluxo: controlando a execução do programa com condicionais
- Loops: utilizando intervalos, map-over e while loops

Formatação

No FatScript, espaços em branco e indentação são irrelevantes, porém são muito bem-vindos para tornar o código mais legível e fácil de entender.

Espaços em branco

- Um caractere de nova linha (`\n`) indica o final de uma expressão, exceto quando:
  - o último token na linha é um operador
  - o primeiro token da próxima linha é um operador não-unário
  - usando parênteses para agrupar expressões
- Expressões podem estar na mesma linha se separadas por vírgula (`,`) ou ponto-e-vírgula (`;`)

Comentários

Comentários começam com # e são terminados por uma nova linha:

```
a = 5  # este é um comentário
```

Nota

FatScript não suporta comentários multi-linhas no momento. Além disso, literais de texto podem acabar como um valor de retorno válido se deixados como a última linha restante, devido à funcionalidade de auto-retorno. Portanto, é recomendável se ater ao formato de comentário de linha única.

Veja também

- Autoformatador de código fonte

Importações

Vamos desvendar a arte de importar arquivos e bibliotecas no FatScript! Por quê? Bem, porque nesta linguagem você pode importar sempre que seu coração desejar, simplesmente usando uma seta para a esquerda `<-`.

Sintaxe de ponto

Para usar importações com sintaxe de ponto, os nomes dos arquivos e pastas do projeto não devem começar com dígito nem conter símbolos.

> você pode forçar qualquer caminho que desejar usando caminhos literais

Importação nomeada

Para importar arquivos, use a extensão `.fat` para nomes de arquivo (ou nenhuma extensão), mas omita a extensão na declaração de importação. Aqui está um exemplo:

```
ref <- nomeDoArquivo
```

> se ambos os arquivos `x` e `x.fat` existirem, o último terá precedência

Para importar arquivos de pastas:

```
ref1 <- pasta.nomeDoArquivo
ref2 <- pasta.subPasta.nomeDoArquivo
```

Para importar todos os arquivos de uma pasta, use a sintaxe de "ponto-sublinhado":

```
lib <- pasta._
```

Observe: apenas os arquivos imediatamente dentro da pasta são incluídos usando a sintaxe acima. Para incluir arquivos de subpastas, mencione-os explicitamente. Além disso, um arquivo "_.fat" (ou arquivo "_") dentro de uma pasta pode substituir o comportamento de importação de "ponto-sublinhado".

Acesso de elementos

Uma vez importado, acesse os elementos usando a sintaxe de ponto:

```
ref1.elemento1
```

Extração de elementos

Para extrair elementos específicos de uma importação nomeada ou para evitar adicionar o nome do módulo todas as vezes (por exemplo, `lib.foo`), use atribuição por desestruturação:

```
{ foo, bar } = lib
```

Visibilidade

Importações nomeadas são resolvidas no escopo global, independentemente de onde forem declaradas. Isso significa que mesmo se você declarar uma importação nomeada dentro de uma função ou escopo local, ela será globalmente acessível.

Importação local

Para importar no escopo atual, use:

```
_ <- nomeDoArquivo
```

Importações locais, ao contrário das nomeadas, despejam o conteúdo do arquivo diretamente no escopo atual. Assim, um método importado pode ser invocado como `baz(arg)` em vez de `ref.baz(arg)`.

Embora as importações locais sejam bem apropriadas para importar tipos no escopo global, elas devem ser usadas com cautela ao importar conteúdos de biblioteca. O uso excessivo de importações locais pode levar a poluição do namespace, tornando mais desafiador seguir o código, porque fica menos aparente de onde vêm os métodos.

Importação local seletiva

Você também pode descartar elementos de uma importação local usando atribuição por desestruturação:

```
{ foo } = { _ <- lib }
```

> o ponte é evitar a poluição do namespace, pois todo o conteúdo será processado

Caminhos literais

Com caminhos literais, você pode usar qualquer nome de arquivo ou extensão. No entanto, observe que essas importações não são avaliadas durante o empacotamento, mas em tempo de execução. Aqui está um exemplo:

```
ref <- '_pasta/fonte-2.outro'
```

Você também pode usar textos inteligentes como caminhos literais:

```
base = 'pasta'
arquivo = 'fonte.xyz'
ref <- '{base}/{arquivo}'
```

Como o FatScript também aceita sintaxe semelhante a JSON, você pode até mesmo carregar um arquivo JSON diretamente como uma importação:

```
json <- 'sample/data.json'
```

> embora possível, é mais aconselhável usar file.read e em seguida recode.fromJSON

Lembre-se de que caminhos literais podem tornar seu código mais complexo e essas importações só podem ser resolvidas dinamicamente, então use-os com moderação.

Política de importação

O FatScript utiliza uma estratégia de "importar apenas uma vez" com um mecanismo de flags no escopo, evitando automaticamente arquivos que já foram importados.

De maneira geral, importações são eficientes em termos de recursos. No entanto, **importações locais dentro de corpos de métodos** devem ser evitados, pois são re-avaliados a cada invocação, podendo causar retenção de memória.

Esse comportamento não é classificado como um bug per se, mas sim uma consequência das escolhas de design no sistema de coleta de lixo (GC) do FatScript. As otimizações do GC excluem nós diretamente derivados do código-fonte, permitindo que eles sejam evitados nos procedimentos padrões de marcação-e-varredura. Como resultado, importações locais dentro de métodos não tem a oportunidade de deduplicação, fazendo com que seus nós permaneçam residentes até o final do programa:

```
meuMetodo = -> {
  _ <- lib  # potencial vazamento de memória
  ...
}
```

Aqui estão algumas estratégias para resolver este problema:

- Mova a declaração de importação para o escopo externo.
- Prefira usar uma importação nomeada como alternativa.
- Reorganize a estrutura de 'lib' para exportar um método.

Entradas

As entradas são pares chave-valor que existem no escopo onde são declaradas.

Nomeação

Os nomes das entradas (chaves) **não** podem começar com uma letra maiúscula, que é a distinção com relação aos tipos. Os identificadores são sensíveis a maiúsculas e minúsculas, portanto "batatasfritas" e "batatasFritas" seriam considerados entradas diferentes.

A convenção recomendada é usar `camelCase` para as entradas.

> você pode usar um nome arbitrário como chave usando nomeação dinâmica

Declaração e atribuição

Em FatScript, você pode declarar entradas simplesmente atribuindo um valor:

```
isOnline: Boolean = true
age: Number       = 25
name: Text        = 'João'
```

Os tipos também podem ser inferidos a partir da atribuição:

```
isOnline = true    # Boolean
age      = 25      # Number
name     = 'João'  # Text
```

Entradas imutáveis

No FatScript, ao declarar uma entrada, ela é por padrão imutável, o que significa que, uma vez atribuído, seu valor não pode ser alterado. Essa imutabilidade garante consistência ao longo da execução do programa:

```
fruta = 'banana'
fruta = 'abacate'  # gera um erro porque 'fruta' é imutável
```

Exceção à Regra

A imutabilidade no FatScript aplica-se à vinculação da entrada, não ao conteúdo de escopos. Embora uma entrada seja imutável, se ela contiver um escopo, o conteúdo desse escopo pode ser modificado, seja pela adição de novas entradas, ou pela alteração de entradas mutáveis contidas no escopo:

```
s = { a = 1, b = 2 }
s.c = 3  # mesmo que 's' seja imutável, ele aceita o novo valor de 'c'
s        # agora é { a = 1, b = 2, c = 3 }
```

Essa escolha de design oferece flexibilidade com modificações de escopo. Em contraste, listas impõem uma imutabilidade mais estrita, impedindo a adição de novas entradas a listas imutáveis.

Note também que escopos são sempre passados por referência. Para modificar o conteúdo de um escopo sem alterar sua referência original, use o método `copy` da Extensão de Protótipo de Escopo para criar uma duplicata.

Entradas mutáveis

Sim, você pode declarar entradas mutáveis, também conhecidas como variáveis. Para declarar uma entrada mutável, use o operador til ~:

```
~ fruta = 'banana'
fruta   = 'abacate'  # ok
```

Observe que mesmo uma entrada mutável não pode mudar imediatamente seu tipo, a menos que seja apagada do escopo. Para apagar uma entrada, atribua `null` a ela e, em seguida, redeclare-a com um novo tipo. Mudar tipos é desencorajado pela sintaxe e não é recomendado, mas é possível:

```
~ color = 32    # cria a entrada color como um número mutável
color = 'blue'  # gera um TypeError porque color é um número
color = null    # a entrada é apagada
color = 'blue'  # redefine color com um tipo diferente (Texto)
```

> você deve declarar a entrada como mutável novamente usando o til ~ ao redefini-la após a exclusão se quiser que o próximo valor seja mutável

Entradas dinâmicas

Você pode criar entradas com nomes dinâmicos usando colchetes `[ ref ]`:

```
ref = 'pipoca'  # texto será o nome da entrada

opcoes = { [ref] = 'é saborosa' }

opcoes.[ref]   # sintaxe dinâmica:    'é saborosa', com acesso de leitura e gravação
opcoes(ref)    # sintaxe de obtenção: 'é saborosa', mas o valor é somente leitura
opcoes.pipoca  # sintaxe de ponto:    'é saborosa', mas deve seguir a nomenclatura
```

> todas as declarações dinâmicas são entradas mutáveis

Essa funcionalidade permite definir dinamicamente os nomes dentro de um escopo e criar entradas com nomes que, de outra forma, não seriam aceitos pelo FatScript.

As entradas dinâmicas também podem usar referências numéricas, mas a referência é convertida em texto automaticamente, e.g.:

```
[ 5 ] = 'texto armazenado na entrada 5'
self.['5']  # devolve 'texto armazenado na entrada 5'
self.[5]    # devolve 'texto armazenado na entrada 5'
```

> em um contexto diferente, não seguido por uma atribuição `=` ou precedido por notação de ponto `.`, a sintaxe dinâmica será interpretada como uma declaração de lista

Entradas especiais (DESCONTINUADO)

Entradas com nomes que começam com o sublinhado _ são completamente livres e dinâmicas, não requerem til ~ e também podem mudar de tipo sem a necessidade de apagamento, como variáveis no JavaScript ou Python.

> Para permitir uma otimização mais agressiva no interpretador, as "entradas especiais" serão descontinuadas a partir da versão `3.x.x`. Em vez disso, serão tratadas como imutáveis, a menos que sejam declaradas com ~, indicando que são mutáveis, mas não poderão mudar de tipo sem a necessidade de apagamento. Ou seja, não haverá um tratamento especial para entradas com nomes que começam com o sublinhado.

Atribuição por desestruturação

Você pode copiar os valores de um escopo em outro escopo assim:

```
_ <- fat.math
distancia = (posicao: Scope/Number): Number -> {
  { x, y } = posicao     # atribuição por desestruturação no escopo do método
  sqrt(x ** 2 + y ** 2)  # calcula a distância entre a origem e (x, y)
}
distancia({ x = 3, y = 5 })  # 5.83095189485
```

A mesma sintaxe funciona de forma semelhante para listas:

```
distancia = (posicao: List/Number): Number -> {
  { x, y } = posicao  # extrai os dois primeiros itens para 'x' e 'y'
  sqrt(x ** 2 + y ** 2)
}
distancia([ 3, 5 ])  # 5.83095189485
```

Você também pode usar a atribuição por desestruturação para expor um determinado método ou propriedade de uma importação nomeada:

```
console <- fat.console
{ log } = console
```

```
log('Olá Mundo')
```

usando essa sintaxe com importações, você pode escolher trazer para o escopo atual apenas os elementos da biblioteca que você está interessado em usar, evitando assim a poluição do namespace com nomes que não teriam uso ou poderiam entrar em conflito com os de sua própria autoria

Sintaxe semelhante a JSON

O FatScript também suporta sintaxe semelhante a JSON para declarar entradas:

```
"nada": null,                  # entrada Void - comportamento distinto, veja abaixo
"estaOnline": true,            # entrada Boolean
"idade": 25,                   # entrada Number
"nome": "João",                # entrada Text
"tags": [ "a", "b" ],          # entrada List
"opcoes": { "prop": "outra" }  # entrada Scope
```

Embora possa parecer que declarar "nada" cria um valor "nada" de `null`, é importante observar que a "entrada resultante" na verdade não existe no escopo. Quando você tenta acessar esse "nada", o FatScript retorna `null`, mas se você tentar mapear o escopo, o nome dessa entrada estará faltando, pois nunca foi realmente criado.

É importante observar que as declarações semelhantes a JSON sempre criam entradas imutáveis, portanto você não pode adicionar o caractere til ~ para torná-las mutáveis.

Tipos

Os tipos são usados no FatScript para combinar dados e comportamentos, atuando como modelos para a criação de novas réplicas.

Nomeação

Os nomes de tipo são sensíveis a maiúsculas e minúsculas, devendo começar com uma letra maiúscula.

A convenção recomendada para identificadores de tipo é `PascalCase`.

Tipos Nativos

O FatScript fornece vários tipos nativos:

- Any - qualquer coisa
- Void - nada
- Boolean - primitivo
- Number - primitivo
- HugeInt - primitivo
- Text - primitivo
- Method - função ou lambda
- List - como uma matriz ou pilha
- Scope - como um objeto ou dicionário
- Error - sim, para erros
- Chunk - dados binários

No entanto, é necessário importar o pacote type para acessar os membros de protótipo de cada tipo.

Tipos Adicionais

Os tipos nativos do FatScript são enriquecidos com uma coleção de tipos extras que expandem as funcionalidades básicas de seus tipos nativos. Criados em FatScript puro, esses tipos adicionais atendem a várias necessidades de programação avançada e facilitam padrões de projeto comuns.

Além disso, você encontrará tipos de domínio específico incorporados nas bibliotecas, como `Worker` na biblioteca async, `FileInfo` em file, `HttpRequest` (entre outros) em http, `CommandResult` em system etc.

Tipos Personalizados

Além de usar os tipos fornecidos pela linguagem ou por uma biblioteca externa, você também pode criar seus próprios tipos ou estender os existentes com novos comportamentos.

Declaração

Para definir um tipo personalizado no FatScript, você pode usar uma simples declaração de atribuição. A definição de tipo pode ser envolvida em parênteses ou chaves. Ambas as sintaxes são válidas e têm o mesmo efeito. Você também pode opcionalmente definir valores padrão para as propriedades do tipo, como mostrado no seguinte exemplo:

```
# Definição de tipo usando parênteses com valors padrão
Carro = (km: Number = 0, cor: Text = 'branco')
```

Unicidade Global

O FatScript possui um único meta-espaço global, exigindo que os nomes dos tipos sejam únicos em todo o seu programa e em quaisquer bibliotecas incluídas. Tentar definir um tipo que compartilha um nome com um tipo existente, mesmo que em um escopo diferente, aciona um `AssignError`. No entanto, se a nova definição for idêntica à existente, ela será simplesmente ignorada.

Para examinar os tipos presentes no meta-espaço global, o comando `_<-fat.std; sdk.getTypes;` se mostra útil. Esta função enumera todos os tipos definidos e detalha seus locais de definição com marcadores `source:line:column`. Este recurso ajuda na navegação e compreensão da estrutura do seu código e suas dependências.

É prudente evitar nomes já utilizados pelos tipos de biblioteca `fat.std` ao definir novos tipos.

Embora o FatScript não imponha um protocolo estrito de nomes para o desenvolvimento de bibliotecas, é recomendado adotar uma estratégia de nomes que evite conflitos. Uma prática comum envolve prefixar os nomes dos tipos com algum identificador único que reflita o nome da sua biblioteca, reduzindo assim a probabilidade de choques de nomes.

Uso

Para criar instâncias de um tipo personalizado, chame o nome do tipo como se fosse um método, opcionalmente passando valores para as propriedades:

```
# Uso do tipo com padrões
carro = Carro()
# saída: { km: Number = 0, cor: Text = 'branco' }

# Uso do tipo, definindo uma das propriedades
carroVermelho = Carro(cor = 'vermelho')
# saída: { km: Number = 0, cor: Text = 'vermelho' }

# Uso do tipo, totalmente qualificado
carroVelho1 = Carro(cor = 'azul', km = 38000)
# substitui ambos os valores

# Uso do tipo, argumentos usando a sequência das propriedades
carroVelho2 = Carro(41000, 'verde')
# substitui valores usando a ordem da definição do tipo
```

Por padrão, os tipos personalizados retornam um escopo de suas propriedades. No entanto, se você definir um método `apply`, o tipo poderá retornar um valor diferente. Por exemplo, aqui está um tipo personalizado `Soma` com um método `apply` que retorna a soma de suas propriedades `a` e `b`:

```
Soma = (a: Number, b: Number, apply = -> a + b)
Soma(1, 2)  # saída: 3
```

observe que os métodos apply têm acesso direto às propriedades da instância

Neste exemplo, o tipo base de saída do `apply` é um número, não um escopo. Isso também significa que as propriedades originais do tipo personalizado são perdidas durante a instanciação e não podem ser acessadas novamente.

Membros do protótipo

Esses são um tipo especial de método, armazenados dentro da definição do tipo:

```
TipoComMembrosDePrototipo = {
  ~ a: Number
  ~ b: Number

  setA = (novoA: Number) -> self.a = novoA
  setB = (novoB: Number) -> self.b = novoB
  soma = (): Number -> self.a + self.b
}
```

Neste exemplo, `setA`, `setB` e `soma` são membros do protótipo. Observe que precisamos usar `self`, que é uma palavra-chave que fornece uma referência ao escopo da própria instância (ou método), para que nós pudéssemos ganhar acesso às propriedades.

Checando tipos

Se você não sabe qual é o tipo de uma entrada, pode simplesmente verificar comparando com um nome de tipo:

```
lugar = 'restaurante'
lugar == Number  # false
lugar == Text    # true
```

alternativamente, use o método `typeOf` da biblioteca SDK para extrair o nome do tipo

Qualquer coisa pode ser comparada com a palavra reservada `Type`, que identifica se se refere a um tipo:

```
Number == Type  # true
```

`Type` também pode ser usado para especificar que um método recebe um parâmetro de tipo:

```
combine = (t: Type, val: Any): Any -> ...
```

Alias de tipo

No FatScript, você pode criar subtipos atribuindo um nome diferente a um tipo existente. Isso significa que o novo tipo herdará todas as propriedades do tipo base. Aqui está um exemplo:

```
_ <- fat.type.Text
Id = Text  # cria um alias
```

Observe que os aliases de tipo são hierárquicos e podem ser usados para classificar valores enquanto ainda herdam o mesmo comportamento. No entanto, embora o alias seja considerado igual ao tipo base, as instâncias do novo tipo não são consideradas iguais ao tipo base.

Para verificar se um valor é uma instância de um alias de tipo ou do tipo base, você pode usar o operador de comparação de menor-ou-igual `<=`. Isso permite que você aceite qualquer tipo na cadeia de aliases, até o tipo base. Aqui está um exemplo:

```
Id == Text   # verdadeiro, já que Id é um alias de Text
x = Id(123)  # id: Id = '123'
x == Text    # falso, no entanto x é do tipo Id, não Text
x == Id      # verdadeiro, como o esperado x é do tipo Id
x <= Text    # verdadeiro, já que x é do tipo Id, que é um alias de Text
```

Essa funcionalidade permite uma validação refinada em tipos específicos, mantendo a flexibilidade de usar diferentes aliases para o mesmo tipo subjacente.

limitação: não é possível criar alias para `Any`, `Type` ou `Method`

Restrições de tipo

No FatScript, você pode declarar restrições de tipo para parâmetros de método. Quando um método é chamado, o argumento é verificado automaticamente em relação à restrição de tipo. Se o argumento não for do tipo esperado ou um de seus subtipos, um `TypeError` é gerado.

Se a restrição de tipo for um tipo base, qualquer subtipo desse tipo também será aceito como argumento. No entanto, se a restrição de tipo for um subtipo, somente argumentos que correspondam ao subtipo serão aceitos. Aqui está um exemplo:

```
generalista = (x: Text) -> x
restritivo = (x: Id) -> x
```

Neste exemplo, o método `generalista` aceita argumentos `Text` e `Id`, porque `Id` é um subtipo de `Text`. O método `restritivo` aceita apenas argumentos `Id` e não `Text`, porque `Id` é um subtipo de `Text`, mas não o contrário.

É importante enfatizar que os tipos personalizados são derivados de `Scope`. Nesse contexto, `Scope` seria o tipo generalista para, por exemplo, o tipo personalizado `Carro`.

Mixin (avançado)

Ao definir um tipo, você pode adicionar os recursos de um tipo existente simplesmente mencionando-o na definição de tipo. Isso é chamado de inclusão de tipo ou mixin.

Por exemplo, para criar um novo tipo `CarroAlugado` com as propriedades de `Carro` e uma propriedade adicional `preco`, você pode escrever:

```
CarroAlugado = {
  # Inclusões
  Carro

  # Propriedade adicional
  preco: Number
}

CarroAlugado(50)  # { cor: Text = 'branco', km: Number = 0, preco: Number = 50 }
```

Se uma propriedade não estiver definida no novo tipo, ela herdará o valor padrão do tipo incluído. No exemplo acima, as propriedades `cor` e `km` do `Carro` estão presentes no `CarroAlugado`, com seus valores padrão.

Herdando métodos de protótipo

Suponha que continuemos a partir do exemplo anterior do tipo `TipoComMembrosDePrototipo` que tem duas propriedades `a` e `b`, e três métodos de protótipo `setA`, `setB` e `soma`. Para criar um novo tipo `ComMaisMembros` que adiciona uma propriedade `c`, um método `setC` e substitui o método `soma`, você pode escrever:

```
ComMaisMembros = {
  # Inclusões
  TipoComMembrosDePrototipo

  # Propriedades (parâmetros da instância)
  ~ a: Number
  ~ b: Number
  ~ c: Number

  # Membros de protótipo (métodos)
  setC = (novoC: Number) -> self.c = novoC
  soma = (): Number      -> self.a + self.b + self.c
}
```

redeclarando as propriedades permite que o novo tipo também aceite argumentos no momento da instanciação, por exemplo: `ComMaisMembros(1, 2, 3)` define `a`, `b` e `c`

Ao criar uma nova instância de `ComMaisMembros`, todos os quatro métodos de protótipo `setA`, `setB`, `setC` e `soma` estarão disponíveis.

Observe que se houver uma redefinição de uma propriedade ou método no novo tipo, a nova definição terá precedência.

Conversão de tipos

No FatScript, o símbolo `*` é usado para conversão de tipo, permitindo que você trate um tipo de dado como outro sem alterar os dados subjacentes. Essa capacidade é especialmente útil para especificar explicitamente o tipo ou para tratar valores como tipos compatíveis, por exemplo:

```
time.format(Epoch * 1688257765448)  # trata o número como um valor de Época Unix
```

Aceitação flexível de tipos

FatScript oferece flexibilidade na aceitação de tipos implementando um sistema baseado na inclusão de tipos. Isso cria tipos inter-relacionados que podem ser usados de forma intercambiável em um método ou como itens de uma Lista.

Quando você define um tipo, é possível incorporar um ou mais tipos adicionais dentro dessa definição. Por exemplo, os tipos `A`, `B`, e `C`. Se os tipos `B` e `C` incluem o tipo `A` em suas definições, eles são vistos como compartilhando o mesmo conjunto de características derivadas de `A`. Isso significa que `B` e `C` são considerados tipos irmãos sob o guarda-chuva de `A`.

Este sistema permite que um método que foi projetado para aceitar um objeto do tipo `B` também seja capaz de aceitar um objeto do tipo `C`, e vice-versa. Isso ocorre pelo fato de que ambos os tipos `B` e `C` compartilham uma base comum no tipo `A`.

Aqui está como isso parece no código:

```
A = (_)
B = (A, b = true)
C = (A, c = true)

# o método1 aceita tanto B quanto C, porque ambos incluem A
method1 = (a: A) -> 'valid'

# o método2 aceita C, já que B e C incluem o mesmo conjunto de tipos
# (tornando-os tipos irmãos)
method2 = (x: B) -> 'valid'

# essa lógica também se aplica a tipos de Lista, como visto com mixedList
mixedList: List/A = [ B(), C() ]
```

a flexibilidade do tipo só é possível se o tipo de dados é baseado em `Scope`

Advertência

Você pode ter que verificar explicitamente o tipo, por exemplo, `x == B` dentro do corpo do método se você quiser lidar apenas com `B`, mas não com `C` em seu método. Ou você pode criar um alias, por exemplo, `D = A` e usar `C = (D, c = true)` como inclusão de tipo para evitar completamente o comportamento flexível.

Tipos compostos

No FatScript, tipos compostos permitem que você defina estruturas de dados complexas compostas por tipos mais simples. Eles são representados usando barras `/` para separar os tipos na definição do tipo composto.

Vamos ver alguns exemplos e entender como os tipos compostos funcionam:

1. `ListOfNumbers = List/Number`, define um tipo composto `ListOfNumbers`, que é uma lista que só pode conter números.
2. `Matrix = List/List/Number`, define um tipo composto `Matrix`, que é uma lista de listas que só pode conter números.
3. `MethodReturningListOfNumbers = Method/ListOfNumbers`, define um tipo composto `MethodReturningListOfNumbers`, que é um método que retorna um `ListOfNumbers`.
4. `NumericScope = Scope/Number` define um tipo composto `NumericScope`, que é um escopo cujas entradas podem ser apenas do tipo número.

Veja também

- Pacote type

Any

Um tipo virtual que engloba todos os tipos e nenhum tipo ao mesmo tempo.

Tipo padrão

`Any` é o tipo inferido e o tipo de retorno quando nenhum tipo é explicitamente anotado em um método. Por exemplo:

```
identity = _ -> _
```

é equivalente a:

```
identity = (_: Any): Any -> _
```

Usar `Any`, seja implicitamente ou explicitamente, desabilita a verificação de tipos para um parâmetro. A anotação explícita pode ser útil em casos em que você deseja deixar claro que está dando flexibilidade ao tipo que é aceito.

Ser muito liberal com `Any` pode tornar seu código menos previsível e mais difícil de manter. É geralmente recomendado ser mais específico com as anotações de tipo sempre que possível:

```
# Exemplo de uso de Any que pode levar a problemas

console <- fat.console

doubleIt = (arg: Any): Void -> console.log(arg * 2)

doubleIt(2)    # imprime: '4'
doubleIt('a')  # gera: Error: unsupported expression > Text <multiply> Number
```

Este exemplo mostra que, embora a anotação de tipo `Any` permita flexibilidade no tipo do parâmetro, também pode resultar em comportamento inesperado se um argumento de um tipo inesperado for passado. Ao ser mais específico com a anotação de tipo, como `Number`, você pode tornar seu código mais previsível e autoexplicativo.

```
# Exemplo de uso de uma anotação de tipo específica para maior previsibilidade

console <- fat.console

doubleIt = (num: Number): Void -> console.log(num * 2)

doubleIt(2)    # imprime: '4'
doubleIt('a')  # gera: TypeError: type mismatch > num
```

Ao usar `Number` como a anotação de tipo, o método `doubleIt` agora é mais específico e só aceita argumentos do tipo `Number`.

Comparação

A única operação possível com `Any` é a comparação com ele, mas note que `Any` aceita todos os valores indistintamente, então não há uso prático para isso:

```
null      == Any  # verdadeiro
true      == Any  # verdadeiro
12345     == Any  # verdadeiro
'abcd'    == Any  # verdadeiro
[ 1, 2 ]  == Any  # verdadeiro
{ a = 8 } == Any  # verdadeiro
```

> as comparações com `Any` não podem ser usadas para verificar a presença de um valor em um escopo, pois até mesmo `null` é aceito

Void

Quando você olha para o 'Vazio', apenas 'nulo' pode ser visto.

Tem alguém aí fora?

Uma entrada é avaliada como `null` se não estiver definida no escopo atual.

Você pode comparar com `null` usando igualdade `==` ou desigualdade `!=`, como:

```
a == null  # verdadeiro, se 'a' não estiver definida
0 != null  # verdadeiro, porque 0 é um valor definido
```

Tenha em mente que você não pode declarar uma entrada sem valor no FatScript.

Embora você possa atribuir `null` a uma entrada, isso causa comportamentos diferentes, dependendo se a entrada já existe no escopo e se é mutável ou não:

- Se uma entrada ainda não foi declarada, atribuir `null` não tem efeito.
- Se já existe e é imutável, atribuir `null` gera um erro.
- Se já existe e é mutável, atribuir `null` remove a entrada.

Declaração de exclusão

Atribuir `null` a uma entrada mutável é o mesmo que excluir essa entrada do escopo. Se excluído, nada é lembrado sobre essa entrada no escopo, nem mesmo seu tipo original.

```
~ m = 4   # entrada de número mutável
m = null  # exclui m do escopo
```

> "valores" null são sempre mutáveis, pois na verdade nada é armazenado sobre eles e, portanto, são o único tipo de "valor" que pode fazer a transição de um estado mutável para um estado imutável quando "reatribuído"

Comparações

Você também pode usar `Void` para verificar o valor de uma entrada, como:

```
()    == Void  # verdadeiro
null  == Void  # verdadeiro
false == Void  # falso
0     == Void  # falso
''    == Void  # falso
[]    == Void  # falso
{}    == Void  # falso
```

Observe que `Void` só aceita `()` e `null`.

Formas de vazio

Em FatScript, o conceito de "vazio" ou a ausência de um valor pode ser representado de duas maneiras: usando `null` ou parênteses vazios `()`. Eles são efetivamente idênticos, em termos de comportamento no código:

```
null  == null  # verdadeiro
()    == null  # verdadeiro
()    == ()    # verdadeiro
```

Usando null

A palavra-chave `null` denota explicitamente a ausência de um valor. É comumente usada em cenários onde um parâmetro ou valor de retorno pode não apontar para nenhum valor.

```
method(null, otherParam)

var = null
```

Também pode ser usado para tornar um parâmetro opcional, permitindo que métodos sejam chamados com diferentes números de argumentos:

```
method = (mandatory: Text, optional: Text = null) -> {
  ...
}
```

> `null` pode ser usado explicitamente em qualquer contexto onde uma ausência de valor precisa ser representada

Usando parênteses vazios

Quando usado no contexto de retornos de métodos, `()` pode significar que o método não retorna nenhum valor significativo.

```
fn = -> {
  doSomething

  ()
}
```

Aqui, `fn` executa alguma ação e então usa `()` para indicar a ausência de um valor de retorno significativo, retornando efetivamente void.

A diferença reside no estilo de código, então isso é apenas uma sugestão, não uma regra rígida.

> nas versões modernas do interpretador, parênteses vazios `()` são tratados como `null`, garantindo um comportamento consistente, mas, versões anteriores exigiam o uso explícito de `null` para denotar a ausência de um valor de retorno

Veja também

- Extensões do protótipo Void

Boolean

Booleanos são muito primitivos, eles só podem ser 'verdadeiro' ou 'falso'.

Comparação

Além de igualdade `==` e desigualdade `!=`, os booleanos também aceitam os seguintes operadores:

& AND lógico

```
true  & true  == true
true  & false == false
false & true  == false
false & false == false
```

AND interrompe a expressão se o lado esquerdo for falso

| OR lógico

```
true  | true  == true
true  | false == true
false | true  == true
false | false == false
```

OR interrompe a expressão se o lado esquerdo for verdadeiro

% XOR lógico (OR exclusivo)

```
true  % true  == false
true  % false == true
false % true  == true
false % false == false
```

XOR sempre avalia ambos os lados da expressão

Operador Bang

`!!` converte qualquer tipo em booleano, assim:

- null -> false
- zero (número) -> false
- não-zero (número) -> true
- vazio (texto/lista/escopo/chunk) -> false
- não-vazio (texto/lista/escopo/chunk) -> true
- método -> true
- erro -> false

AND/OR lógicos (`&`, `|`) e fluxos condicionais (`=>`, `?`) converterão implicitamente para booleano

Veja também

- Extensões do protótipo Boolean
- Controle de fluxo

Number

Um conceito matemático usado para contar, medir e fazer outras coisas de matemáticas.

Declaração

O tipo `Number` é implementado como `double`. Veja como declarar um número:

```
a = 5              # declaração de número (imutável)
b: Number = 5      # mesmo efeito, com verificação de tipo
c: Number = a      # iniciando com o valor da entrada, também 5
d = 43.14          # com casas decimais
```

Para declarar uma entrada mutável, coloque o operador til antes:

```
~ a = 6  # entrada de número mutável
a += 1   # adiciona 1 a 'a', resultando em 7
```

Operações com números

Números aceitam várias operações:

- `==` igual
- `!=` diferente
- `+` soma
- `-` subtração
- `*` multiplicação
- `/` divisão
- `%` módulo
- `**` potência
- `<` menor
- `<=` menor ou igual
- `>` maior
- `>=` maior ou igual
- `&` AND lógico
- `|` OR lógico

Ressalvas

Para operações lógicas e controle de fluxo, lembre-se de que zero é considerado falso e um não-zero é considerado verdadeiro.

Para operadores de igualdade, embora `0` e `null` sejam avaliados como falsos, no FatScript eles não são iguais:

```
0 == null  # falso
```

Precisão

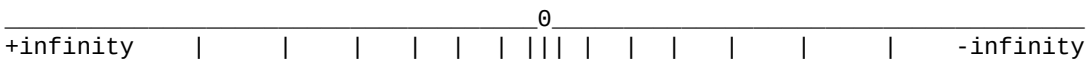
Embora a precisão aritmética de um `IEEE 754 double` seja maior, o `fry` utiliza truques de arredondamento para melhorar a legibilidade humana ao imprimir sequências longas de noves ou zeros decimais como texto. Além disso, ele usa um epsilon de `1.0e-06` para comparações de 'igualdade' entre números.

Em 99,999% dos casos de uso, essa abordagem fornece tanto comparações mais convenientes quanto números mais naturais:

```
# Epsilon de igualdade
x = 1.0e-06
x: Number = 0.000001

# Diferenças menores são tratadas como o "mesmo" número pela comparação
x == 0.0000015
Boolean: true  # a diferença de 0,0000005 é ignorada
```

Os números de ponto flutuante não são distribuídos uniformemente na reta numérica. Eles são densos em torno de 0 e, à medida que a magnitude aumenta, o 'delta' entre dois valores expressivos aumenta:



o maior inteiro contíguo é 9.007.199.254.740.992 ou 2^53

Ainda é possível ter números muito maiores, em torno de 10^308, que é:

```
100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Tenha em mente que se você somar 1 a 10^308, não importa quantas vezes você o fizer, sempre resultará no mesmo valor! Você precisa adicionar pelo menos algo próximo a 10^293 em uma única operação para que seja considerado, pois os números precisam ser de ordens de magnitude semelhantes. Para lidar de maneira discreta com números que excedem 2^53 considere utilizar o tipo HugeInt.

Além disso, a palavra-chave `infinity` fornece uma representação clara e inequívoca de valores que se elevam aos reinos além dos maiores números expressáveis, aproximando-se da infinitude teórica.

Veja também

- Extensões do protótipo Number
- Biblioteca math

HugeInt

Um tipo de dado numérico avançado projetado para lidar com inteiros muito grandes.

Declaração

O tipo `HugeInt` suporta inteiros de até 4096 bits. Veja como você pode declarar um `HugeInt`:

```
h = 0x123456789abcdef  # declaração de um HugeInt
```

> `HugeInt` é sempre expresso em formato hexadecimal

Operando HugeInts

`HugeInt` suporta uma variedade de operações, tornando-o versátil para cálculos complexos:

- `==` igual
- `!=` diferente
- `+` soma
- `-` subtração
- `*` multiplicação
- `/` divisão
- `%` módulo
- `**` potência
- `<` menor
- `<=` menor ou igual
- `>` maior
- `>=` maior ou igual
- `&` AND lógico
- `|` OR lógico

Cuidados

No FatScript, `HugeInt` é projetado especificamente como um tipo não sinalizado, e, portanto, só pode representar valores positivos.

Interações entre `HugeInt` e outros tipos numéricos, como Number, não estão diretamente disponíveis. Para realizar tais operações, você deve converter o valor para `HugeInt` usando seu construtor (disponível através das extensões de protótipo).

Precisão

`HugeInt` oferece alta precisão para inteiros muito grandes, essencial em campos como criptografia e computações de grande escala. Esta precisão permanece consistente em toda a sua faixa.

```
prime = 0xfffffffffffffffc90fdA... # um grande número primo
```

Ao contrário dos números de ponto flutuante, `HugeInt` representa valores inteiros discretos, mantendo precisão e espaçamento consistentes em toda a sua faixa:

```
0_____________________________________________
|    |    |    |    |    |    |    |    | estouro
```

> o valor máximo é 2^4096 - 1, equivalente a um número com 1233 dígitos decimais ou o literal 0xfff... (com 1024 repetições da letra f)

`HugeInt` é particularmente adequado para cenários que exigem aritmética inteira exata sem erros de arredondamento, especialmente ao lidar com valores muito além dos limites do tipo Number. É importante garantir que todas as operações permaneçam dentro de sua capacidade suportada, pois exceder esse limite acarretará em um `ValueError`.

Veja também

- Extensões do protótipo HugeInt

Text

Textos podem conter muitos caracteres e são às vezes chamados de strings.

Declaração

Entradas de texto são declaradas usando aspas:

```
a = 'hello world'        # declaração de texto inteligente
a = "hello world"        # declaração de texto bruto
a: Text = 'hello world'  # inteligente, opcionalmente verboso
```

Manipulando texto

Concatenação

No FatScript, você pode concatenar, ou juntar, dois textos usando o operador `+`. Essa operação conecta os dois textos em um. Por exemplo:

```
x1 = 'ab' + 'cd'  # Retorna 'abcd'
```

Subtração de texto

FatScript também suporta uma operação de subtração de texto usando o operador `-`. Essa operação remove uma substring especificada do texto. Por exemplo:

```
x2 = 'ab cd'
x2 - ' ' == 'abcd'  # Retorna true
```

No exemplo acima, o caractere de espaço `' '` é removido do texto original `'ab cd'`, resultando em `'abcd'`.

Seleção de texto

A seleção permite que você acesse partes específicas de um texto usando índices. No FatScript, você pode usar índices positivos ou negativos. Os índices positivos começam do início do texto (`0` é o primeiro caractere), e os índices negativos começam do final do texto (`-1` é o último caractere).

> para uma explicação detalhada sobre o sistema de indexação no FatScript, consulte a seção sobre acesso e seleção de itens em List

Quando apenas um índice é passado para a função de seleção, um único caractere do texto é selecionado. Quando um intervalo é passados para a função, um fragmento do texto é selecionado. Essa seleção é inclusiva, o que significa que inclui os caracteres nos índices inicial e final, a menos que se use o operador de intervalo semiaberto `..<`, exclusivo no lado direito.

Assim como com as listas, acessar itens que estão fora dos índices válidos irá gerar um erro. Para seleções, não são gerados erros ao acessar índices fora dos limites; em vez disso, um texto vazio é retornado.

```
x3 = 'exemplo'
x3(1)     # 'x'
x3(2, 4)  # 'emp'
x3(..2)   # 'exe'
x3(..<2)  # 'ex'
```

Caracteres especiais

Caracteres como aspas `'` / `"` podem ser escapados com a barra invertida `\`.

```
'Rock\'n\'roll'
"Onde fica \"aqui\"?"
```

> você só precisa escapar as aspas do mesmo tipo usadas como delimitador de texto

Outras sequências de escape suportadas são:

- backspace `\b`

- nova linha `\n`
- retorno de carro `\r`
- tabulação `\t`
- escape `\e`
- octeto em representação base-8 `\ooo`
- a própria barra invertida `\\`

Textos inteligentes

Quando declarado com aspas simples `'`, o modo inteligente é habilitado e a interpolação é realizada para qualquer código envolto em chaves `{...}`:

```
texto = 'mundo'
interpolado = 'olá {texto}'  # resulta em 'olá mundo'
```

> o template é processado em uma camada com acesso ao escopo atual

Observe que o uso de novas linhas ou outros textos inteligentes dentro do template de interpolação não é suportado, mas você pode fazer chamadas de método, se precisar compor o resultado com algo mais complexo.

Você pode evitar a interpolação escapando o colchete de abertura:

```
escapado = 'olá \{texto}'  # resulta em 'olá {texto}'
```

Alternativamente, você pode evitar a interpolação usando textos brutos.

Textos brutos

Quando declarado com aspas duplas `"`, o modo de texto bruto é assumido e a interpolação é desativada.

Exemplo de modo inteligente vs. modo bruto:

```
'Sou inteligente: {interpolado}'  # usando o valor do exemplo anterior
Sou inteligente: olá mundo        # substituição ocorre

"Sou bruto: {interpolado}"  # colchetes são apenas caracteres comuns
Sou bruto: {interpolado}    # nenhuma interpolação ocorre
```

Operações com textos

- `==` igual
- `!=` diferente
- `+` soma (concatenar)
- `-` subtração (remove substring)
- `<` menor (alfanumérico)
- `<=` menor ou igual (alfanumérico)
- `>` maior (alfanumérico)
- `>=` maior ou igual (alfanumérico)
- `&` AND lógico (convertido para booleano)
- `|` OR lógico (convertido para booleano)

Codificação

FatScript é projetado para operar com textos codificados em UTF-8. Essa escolha de design reconhece a prevalência desses sistemas de codificação e otimiza a linguagem para ampla compatibilidade.

UTF-8 é um sistema de codificação de vários bytes capaz de representar qualquer caractere no padrão Unicode. Este esquema de codificação de caracteres universais usa de 8 a 32 bits para representar um caractere, permitindo a representação de uma vasta gama de símbolos de diversas línguas e sistemas de escrita. Notavelmente, os primeiros 128 caracteres (0-127) do UTF-8 se alinham precisamente com o conjunto ASCII, tornando qualquer texto ASCII uma string válida codificada em UTF-8.

No FatScript, o tipo de dados Text é uma sequência de caracteres Unicode, inerentemente codificada em UTF-8, portanto, operações como `text.size`, `text(index)` e `text(1..4)` irão contar, acessar ou fatiar corretamente o texto, independentemente da complexidade dos caracteres. Essas operações consideram um caractere UTF-8 multi-byte completo como uma única unidade, garantindo um comportamento correto e previsível.

Veja também

- Extensões do protótipo Text

Method

Métodos são receitas que podem receber argumentos para "preencher as lacunas em branco".

Definição

Um método é definido anonimamente com uma seta fina `->`, assim:

```
<parâmetros> -> <receita>
```

Os parâmetros podem ser omitidos se nenhum for necessário:

```
-> <receita>  # aridade zero
```

Para registrar um método no escopo, atribua-o a um identificador:

```
<identificador> = <parâmetros> -> <receita>
```

Parâmetros dentro do escopo de execução de um método são imutáveis, garantindo que as operações do método não alterem seu estado original. Para um comportamento mutável, considere passar um escopo ou utilizar um tipo personalizado capaz de encapsular múltiplos valores e estados.

Parâmetros opcionais

Enquanto assinaturas de métodos tipicamente requerem um número fixo de parâmetros obrigatórios, FatScript suporta parâmetros opcionais através de valores padrão:

```
greet = (mensagem: Texto, nome: Texto = 'Mundo') -> {
  "Olá, {nome}, {mensagem}"
}
```

Neste exemplo, o parâmetro `nome` é opcional, assumindo 'Mundo' como valor padrão se não for fornecido. Esta característica permite invocações de métodos mais flexíveis.

Tratamento de argumentos

Chamadas de métodos em FatScript são projetadas para aceitar o fornecimento de mais argumentos do que o requerido; argumentos extras são simplesmente ignorados. Este comportamento faz parte do design da linguagem para aumentar a flexibilidade e desempenho.

Auto-retorno

FatScript usa o auto-retorno, ou seja, o último valor é retornado automaticamente:

```
resposta: Method = (aGrandeQuestao) -> {
  # TODO: explicar a vida, o universo e tudo mais
  42
}

resposta("6 x 7 = ?")  # retorna: 42
```

Chamadas automáticas

O FatScript introduz um recurso único que simplifica as chamadas de métodos quando não envolvem argumentos. Esse recurso é conhecido como "truque de chamada automática" e oferece várias vantagens principais:

- **Redução de Código Desnecessário**: Reduz a necessidade de parênteses, tornando o código mais limpo e conciso, para métodos sem parâmetros que agem como propriedades.
- **Computação Dinâmica**: Permite a computação dinâmica com saídas que podem mudar com base no estado interno do objeto ou no estado global.
- **Execução Diferida**: Habilita a execução diferida, útil em programação assíncrona e padrões de inicialização complexos.

Uso básico

No FatScript, um método definido sem parâmetros é executado "automágicamente" quando referenciado:

```
foo = {
  bar = -> 'Olá!'
}

# Ambas as linhas abaixo produzem 'Olá!'
foo.bar()  # chamada explícita
foo.bar    # chamada automática
```

Referenciando

Para referenciar um método sem acionar o recurso de chamada automática, você pode usar a sintaxe de obtenção:

```
foo('bar')  # gera uma referência para foo.bar, sem chamá-lo
```

O FatScript também oferece as palavras-chave `self` e `root` para referenciar métodos nos níveis local e global, respectivamente:

```
self('meuMetodoLocal')
root('meuMetodoGlobal')
```

Optando por não usar chamadas automáticas

O operador til ~ permite que você contorne o recurso de chamada automática, proporcionando flexibilidade no manuseio de métodos:

```
# Ambas as linhas abaixo buscam a referência do método, sem chamá-lo
foo.~bar
~ meuMetodo
```

Ou você pode simplesmente encapsular a chamada do método em outro método (anônimo):

```
-> foo.bar
```

Aviso: passando métodos como argumentos

Há uma exceção importante com relação a passagem de métodos como argumentos, especificamente quando se trata de um método local:

```
outro(bar)  # passa `bar` como uma referência, sem executá-lo
```

> no entanto, isso não se aplica com encadeamento: `outro(foo.bar)` passa o resultado de `bar`, e não a referência

Para passar o valor resultante do método local `bar`, uma chamada explícita tem que ser feita:

```
outro(bar())
```

> este comportamento pode parecer contra-intuitivo, mas é extremamente útil em diversos casos de uso, como, por exemplo, ao passar métodos para reduce, para uma tarefa assíncrona, para uma operação de mapeamento etc.

Argumento implícito

Uma conveniência oferecida pelo FatScript é a possibilidade de fazer referência a um valor passado para o método sem precisar especificar um nome a ele explicitamente. Neste caso, o argumento implícito é então representado pelo sublinhado _.

Aqui está um exemplo que ilustra o uso do argumento implícito:

```
dobro = -> _ * 2
dobro(3)  # saída: 6
```

Você pode usar um argumento implícito sempre que precisar realizar uma operação simples em um único parâmetro sem atribuir um nome específico a ele, mas note que o método deve ter aridade zero para ativá-lo.

Veja também

- Extensões do protótipo Method

List

Listas são coleções ordenadas de itens do mesmo tipo, acessados por índice.

Definição

As listas são definidas com colchetes `[]`, como no exemplo a seguir:

```
lista: List/Text = [ 'maçã', 'pizza', 'pêra' ]
```

Listas não permitem a mistura de tipos. O tipo de uma lista é determinado pelo primeiro item adicionado a ela; consequentemente, as listas vazias não têm tipo.

As listas pulam posições vazias, então um item que avalia para `null` é ignorado:

```
a = 1
c = 3
[ a, b, c ]  # retorna: [ 1, 3 ] (b é ignorado)
```

Acesso

Itens individuais

Os itens da lista podem ser acessados individualmente com chamada de índice baseado em zero:

```
lista(0)  # 'maçã'
lista(2)  # 'pêra'
```

Valores negativos indexam a lista de trás para frente, começando em -1 como o último item:

```
lista(-1)  # 'pêra'
```

O acesso a itens que estão fora dos índices válidos gera um erro:

```
         0        1        2       > 2
Erro   [ 'maçã', 'pizza', 'pêra' ] Erro
 < -3       -3       -2      -1
```

Seleções

Os índices de início e fim funcionam exatamente da mesma maneira que ao acessar itens individuais; então, os valores negativos contam a partir do último item e podem ser regressivos. No entanto, ao usar intervalos, nenhum erro é gerado ao acessar índices fora dos limites; em vez disso, uma lista vazia é retornada.

```
lista(0..0)   # [ 'maçã' ]
lista(4..8)   # []
lista(1..-1)  # [ 'pizza', 'pêra' ]
```

Um índice pode ser deixado em branco, e o início a partir do primeiro item ou o fim no último item é assumido:

```
lista(..1)   # [ 'maçã', 'pizza' ]
lista(1..)   # [ 'pizza', 'pêra' ]
```

Aviso de descontinuação

A opção de passar um segundo argumento (usando uma vírgula e não ponto-ponto) para realizar uma seleção está sendo descontinuada à partir da versão 3.x.x:

```
lista(0, 0)   # [ 'maçã' ]           # sintaxe descontinuada
lista(4, 8)   # []                   # sintaxe descontinuada
lista(1, -1)  # [ 'pizza', 'pêra' ]  # sintaxe descontinuada
```

Listas aninhadas

Uma matriz pode ser usada e acessada da seguinte maneira:

```
matriz = [
  [ 1, 2, 3 ]
  [ 4, 5, 6 ]
]

matriz(1)(0)  # retorna 4 (1: segunda linha, em seguida, 0: primeiro índice)
```

para simplificar, o exemplo usa uma matriz 2D, mas poderia ser n-dimensional

Operações

- == igual
- != diferente
- + adição (efeito de concatenação)
- - subtração (efeito de diferença)
- & AND lógico
- | OR lógico

AND/OR lógicos avaliam listas vazias como `false`, caso contrário `true`

Adição de lista (concatenação)

A operação de adição de listas permite combinar duas listas em uma nova lista:

```
x = [ 1, 2, 2, 3 ]
y = [ 3, 3, 4, 4 ]

x + y  # resultado: [ 1, 2, 2, 3, 3, 3, 4, 4 ]
```

Nesse caso, ao usar o operador de adição `+` para unir as listas `x` e `y`, os elementos de ambas as listas são combinados em uma única lista. A ordem dos elementos na lista resultante é determinada pela ordem em que as listas foram adicionadas.

não há remoção de elementos duplicados durante a concatenação

Concatenação rápida

Para melhor desempenho, você pode aproveitar o operador `+=`, por exemplo:

```
~ lista += [ valor ]  # mais rápido

# mesmo efeito que
~ lista = []
lista = lista + [ valor ]  # concatenação (mais lenta)
```

Outro detalhe do operador `+=`, que se aplica também a outros tipos, é a inicialização automática por omissão, onde caso a entrada ainda não tenha sido declarada anteriormente, atua como uma simples atribuição.

Subtração de lista (diferença)

A operação de subtração de listas, permite remover os elementos do segundo operando que estão presentes no primeiro operando, resultando em uma lista contendo apenas valores únicos:

```
x = [ 1, 2, 2, 3 ]
y = [ 3, 3, 4, 4 ]

x - y  # resultado: [ 1, 2 ]
y - x  # resultado: [ 4 ]
```

Nesse caso, ao subtrairmos a lista `y` da lista `x`, os elementos com valor 3 são removidos, já que estão presentes em ambas as listas. O resultado é a lista `[1, 2]`. Da mesma forma, ao subtrairmos a lista `x` da lista `y`, o único elemento restante é o valor 4.

apenas valores exatamente idênticos são removidos durante a subtração

Veja também

- Extensões do protótipo List
- Mapeando uma lista

Scope

Um escopo é semelhante a um dicionário, onde chaves estão associadas a valores.

Definição

Escopos são definidos usando chaves `{}`, conforme mostrado abaixo:

```
meuEscopoBacana = {
  lugar = 'aqui'
  quando = 'agora'
}
```

Escopos armazenam entradas em ordem alfabética, uma característica que se torna aparente ao mapear sobre um escopo.

Acesso

Há três maneiras de acessar diretamente as entradas dentro de um escopo.

Sintaxe de ponto

```
meuEscopoBacana.lugar  # retorna: 'aqui'
```

Sintaxe de obtenção

```
# assumindo que prop = 'lugar'
meuEscopoBacana(prop)  # retorna: 'aqui'
```

Em ambos os métodos, se a propriedade não estiver presente, `null` é retornado. Se o escopo externo não for encontrado, um erro é gerado.

Sintaxe de encadeamento opcional

Use o operador interrogação-ponto `?.` para encadear com segurança escopos externos potencialmente inexistentes:

```
naoExistente?.prop  # retorna null
```

A sintaxe de encadeamento opcional não gera um erro quando o escopo externo é `null`.

Operações

- == igual
- != diferente
- + adição (efeito de mesclagem)
- - subtração (efeito de diferença)
- & AND lógico
- | OR lógico

AND/OR lógico avaliam escopos vazios como `false`, caso contrário, `true`

Adição de escopo (mesclagem)

O segundo operando age como um patch para o primeiro:

```
x = { a = 1, b = 3 }
y = { b = 2 }

x + y  # resulta em { a = 1, b = 2 }
y + x  # resulta em { a = 1, b = 3 }
```

valores do segundo operando substituem os do primeiro

Subtração de escopo (diferença)

A subtração remove elementos do primeiro operando que são idênticos aos do segundo:

```
x = { a = 1, b = 3 }
y = { a = 1 }

x - y  # resulta em { b = 3 }
```

apenas valores exatamente idênticos são removidos

Blocos Escopados

Blocos Escopados em FatScript permitem a execução de declarações dentro do contexto de um escopo específico:

```
objeto.{
  # Declarações executadas no contexto de 'objeto'
}
```

Aqui, `objeto` é o escopo alvo. Dentro do bloco, você pode acessar e modificar diretamente as propriedades de `objeto`.

Características

- **Isolamento**: entradas declaradas dentro de um Bloco Escopado são locais para aquele escopo e não afetam o escopo externo.
- **Acesso ao Escopo Externo**: Blocos Escopados podem acessar entradas do escopo externo.

Exemplo

```
x = {}

x.{
  a = 5      # 'a' agora é uma propriedade de 'x'
  b = a + 3  # 'b' agora também é uma propriedade de 'x'
}
```

Interações de escopos

FatScript utiliza mecanismos sofisticados para gerenciar variáveis em diferentes escopos, aproveitando conceitos de escopo léxico e sombreamento para fornecer capacidades de programação poderosas. Esta seção explora esses mecanismos, incluindo nuances de atribuição, comportamentos de incremento/decremento e o uso inovador do operador `+=` para alternância de booleanos.

Atribuição

O operador de atribuição (=) copia valores de escopos externos para o escopo atual, definindo um novo valor:

```
~ n = 1
x = {}
x.{ ~ n = n }  # agora x.n == 1, e x.n é independente de root.n
x.{ c = n }    # tem efeito semelhante, porém 'c' é imutável
```

o mesmo conceito se aplica ao código executado em um escopo de método

Ressalva

Usar `~ n = n + 1` dentro de um bloco ou método adiciona um novo 'n' no escopo atual, inicializado com o valor de `n + 1` do escopo envolvente mais próximo, sem alterar o 'n' externo.

Incrementando e decrementando

Operações de incremento (+=) e decremento (-=) interagem com o escopo de variáveis de uma maneira diferente. Essas operações buscam a instância mais próxima de uma variável, começando do escopo atual e movendo-se para fora recursivamente, e então modificam essa instância diretamente.

```
~ outerN = 1
fn = -> {
  outerN += 1  # alveja e incrementa 'outerN' no escopo externo
}
```

Auto-inicialização com +=

FatScript também fornece um comportamento especial em relação ao operador de incremento (+=). Se não existir o incremento funciona como uma atribuição regular como se você tivesse escrito o seguinte para `n += 1`:

```
n == Void ? n = 1 : n += 1
```

O recurso de auto-inicialização pode ser particularmente útil quando usado em combinação com entradas dinâmicas para programação dinâmica.

> este recurso está disponível exclusivamente para o operador de incremento, decremento não pode inicializar valores não existentes

Alternância de booleanos com `+=`

Geralmente, booleanos não permitem operações de adição. FatScript, no entanto, estende a funcionalidade do operador `+=` para tipos booleanos, permitindo um mecanismo de alternância intuitivo dentro de escopos internos.

A expressão `flag += !flag` alterna efetivamente o valor booleano, mesmo quando `flag` é definido em um escopo externo.

> no caso particular de booleanos, a única distinção entre `=` e `+=` é o escopo

Outros operadores de atribuição compostos

Da mesma forma, outras operações de atribuição compostas, como `*=`, `/=`, `%=` e `**=`, são suportados por tipos numéricos e respeitam as mesmas regras de escopo que se aplicam às operações de incremento e decremento.

Veja também

- Entradas dinâmicas
- Extensões do protótipo Scope
- Mapeando um escopo

Error

Há grande sabedoria em esperar pelo inesperado também.

Subtipos padrão

Enquanto alguns erros genéricos, como problemas de sintaxe, importações inválidas, etc. são gerados com o tipo base Error, outros são subtipados.

Veja as definições nas extensões do protótipo Error.

Declaração

Erros também podem ser declarados explicitamente; você deve usar o construtor de tipo:

```
_ <- fat.type.Error

Error('ocorreu um erro')  # gera um erro genérico

MyMistake = Error
MyMistake('ocorreu outro erro')  # gera um erro do subtipo MyMistake
```

Comparações

Erros sempre avaliam como falso:

```
Error() ? 'é verdadeiro' : 'é falso'  # é falso
```

Erros são comparáveis ao seu tipo:

```
Error() == Error  # verdadeiro
```

leia também a sintaxe de comparação de tipo

Uma maneira ingênua de lidar com erros poderia ser:

```
_ <- fat.console
# lidando com o erro retornado
talvezFalhe() <= Error => log('um erro aconteceu')
_                      => log('sucesso')
```

isso só funciona se a opção `-e / continuar em caso de erro` estiver definida

Uma outra maneira ingênua de lidar com errors, mas que funciona sempre é utilizar uma operação padrão:

```
talvezFalhe() ?? log('um erro aconteceu')
```

Embora a abordagem ingênua possa funcionar, uma maneira mais adequada de lidar com erros é definindo um manipulador de erro usando o método `trapWith` encontrado na biblioteca failure.

Veja também

- Biblioteca failure
- Extensões do protótipo Error

Chunk

Chunks são apenas blocos binários de dados.

Declaração

Chunks não podem ser declarados explicitamente; você deve usar o construtor de tipo e aplicar uma das seguintes estratégias:

```
_ <- fat.type.Chunk

Chunk(null)            # Void -> (chunk vazio)
Chunk(true)            # Boolean -> '\001'
Chunk(65)              # Number -> 'A'
Chunk('ABC')           # Text -> 'ABC'
Chunk([ 65, 66, 67 ])  # List/Number -> 'ABC'
```

> espera-se que os números sejam valores de byte válidos (0-255), caso contrário, um erro será gerado

Manipulando Chunks

Concatenação

No FatScript, você pode concatenar, ou juntar, dois chunks usando o operador `+`. Por exemplo:

```
abCombinados = chunkA + chunkB
```

Seleção de blocos

A seleção permite acesso a partes específicas de um chunk usando índices. O FatScript suporta índices positivos e negativos. Índices positivos começam a partir do início do chunk (com `0` como o primeiro byte), enquanto índices negativos começam a partir do final (`-1` é o último byte).

> para uma explicação detalhada sobre o sistema de indexação no FatScript, consulte a seção sobre acesso e seleção de itens em Lista

Selecionar com um índice recupera um único byte do chunk (como número). Usando um intervalo de bytes, seleciona um fragment incluindo os índices de início e fim, exceto quando se usa o operador de intervalo semiaberto `..<`, que é exclusivo no lado direito.

Acessar índices fora do intervalo válido gerará um erro para seleções individuais. Para seleções de intervalo, índices fora dos limites resultam em um chunk vazio.

```
x3 = Chunk('example')
x3(1)     # 120 (valor ASCII de 'x')
x3(..2)   # novo Chunk contendo 3 bytes (correspondendo a 'exa')
```

Comparações

São suportadas comparações de igualdade `==` e desigualdade `!=` de chunks.

Veja também

- Extensões do protótipo Chunk

Controle de fluxo

Avance em um fluxo contínuo de decisões que devem ser tomadas.

Fallback

As operações padrão ou de coalescência nula são definidas com dois pontos de interrogação `??` e funcionam da seguinte maneira:

```
<expressãoTalvezNulaOuFalha> ?? <valorFallback>
```

Caso o lado esquerdo não seja `null` nem `Error`, então ele é usado; caso contrário, o valor de fallback é retornado.

de forma semelhante, você pode usar o operador de atribuição por coalescência nula `??=`

If

Declarações `If` são definidas com um ponto de interrogação `?`, como abaixo:

```
<condição> ? <resposta>
```

como não há alternativa, `null` é retornado se a condição não for atendida

If-Else

Declarações `If-Else` são definidas com um ponto de interrogação `?` seguido de dois pontos `:`, como abaixo:

```
<condição> ? <resposta> : <alternativa>
```

Para usar declarações `If-Else` multilinhas, envolva a resposta em chaves `{...}` assim:

```
<condição> ? {
  <resposta>
} : {
  <alternativa>
}
```

Cases

Cases são definidos com a seta espessa `=>` e são automaticamente encadeados, criando uma sintaxe intuitiva e simplificada, semelhante a uma declaração switch, sem a possibilidade de queda. Isso permite que condições não relacionadas sejam misturadas, resultando em uma estrutura if-else-if-else mais concisa:

```
<condição1> => <respostaPara1>
<condição2> => <respostaPara2>
<condição3> => <respostaPara3>
...
```

Exemplo:

```
escolha = (x) -> {
  x == 1 => 'a'
  x == 2 => 'b'
  x == 3 => 'c'
}

escolha(2)  # 'b'
escolha(8)  # null
```

Para fornecer um valor padrão para seu método, você pode adicionar um caso pega-tudo usando um sublinhado _ no final da sequência:

```
escolha = (x) -> {
  x == 1 => 'a'
  x == 2 => 'b'
  x == 3 => 'c'
```

```
  _      => 'd'
}

escolha(2)  # 'b'
escolha(8)  # 'd'
```

Para cenários mais complexos, você pode usar blocos como resultados para cada caso:

```
...
  condição => {
    # faça algo
    'foo'
  }
  _ => {
    # faça outra coisa
    'bar'
  }
...
```

Cases devem terminar em um caso pega-tudo _ ou final do bloco. O uso mais efetivo de Cases é dentro de métodos na parte inferior do corpo do método.

Embora seja possível adicionar Cases aninhados, é melhor evitar construções excessivamente complexas. Isso torna o código mais difícil de seguir e provavelmente perde o objetivo de usar esse recurso.

Pode ser mais apropriado extrair essa lógica para um método separado. O FatScript incentiva os desenvolvedores a dividir a lógica em métodos distintos, ajudando a evitar código spaghetti.

Switch

O operador Switch é denotado pelo símbolo >>, que guia o fluxo de controle baseado na correspondência do valor com uma série de casos:

Sintaxe:

```
<valor> >> {
  <valorCaso1> => <respostaPara1>
  <valorCaso2> => <respostaPara2>
  ...
  _ => <respostaPadrão>
}
```

Cada caso no bloco Switch é avaliado em ordem até que uma correspondência seja encontrada e o resultado do caso correspondente é retornado:

```
escolher = -> _ >> {
  1 => 'um'
  2 => 'dois'
  3 => 'três'
  _ => 'outro'
}

escolher(2)  # 'dois'
escolher(4)  # 'outro'
```

Os casos do Switch também podem envolver expressões, permitindo correspondências dinâmicas:

```
avaliar = (x, y) -> x >> {
  y + 1 => 'logo acima de y'
  y - 1 => 'logo abaixo de y'
  _     => 'não diretamente ao redor de y'
}

avaliar(5, 4)  # 'logo acima de y'
avaliar(3, 4)  # 'logo abaixo de y'
avaliar(7, 4)  # 'não diretamente ao redor de y'
```

Loops

Repetir, repetir, repetir, repetir, repetir...

Sintaxe base

Todos os loops são criados com o sinal de arroba `@`, por exemplo:

```
<expressão> @ <corpoDoLoop>
```

Loop tipo "enquanto"

O corpo do loop irá executar enquanto a expressão avaliar para:

- verdadeiro
- número não zero
- texto não vazio

A execução irá terminar quando a expressão avaliar para:

- falso
- nulo
- número zero
- texto vazio
- erro

Por exemplo, este loop imprime números de 0 a 3:

```
_ <- fat.console

~ i = 0

(i < 4) @ {
  log(i)
  i += 1
}
```

Sintaxe de mapeamento

Você pode mapear intervalos, listas e escopos com um mapeador, assim:

```
<intervalo|coleção> @ <mapeador>
```

Uma nova lista é gerada com base nos valores de retorno do mapeador.

Mapeando um intervalo

Utilizando o operador de intervalo `..` o mapeador receberá um número como entrada sequencialmente do limite esquerdo até o limite direito:

```
4..0 @ num -> num + 1  # retorna [ 5, 4, 3, 2, 1 ]
```

> a sintaxe de intervalo é inclusiva em ambos os lados, por exemplo, 0..2 retorna 0, 1, 2.

Há também o operador de intervalo semiaberto `..<`, exclusivo no lado direito.

> ressalva: o intervalo semiaberto não funciona com direção inversa, sempre precisa ser do mínimo para máximo

Mapeando uma lista

O mapeador receberá os itens em ordem (da esquerda para a direita):

```
[ 3, 1, 2 ] @ item -> item + 1  # retorna [ 4, 2, 3 ]
```

Mapeando um escopo

O mapeador receberá os nomes (chaves) das entradas armazenadas no escopo em ordem alfabética:

```
{ c = 3, a = 1, b = 2 } @ chave -> chave  # retorna [ 'a', 'b', 'c' ]
```

> nos exemplos, usamos literais de lista e escopo, mas uma entrada ou chamada que avalia para uma lista ou um escopo terá o mesmo efeito

Você pode acessar as entradas de um escopo referindo-se a ele pelo nome, mas neste caso precisa que ele esteja definido no escopo externo, por exemplo:

```
meuEscopo = { c = 3, a = 1, b = 2 }
meuEscopo @ chave -> meuEscopo(chave)  # retorna [ 1, 2, 3 ]
```

O FatScript utiliza um recurso de caching inteligente que faz com que esta sintaxe não gere um esforço adicional para buscar o elemento da vez no escopo durante o mapeamento.

Bibliotecas

Vamos falar sobre os doces recheios embutidos no FatScript: as bibliotecas!

Bibliotecas padrão

Essenciais

Estas são as bibliotecas fundamentais que você espera que estejam disponíveis em uma linguagem de programação, fornecendo funcionalidades essenciais:

- async - Trabalhadores e tarefas assíncronas
- color - Códigos de cores ANSI para console
- console - Operações de entrada e saída do console
- curses - Interface de usuário baseada em terminal
- enigma - Métodos de criptografia, hash e UUID
- failure - Tratamento de erros e gerenciamento de exceções
- file - Operações de entrada e saída de arquivos
- http - Framework de manipulação HTTP
- math - Operações e funções matemáticas
- recode - Conversão de dados entre vários formatos
- sdk - Utilitários do kit de desenvolvimento de software do fry
- system - Operações e informações no nível do sistema
- time - Manipulação de data e hora

Pacote de tipos

Este pacote estende os recursos dos tipos nativos do FatScript:

- Void
- Boolean
- Number
- HugeInt
- Text
- Method
- List
- Scope
- Error
- Chunk

Pacote Extra

Tipos adicionais implementados em FatScript puro:

- Date - Gerenciamento de calendário e datas
- Duration - Construtor de duração em milissegundos
- HashMap - Armazenamento rápido de chave-valor
- Logger - Suporte ao registro de logs
- Memo - Utilitário de memoização genérica
- Option - Encapsulamento de valor opcional
- Param - Verificação de presença e tipo de parâmetro
- Sound - Interface de reprodução de som
- Storable - Armazenamento de dados

Atalho de importação

Se você deseja torná-los todos disponíveis de uma vez, pode simplesmente fazer o seguinte, e todas essas coisas boas estarão disponíveis para o seu código:

```
_ <- fat._
```

Embora esse recurso possa ser conveniente ao experimentar no REPL, esteja ciente de que ele traz todas as constantes e nomes de método da biblioteca, potencialmente poluindo seu namespace global.

fat.std

Alternativamente, importe a biblioteca "standard" que importa todos os tipos (inclusive do pacote extra), bem como aplica importações nomeadas de todos os demais pacotes, assim:

```
_ <- fat.std
```

O que equivale a:

```
_       <- fat.type._
_       <- fat.extra._
async   <- fat.async
color   <- fat.color
console <- fat.console
curses  <- fat.curses
enigma  <- fat.enigma
failure <- fat.failure
http    <- fat.http
file    <- fat.file
math    <- fat.math
recode  <- fat.recode
sdk     <- fat.sdk
system  <- fat.system
time    <- fat.time
```

Observe que, a importação de tudo antecipadamente pode adicionar uma sobrecarga desnecessária ao tempo de inicialização do seu programa, mesmo que você precise usar apenas alguns métodos.

Como boa prática, considere importar apenas os módulos específicos de que você precisa, com importações nomeadas. Dessa forma, você pode manter seu código limpo e conciso, minimizando o risco de conflitos de nome ou problemas de desempenho.

Hacking e mais

Sob o capô, as bibliotecas são construídas usando comandos embutidos. Para obter uma compreensão mais profunda e explorar o funcionamento interno do interpretador, mergulhe neste tópico mais avançado.

async

Trabalhadores e tarefas assíncronas

Importação

```
_ <- fat.async
```

Tipos

A biblioteca `async` introduz o tipo `Worker`.

Worker

O `Worker` é um simples invólucro para uma operação assíncrona.

Construtor

Nome	Assinatura	Breve descrição
Worker	(task: Method)	Cria um Worker em modo espera

O construtor `Worker` aceita dois argumentos:

- **task**: O método a ser executado de forma assíncrona (o método pode não aceitar argumentos diretamente, mas você pode adicionar esses usando duas setas na definição `-> ->`).
- **wait** (opcional): O tempo limite em milissegundos. Se a tarefa não terminar dentro desse tempo, ela será cancelada.

Membros do protótipo

Nome	Assinatura	Breve descrição
start	(): Worker	Inicia a tarefa
cancel	(): Void	Cancela a tarefa
await	(): Worker	Aguarda pela conclusão da tarefa
isDone	(): Boolean	Verifica se a tarefa foi concluída
hasStarted	Boolean	Definido pelo método start
hasAwaited	Boolean	Definido pelo método await
isCanceled	Boolean	Definido pelo método cancel
result	Any	Definido pelo método await

Métodos avulso

Nome	Assinatura	Breve descrição
atomic	(op: Method): Any	Executa a operação atomicamente
selfCancel	(): *	Encerra a execução da thread
processors	(): Number	Retorna o número de processadores

Notas de uso

As instâncias de `Worker` são mapeadas para threads do sistema em uma base um-para-um e são executadas conforme o agendamento do sistema. Isso implica que sua execução pode nem sempre ser imediata. Para aguardar o resultado de um `Worker`, use o método `await`.

Diferentemente de outros contextos, no código assíncrono, a `task: Method` executa sem acesso ao escopo no qual é criada. Ela só pode acessar propriedades que foram 'curryficadas' `-> ->` para dentro do seu escopo de execução ou aquelas que estão diretamente acessíveis no escopo global.

> para manter o máximo desempenho, evite usar interpolação de texto dentro de tarefas assíncronas

Exemplos

```
async <- fat.async
math  <- fat.math
time  <- fat.time

# Define uma tarefa lenta
slowTask = (seconds: Number): Text -> -> {
  time.wait(seconds * 1000)
  'done'
}

# Inicia a tarefa como Worker
worker = Worker(slowTask(5)).start

# Obtém o resultado do worker
result1 = worker.await.result  # bloqueia até a tarefa ser concluída

# Inicia uma tarefa com timeout
task = Worker(slowTask(5), 3000).start  # a tarefa deve expirar

# Obtém o resultado da tarefa
result2 = task.await.result  # bloqueia até a tarefa ser concluída ou ocorrer timeout
```

> o método `await` da gera um `AsyncError` se a tarefa exceder o tempo antes da conclusão

atomic

O encapsulador `atomic` é uma ferramenta crítica para garantir a segurança das threads e a integridade dos dados na programação concorrente. Quando múltiplos workers ou tarefas assíncronas acessam e modificam recursos compartilhados, condições de corrida podem ocorrer, levando a resultados imprevisíveis e errôneos. A operação `atomic` aborda essa questão garantindo que o método que ela envolve seja executado de forma atômica. Isso significa que a operação inteira é completada como uma única unidade indivisível, sem possibilidade de outras threads intervirem no meio do caminho para a mesma operação. Isso é particularmente importante para operações como incrementar um contador, atualizar estruturas de dados compartilhadas ou arquivos, ou realizar qualquer ação onde a ordem de execução importa:

```
async.atomic(-> file.append(logFile, line))
```

Embora as operações `atomic` sejam uma ferramenta poderosa para garantir a consistência, é importante estar atento ao potencial de contenção que elas introduzem. A contenção ocorre quando múltiplas "threads" ou tarefas tentam executar uma operação simultaneamente, levando a potenciais gargalos de desempenho à medida que cada tarefa espera sua vez. O uso excessivo ou desnecessário de operações `atomic` pode degradar significativamente o desempenho de sua aplicação, reduzindo a concorrência. Mantenha apenas a seção crítica de código que absolutamente requer atomicidade envolvida como uma operação `atomic`.

> por baixo dos panos, operações atômicas são fundamentalmente protegidas por um único `mutex` global

Async na Web Build

Ao usar `fry` construído com Emscripten (por exemplo, ao usar FatScript Playground), o suporte limitado da plataforma para multithreading afeta a implementação do `Worker`. Para maximizar a compatibilidade do código entre plataformas, as tarefas do `Worker` são executadas inline e bloqueiam a thread principal quando o método `start` é chamado. Esta abordagem compromete as vantagens da execução assíncrona, mas permite uma implementação consistente entre plataformas em muitos dos cenários.

Veja também

- Biblioteca Time

color

Códigos de cores ANSI para console

Importação

```
_ <- fat.color
```

Constantes

- black, 0
- red, 1
- green, 2
- yellow, 3
- blue, 4
- magenta, 5
- cyan, 6
- white, 7
- bright.black, 8
- bright.red, 9
- bright.green, 10
- bright.yellow, 11
- bright.blue, 12
- bright.magenta, 13
- bright.cyan, 14
- bright.white, 15

Métodos

Nome	Assinatura	Breve descrição
detectDepth	(): Number	Obter suporte de cor do console
to8	(xr: Any, g: Number = ø, b: Number = ø)	Converter RGB para 8-cores
to16	(xr: Any, g: Number = ø, b: Number = ø)	Converter RGB para 16-cores
to256	(xr: Any, g: Number = ø, b: Number = ø)	Converter RGB para 256-cores

Notas de Uso

to8, to16 e to256

O parâmetro `xr` pode ser um texto opcional que representa a cor no formato HTML. Por exemplo, pode ser fornecido como 'fae830' ou '#fae830' (amarelo):

```
color   <- fat.console
console <- fat.console

console.log('hey', color.to16('fae830'))
console.log('hey', color.to256('fae830'))
```

No entanto, se `xr` for um número entre 0 e 255 representando `r`, então os parâmetros `g` e `b` serão necessários:

```
console.log('hey', color.to256(250, 232, 48))  // mesmo resultado
```

> estes métodos podem produzir aproximações da cor original nas profundidades 8, 16 ou 256 e não a côr verdadeira exata

Veja também

- Biblioteca console
- Biblioteca curses
- 256 Cores

console

Operações de entrada e saída do console

Importação

```
_ <- fat.console
```

Métodos

Nome	Assinatura	Breve descrição
log	(msg: Any, fg: Number = ø, bg: Number = ø): Void	Imprime msg para stdout, com quebra de linha
print	(msg: Any, fg: Number = ø, bg: Number = ø): Void	Imprime msg para stdout, sem quebra de linha
stderr	(msg: Any, fg: Number = ø, bg: Number = ø): Void	Imprime msg para stderr, com quebra de linha
input	(msg: Any, mode: Text = ø): Text	Imprime msg e retornar entrada de stdin
flush	(): Void	Esvazia buffer de saída padrão
cls	(): Void	Limpa stdout usando códigos de escape ANSI
moveTo	(x: Number, y: Number): Void	Move o cursor usando códigos de escape ANSI
isTTY	(): Boolean	Verifica se stdout é um terminal
isTty	(): Boolean	DESCONTINUADA (será removida em `3.x.x`)
showProgress	(label: Text, fraction: Number): Void	Renderiza barra de progresso, fração 0 a 1

os métodos `log`, `stderr` e `input` garantem segurança de threads em cenários assíncronos

Notas de uso

saída

Por padrão, `stdout` e `stderr` imprimem no console. Os parâmetros de cor de primeiro plano (fg) e cor de plano de fundo (bg) são opcionais.

as cores são automaticamente suprimidas se o buffer de saída não for um TTY

input

O parâmetro opcional `mode` aceita os seguintes valores:

- 'plain', entrada simples (sem cursor readline, sem histórico)
- 'quiet', como modo plain, porém sem feedback
- 'secret', modo especial para leitura de senha
- `null` (padrão), com readline e histórico de entrada

Veja também

- Biblioteca color
- Biblioteca curses

curses

Interface de usuário baseada em terminal

> embora a inspiração seja reconhecida, o FatScript tem seu próprio modo de abordar a interface de usuário no terminal, que difere em muitos aspectos da biblioteca curses original

Importação

```
_ <- fat.curses
```

Methods

Nome	Assinatura	Breve descrição
box	(p1: Scope, p2: Scope): Void	Desenhar quadrado de pos1 a pos2
fill	(p1: Scope, p2: Scope, p: Text = ' '): Void	Preencher de pos1 a pos2 com p
clear	(): Void	Limpar buffer de tela
refresh	(): Void	Renderizar buffer de tela
getMax	(): Scope	Retorna tamanho da tela como x, y
printAt	(pos: Scope, msg: Any, width: Number = ø): Void	Imprimir msg em { x, y } pos
makePair	(fg: Number = ø, bg: Number = ø): Number	Criar um par de cores
usePair	(pair: Number): Void	Aplicar par de cores
frameTo	(cols: Number, rows: Number)	Alinhar área ao centro da tela
readKey	(): Text	Retorna tecla pressionada
readText	(pos: Scope, width: Number, prev: Text = ø): Text	Inicia caixa de texto
flushKeys	(): Void	Limpar buffer de entrada
endCurses	(): Void	Encerrar o modo curses

> as posições (pos) estão no formato { x: Number, y: Number }

> os métodos nesta biblioteca **não garantem** a segurança de threads em cenários assíncronos, ou utilize a thread principal **ou então** um único worker para renderizar atualizações no console

Notas de uso

Qualquer método desta biblioteca, exceto `getMax` e `endCurses`, iniciará o modo curses se ainda não tiver iniciado. Note que métodos como `log`, `stderr` e `input` da biblioteca console chamarão `endCurses` implicitamente. No entanto, `moveTo`, `print` e `flush` não irão alterar o modo de saída e podem ser combinados com métodos curses, o que pode ser útil em algumas circunstâncias.

As letras `x` e `y` representam coluna e linha, respectivamente, ao chamar `printAt`, onde (0, 0) é o canto superior esquerdo e o resultado de `getMax` é apenas a primeira coordenada fora do canto inferior direito.

> caracteres especiais em curses só funcionam se um locale UTF-8 puder ser definido

makePair

Você pode importar a biblioteca color para usar nomes de cores e criar uma combinação de primeiro plano e plano de fundo (par). Passe `null` para aplicar a cor padrão no parâmetro desejado.

usePair

A entrada deste método deve ser um par de cores criado com o método `makePair`. Ele deixa este par habilitado até que você chame-o novamente com um par diferente.

readKey

Este método não bloqueia e retorna `null` se `stdin` estiver vazio, caso contrário retornará um caractere por vez.

Chaves especiais podem ser detectadas e retornar palavras-chave como:

- teclas de seta:
  - up
  - down
  - left
  - right
- teclas de edição:
  - delete
  - backspace
  - enter
  - space
  - tab
  - backTab (shift+tab)
- teclas de controle:
  - pageUp
  - pageDown
  - home
  - end
  - insert
  - esc
- outras:
  - resize (janela do terminal foi redimensionada)

a detecção correta das teclas pode depender do contexto ou da plataforma

readText

Entra em modo captura de texto utilizando uma área demarcada por posição e largura da caixa de texto. Se o texto for maior que o espaço uma rolagem automática do texto é realizada. O texto completo é retornado ao pressionar `enter` ou `tab`, no entanto caso `esc` seja pressionado é retornado `null`.

Veja também

- Biblioteca color
- Biblioteca console

enigma

Métodos de criptografia, hash e UUID

Importação

```
_ <- fat.enigma
```

Métodos

Nome	Assinatura	Breve descrição
getHash	(msg: Text): Number	Obter hash de 32-bits do texto
genUUID	(): Text	Gerar um UUID (versão 4)
genKey	(len: Number): Text	Gerar chave aleatória
derive	(secret: Text): Text	Função de derivação de chave
encrypt	(msg: Text, key: Text = ''): Text	Encriptar msg usando key
decrypt	(msg: Text, key: Text = ''): Text	Decriptar msg usando key

`derive` é determinística e usa o alfabeto Base64 para uma saída de 32 caracteres

Notas de uso

Você pode omitir ou passar uma chave (key) em branco `''` para usar a chave padrão.

Atenção!

Embora `enigma` torne o texto encriptado "não legível por humanos", esse esquema não é criptograficamente seguro! NÃO o utilize sozinho para proteger dados!

Se pareado com uma chave personalizada que não esteja armazenada junto com a mensagem, pode oferecer alguma proteção de dados.

Conformidade do método UUID

Um UUID, ou Universally Unique Identifier, é um número de 128 bits usado para identificar objetos ou entidades em sistemas de computador. A implementação fornecida gera UUIDs aleatórios como texto que segue o formato da versão 4 da especificação RFC 4122, mas não adere estritamente à aleatoriedade criptograficamente segura necessária. Na prática, o risco de colisão tem uma probabilidade extremamente baixa e é muito improvável de ocorrer, e para a maioria das aplicações pode ser considerado bom o bastante.

failure

Tratamento de erros e gerenciamento de exceções

Importação

```
_ <- fat.failure
```

Métodos

Nome	Assinatura	Breve descrição
trap	(): Void	Aplicar manipulador genérico de erro genérico
trapWith	(handler: Method): Void	Definir um manipulador para erros no contexto
untrap	(): Void	Desarmar o manipulador para erros no contexto

Notas de uso

Quando um erro é criado se um manipulador de erro for encontrado, buscando do contexto de execução interno para externo, o manipulador que envolve a falha é invocado automaticamente com esse erro como argumento e o contexto de chamada é encerrado com o valor de retorno do manipulador de erro.

não é possível definir um manipulador para o escopo global

trapWith

Este método vincula um manipulador de erros ao contexto do site de chamada, por exemplo quando usado dentro de um método, ele apenas protegerá a lógica executada dentro do corpo desse método.

Exemplo

Defina um manipulador de erro que imprima o erro e saia:

```
console <- fat.console
system  <- fat.system
sdk     <- fat.sdk

simpleErrorHandler = (error) -> {
  console.log(error)
  sdk.printStack(10)
  system.exit(system.failureCode)
}
```

Finalmente, use o método `trapWith` para atribuir o manipulador de erro:

```
failure <- fat.failure
failure.trapWith(simpleErrorHandler)
```

Trap it!

Você pode lidar com erros esperados ou deixar passar o inesperado:

```
failure <- fat.failure
_       <- fat.type.Error

MyError = Error

errorHandler = -> _ >> {
  MyError => 0  # handle (expected)
  _       => _  # pass through (unexpected)
}

unsafeMethod = (n) -> {
  failure.trapWith(errorHandler)
```

```
  n < 10 ? MyError('arg is less than ten')
  n - 10
}
```

Neste caso, o programa não travará se você chamar `unsafeMethod(5)`, mas se você comentar a linha `trapWith`, verá que ele trava com `MyError`.

Veja também

- Error (sintaxe)
- Extensões do protótipo Error
- Controle de fluxo

file

Operações de entrada e saída de arquivo

Importação

```
_ <- fat.file
```

Contribuições de Tipo

Nome	Assinatura	Breve descrição
FileInfo	(modTime: Epoch, size: Text)	Metadados do arquivo

Métodos

Nome	Assinatura	Breve descrição
basePath	(): Text	Extrair caminho onde o app foi chamado
exists	(path: Text): Boolean	Verificar se existe arquivo no caminho
read	(path: Text): Text	Ler arquivo do caminho (modo de texto)
readBin	(path: Text): Chunk	Ler arquivo do caminho (modo de binário)
write	(path: Text, src): Boolean	Escrever src no arquivo e retornar sucesso
append	(path: Text, src): Boolean	Acrescentar ao arquivo e retornar sucesso
remove	(path: Text): Boolean	Apagar o arquivo e retornar sucesso
isDir	(path: Text): Boolean	Verificar se o caminho é um diretório
mkDir	(path: Text, safe: Boolean)	Criar um diretório a directory
lsDir	(path: Text): List	Obter lista de arquivos em um diretório
stat	(path: Text): FileInfo	Obter metadados do arquivo

Notas de uso

read

Na exceção:

- registra o erro no `stderr`
- retorna `null`

`read` não pode ver "arquivos" embutidos, mas `readLib` da biblioteca SDK pode

write/append

Exceções:

- registra o erro no `stderr`
- retorna `false`

mkDir

Se `safe` estiver definido como `true`, o diretório receberá permissão 0700 em vez do padrão 0755, o que é menos protegido.

Veja também

- Biblioteca recode

http

Framework de manipulação HTTP

Importação

```
_ <- fat.http
```

Route

Uma rota é uma estrutura usada para mapear métodos HTTP para certos padrões de caminho, especificando qual código deve ser executado quando uma requisição é recebida. Cada rota pode definir um comportamento diferente para cada método HTTP (`POST`, `GET`, `PUT`, `DELETE`).

Construtor

Nome	Assinatura	Breve descrição
Route	(path: Text, post: Method, get: Method, put: Method, delete: Method)	Constrói um objeto Route

cada método implementado recebe um HttpRequest como argumento e deve retornar um objeto HttpResponse

HttpRequest

Um `HttpRequest` representa uma mensagem de requisição HTTP. Isso é o que seu servidor recebe de um cliente quando ele faz uma requisição ao seu servidor.

Construtor

Nome	Assinatura	Breve descrição
HttpRequest	(method: Text, path: Text, params: Scope, headers: List/Text, body: Any)	Constrói um objeto HttpRequest

HttpResponse

Um `HttpResponse` representa uma mensagem de resposta HTTP. Isso é o que um servidor envia de volta ao cliente em resposta a uma requisição HTTP.

Construtor

Nome	Assinatura	Breve descrição
HttpResponse	(status: Number, headers: List/Text, body: Any)	Constrói um objeto HttpResponse

Métodos

Nome	Assinatura	Breve descrição
setHeaders	(headers: List): Void	Definir cabeçalhos de solicitações
post	(url: Text, body, wait): HttpResponse	Criar/postar body para url
get	(url: Text, wait): HttpResponse	Ler/obter de url
put	(url: Text, body, wait): HttpResponse	Atualizar/colocar body na url
delete	(url: Text, wait): HttpResponse	Excluir em url
setName	(name: Text): Void	Definir agente/nome do servidor
verifySSL	(enabled: Boolean): Void	Configuração SSL (modo cliente)
setSSL	(certPath: Text, keyPath: Text): Void	Configuração SSL (modo servidor)
listen	(port: Number, routes: List/Route)	Provedor de endpoint (modo servidor)

`body: Any` e `wait: Number` são sempre parâmetros opcionais, sendo que caso `body` não se enquadre como `Text` ou `Chunk` será automaticamente convertido para JSON no processo de envio e `wait` é o tempo máximo de espera e o padrão é 30.000ms (30 segundos)

`verifySSL` está habilitado por padrão para o modo cliente

`setSSL` pode não estar disponível, caso o sistema não tenha a biblioteca OpenSSL

Notas de uso

Modo cliente

Em `HttpResponse.body`, você pode precisar decodificar explicitamente uma resposta JSON para `Scope` usando o método `fromJSON`. Para postar um tipo nativo como JSON, você pode codificá-lo usando o método `toJSON`; no entanto, isso não é estritamente necessário, pois será feito implicitamente. Ambos os métodos estão disponíveis na biblioteca fat.recode.

Se os cabeçalhos não forem definidos, o cabeçalho `Content-Type` padrão para `Chunk` será `application/octet-stream`, para `Text` será `text/plain; charset=UTF-8` e para outros tipos, será `application/json; charset=UTF-8` (devido à conversão implícita).

Você pode definir cabeçalhos de solicitação personalizados da seguinte forma:

```
http <- fat.http

url = ...
token = ...
body = ...

http.setHeaders([
  "Accept: application/json; charset=UTF-8"
  "Content-Type: application/json; charset=UTF-8"
  "Authorization: Bearer " + token  # cabeçalhos personalizado
])

http.post(url, body)
```

definir cabeçalhos substituirá completamente a lista anterior pela nova lista

Ao realizar solicitações assíncronas, você pode precisar chamar `setHeaders`, `setName` e configurar `verifySSL` dentro de cada Worker, já que essas configurações são locais para cada thread.

Modo servidor

Lidando com respostas HTTP

O servidor FatScript lida automaticamente com os códigos de status HTTP comuns, como 200, 400, 404, 405, 500 e 501. Sendo 200 o padrão ao construir um objeto `HttpResponse`.

Além dos códigos de status retornados de forma automática, você também pode retornar explicitamente estes e outros códigos de status, como 201, 202, 203, 204, 205, 301, 401 e 403, especificando o código de status no objeto `HttpResponse`, por exemplo: `HttpResponse(status = 401)`. Em todos os casos, quando aplicável, o servidor fornece corpos de resposta padrão em texto simples. No entanto, você tem a opção de substituir esses valores padrão e fornecer seus próprios corpos de resposta personalizados, quando necessário.

Ao lidar automaticamente com esses códigos de status e fornecer corpos de resposta padrão, o servidor FatScript simplifica o processo de desenvolvimento, ao mesmo tempo em que permite que você tenha controle sobre o conteúdo da resposta quando necessário.

não pertencendo a nenhum dos códigos anteriores, o servidor irá retornar o código 500

Veja um exemplo de um simples servidor HTTP de arquivos:

```
_    <- fat.type.Text
file <- fat.file
http <- fat.http
{ Route, HttpRequest, HttpResponse } = http

# adapte para o local do conteúdo
basePath = '/home/user/contentFolder'

# restrito a algumas extensões somente
getContentType = (path: Text): Text -> {
```

```
  ext2 = path(-3..).toLower
  ext3 = path(-4..).toLower
  ext4 = path(-5..).toLower

  ext4 == '.html' => 'Content-Type: text/html'
  ext3 == '.htm'  => 'Content-Type: text/html'
  ext2 == '.js'   => 'Content-Type: application/javascript'
  ext4 == '.json' => 'Content-Type: application/json'
  ext3 == '.css'  => 'Content-Type: text/css'
  ext2 == '.md'   => 'Content-Type: text/markdown'
  ext3 == '.xml'  => 'Content-Type: application/xml'
  ext3 == '.csv'  => 'Content-Type: text/csv'
  ext3 == '.txt'  => 'Content-Type: text/plain'
  ext4 == '.svg'  => 'Content-Type: image/svg+xml'
  ext3 == '.rss'  => 'Content-Type: application/rss+xml'
  ext4 == '.atom' => 'Content-Type: application/atom+xml'
  ext3 == '.png'  => 'Content-Type: image/png'
  ext3 == '.jpg'  => 'Content-Type: image/jpeg'
  ext4 == '.jpeg' => 'Content-Type: image/jpeg'
  ext3 == '.gif'  => 'Content-Type: image/gif'
  ext3 == '.ico'  => 'Content-Type: image/icon'
}

routes: List/Route = [
  Route(
    '*'
    get = (request: HttpRequest): HttpResponse -> {
      path = basePath + request.path
      type = getContentType(path)

      !type             => HttpResponse(status = 403)  # forbidden
      file.exists(path) => HttpResponse(body = file.readBin(path), headers = [ type
])
      _                 => HttpResponse(status = 404)  # not found
    }
  )
]

http.listen(8080, routes)
```

em uma aplicação real, `request.path` deve ser sanitizado antes de ser utilizado para acessar arquivos no servidor; aqui, é utilizado diretamente apenas como exemplo

math

Operações e funções matemáticas

Importação

```
_ <- fat.math
```

Constantes

- e, logaritmo natural constante 2.71...
- maxInt, 9007199254740992
- minInt, -9007199254740992
- pi, razão do círculo para o seu diâmetro 3.14...

leia mais sobre precisão numérica no FatScript

Funções básicas

Nome	Assinatura	Breve descrição
abs	(x: Number): Number	Retorna o valor absoluto de x
ceil	(x: Number): Number	Retorna o menor inteiro >= x
floor	(x: Number): Number	Retorna o maior inteiro <= x
isInf	(x: Number): Boolean	Retorna true se x for infinito
isNaN	(x: Any): Boolean	Retorna true se x não for um número
logN	(x: Number, base: Number = e): Number	Retorna o logaritmo
random	(): Number	Retorna pseudo-aleatório, onde 0 <= n < 1
sqrt	(x: Number): Number	Retorna a raiz quadrada de x
round	(x: Number): Number	Retorna o inteiro mais próximo de x

Funções trigonométricas

Nome	Assinatura	Breve descrição
sin	(x: Number): Number	Retorna o seno de x
cos	(x: Number): Number	Retorna o cosseno de x
tan	(x: Number): Number	Retorna a tangente de x
asin	(x: Number): Number	Retorna o arco seno de x
acos	(x: Number): Number	Retorna o arco cosseno de x
atan	(x: Number, y = 1): Number	Retorna o arco tangente de x, y
radToDeg	(r: Number): Number	Converte radianos para graus
degToRad	(d: Number): Number	Converte graus para radianos

Funções hiperbólicas

Nome	Assinatura	Breve descrição
sinh	(x: Number): Number	Retorna o seno hiperbólico de x
cosh	(x: Number): Number	Retorna o cosseno hiperbólico de x
tanh	(x: Number): Number	Retorna a tangente hiperbólica de x

Funções estatísticas

Nome	Assinatura	Breve descrição
mean	(v: List/Number): Number	Retorna a média de um vetor
median	(v: List/Number): Number	Retorna a mediana de um vetor

Nome	Assinatura	Breve descrição
sigma	(v: List/Number): Number	Retorna o desvio padrão de um vetor
variance	(v: List/Number): Number	Retorna a variância de um vetor
max	(v: List/Number): Number	Retorna o valor máximo no vetor
min	(v: List/Number): Number	Retorna o valor mínimo no vetor
sum	(v: List/Number): Number	Retorna a soma do vetor

Outras funções

Nome	Assinatura	Breve descrição
fact	(x: Number): Number	Retorna o fatorial de x
exp	(x: Number): Number	Retorna e elevado à potência de x
sigmoid	(x: Number): Number	Retorna o sigmoid de x
relu	(x: Number): Number	Retorna o ReLU de x

Exemplo

```
math <- fat.math  # importação nomeada
math.abs(-52)     # retorna 52
```

Veja também

- Number (sintaxe)
- Extensões do protótipo Number

recode

Conversão de dados entre vários formatos

Importação

```
_ <- fat.recode
```

pacote type é automaticamente importado com esta importação

Constantes

- numeric, definição regex usado por `inferType` (DESCONTINUADA)

a constante `numeric` é redundante e será removida na versão `3.x.x`

Variáveis

Estas configurações podem ser ajustadas para configurar o comportamento das funções de processamento:

- csvSeparator, o padrão é `,` (vírgula)
- csvReplacement, o padrão é vazio (apenas remove vírgulas do texto)
- xmlWarnings, o padrão é `true` - defina como `false` para suprimir avisos XML (DESCONTINUADA)

Funções Base64

Nome	Assinatura	Breve descrição
toBase64	(data: Chunk): Text	Codifica bloco binário para texto base64
fromBase64	(b64: Text): Chunk	Decodifica texto base64 para formato original

Funções JSON

Nome	Assinatura	Breve descrição
toJSON	(_: Any): Text	Codifica JSON a partir de tipos nativos
fromJSON	(json: Text): Any	Decodifica JSON para tipos nativos

Funções URL

Nome	Assinatura	Breve descrição
toURL	(text: Text): Text	Codifica texto para texto escapado URL
fromURL	(url: Text): Text	Decodifica texto escapado URL para formato original
toFormData	(data: Scope): Text	Codifica dados de formulário URL a partir de escopo
fromFormData	(data: Text): Scope	Decodifica dados de formulário URL para escopo

Funções CSV

Nome	Assinatura	Breve descrição
toCSV	(header: List/Text, rows: List/Scope): Text	Codifica CSV a partir de linhas
fromCSV	(csv: Text): List/Scope	Decodifica CSV para linhas

`csvReplacement` é usado por `toCSV` como substituição em caso de um `csvSeparator` ser encontrado dentro de um texto sendo codificado

Funções XML (DESCONTINUADAS)

Atributos XML e tags auto-fechadas não são suportados.

Nome	Assinatura	Breve descrição
toXML	(node: Any): Text	Codifica XML a partir de tipos nativos
fromXML	(text: Text): Any	Decodifica XML para tipos nativos

As funções XML serão removidas das bibliotecas padrão FatScript na versão `3.x.x`, use XMLoaf

Funções RLE

Nome	Assinatura	Breve descrição
toRLE	(chunk: Chunk): Chunk	Comprime para esquema RLE
fromRLE	(chunk: Chunk): Chunk	Descomprime de esquema RLE

Outras funções

Nome	Assinatura	Breve descrição
inferType	(val: Text): Any	Converte texto em void/boolean/number
minify	(src: Text): Text	Minifica código fonte FatScript

`minify` substituirá quaisquer instruções `$break` (ponto de interrupção do depurador) por `()`

Uso

JSON

Uma vez que FatScript aceita alternativamente sintaxe semelhante a JSON, `fromJSON` usa internamente o parser do FatScript, que é extremamente rápido, mas pode ou não produzir exatamente o que se espera de um analisador JSON.

Por exemplo, uma vez que o fragmento abaixo é analisado, já que `null` no FatScript é ausência de valor, não haverá declarações de entrada para "prop":

```
"prop": null
```

Portanto, ler com `fromJSON` e escrever de volta com `toJSON` não é necessariamente uma operação idempotente.

Veja também

- Pacote type
- Biblioteca SDK

sdk

Utilitários do kit de desenvolvimento de software do fry

> uma biblioteca especial que expõe alguns dos elementos internos do interpretador fry

Importação

```
_ <- fat.sdk
```

Métodos

Nome	Assinatura	Breve descrição
ast	(_): Void	Imprime árvore de sintaxe abstrata do nó
stringify	(_): Text	Converte nó em texto json
eval	(_): Any	Interpreta texto como programa FatScript
getVersion	(): Text	Retorna versão do fry
printStack	(depth: Number): Void	Imprime pilha do contexto de execução
readLib	(ref: Text): Text	Retorna o código-fonte da biblioteca fry
typeOf	(_): Text	Retorna o nome do tipo do nó
getTypes	(): List	Retorna info sobre tipos declarados
getDef	(name: Text): Any	Retorna definição de tipo por nome
getMeta	(): Scope	Retorna os metadados do interpretador
setKey	(key: Text): Void	Definir chave para pacotes ofuscados
setMem	(n: Number): Void	Definir limite de memória (contagem de nós)
runGC	(): Number	Rodar o GC, retorna transcorrido em milissegundos
quickGC	(): Number	Roda uma única iteração do GC e retorna ms
setAutoGC	(n: Number): Void	Configura GC para rodar a cada n novos nós

Notas de uso

readLib

```
_ <- fat.sdk
_ <- fat.console

print(readLib('fat.extra.Date'))  # imprime a implementação da biblioteca Date
```

> `readLib` não pode ver arquivos externos, mas `read` da biblioteca file pode

setKey

Use preferencialmente no arquivo `.fryrc` assim:

```
_ <- fat.sdk
setKey('secret')  # irá codificar e decodificar pacotes com esta chave
```

Veja mais sobre ofuscação.

setMem

Use preferencialmente no arquivo `.fryrc` assim:

```
_ <- fat.sdk
setMem(5000)  # ~2mb
```

Escolhendo entre GC completo, rápido e automático

A maioria dos scripts simples em FatScript não precisará se preocupar com a gestão de memória, pois as configurações padrão são projetadas para fornecer aos desenvolvedores uma capacidade de memória razoavelmente grande e um comportamento automático sensato logo de início.

O método `quickGC` oferece uma limpeza rápida e menos exaustiva, tornando-o adequado para cenários onde alguma margem na alocação de memória é aceitável. Por outro lado, `runGC` garante uma coleta de lixo determinística e completa, embora ao custo de tempos de execução mais longos, dependendo de vários fatores como o tamanho e a complexidade do grafo de memória. No entanto, em certos cenários, `quickGC` pode levar ao acúmulo de memória não reclamada e pode não ser a opção mais eficaz.

Além da escolha manual entre `quickGC` e `runGC`, existe também um GC automático baseado em heurísticas. Ele vem desativado por padrão, mas pode ser habilitado chamando `setAutoGC` com um valor não zero. Essa heurística aplica `quickGC` quando há memória livre suficiente, garantindo uma interrupção mínima. Em contraste, sob alta pressão de memória, `fullGC` é executado para uma limpeza abrangente. Esta estratégia equilibra a eficiência da memória com o desempenho da aplicação, adaptando-se dinamicamente ao padrão de uso da memória.

Veja mais sobre gerenciamento de memória.

system

Operações e informações em nível de sistema

Importação

```
_ <- fat.system
```

Tipos

Nome	Assinatura	Breve descrição
CommandResult	(code: ExitCode, out: Text)	Tipo de retorno de capture

Constantes

- successCode, 0: ExitCode
- failureCode, 1: ExitCode

Métodos

Nome	Assinatura	Breve descrição
args	(): List/Text	Retorna lista de argumentos passados pelo shell
exit	(code: Number): *	Sair do programa com o código de saída fornecido
getEnv	(var: Text): Text	Obter o valor da variável env por nome
shell	(cmd: Text): ExitCode	Executa cmd no shell, retorna o código de saída
capture	(cmd: Text): CommandResult	Captura a saída da execução de cmd
fork	(args: List/Text, out: Text = ø)	Inicia um processo em segundo plano, retorna PID
kill	(pid: Number): Void	Envia SIGTERM para o processo pelo PID
getLocale	(): Text	Obter configuração de localidade atual
setLocale	(cmd: Text): Number	Definir configuração de localidade atual
getMacId	(): Text	Obter identificador da máquina (endereço MAC)
blockSig	(enabled: Boolean): Void	Bloqueia SIGINT, SIGHUP e SIGTERM

Notas de uso

Atenção!

É importante agir com cautela e responsabilidade ao utilizar os métodos `getEnv`, `shell`, `capture`, `fork` e `kill`. A biblioteca `system` oferece a capacidade de executar comandos diretamente do sistema operacional, o que pode introduzir riscos de segurança se não forem utilizadas com cuidado.

Para mitigar vulnerabilidades, evite utilizar a entrada do usuário diretamente na construção de comandos passados para esses métodos. A entrada do usuário deve ser validada para prevenir ataques de injeção de comandos e outras violações de segurança.

Outras Limitações (multithreading)

Embora os métodos desta biblioteca suportem uma variedade de tarefas de programação, eles não são otimizados para uso intercalado dentro de Workers assíncronos. Ao iniciar processos de dentro de threads, opte pelos métodos `shell/capture`, ou use exclusivamente `fork/kill`. Misturar esses dois grupos de métodos em aplicações multithread pode resultar em comportamentos imprevisíveis.

> em cada chamada, `shell/capture` definirá `SIGCHLD` para seu comportamento padrão, enquanto `fork` ignorará esse sinal para tentar evitar processos zumbis

get/set locale

O interpretador `fry` tentará inicializar o locale `LC_ALL` para `C.UTF-8` e, se esse locale não estiver disponível no sistema, tentará usar `en_US.UTF-8`, caso contrário, usará o locale padrão.

Veja mais sobre nomes de localidade.

a configuração de localidade aplica-se apenas ao aplicativo e não é mantida após a saída do `fry`

time

Manipulação de hora e data

Importação

```
_ <- fat.time
```

tipo Number é importado automaticamente com esta importação

Métodos

Nome	Assinatura	Breve descrição
setZone	(offset: Number): Void	Definir fuso em milissegundos
getZone	(): Number	Obter diferença de fuso atual
now	(): Epoch	Obtenha o UTC atual em Epoch
format	(date: Text, fmt: Text = ø): Epoch	Converter data para Epoch
parse	(date: Text, fmt: Text = ø): Epoch	Converter Epoch em formato de data
wait	(ms: Number): Void	Aguardar milissegundos (suspender)
getElapsed	(since: Epoch): Text	Retorna o tempo decorrido como texto

Notas de uso

Epoch

No FatScript, o tempo é representado como um tipo aritmético para que você possa fazer contas.

Você pode obter o tempo decorrido entre tempo1 e tempo2 como:

```
decorrido = tempo2 - tempo1
```

Você também pode verificar se `tempo2` acontece após `tempo1`, simplesmente assim:

```
tempo2 > tempo1
```

format

Formata a data em texto como "%Y-%m-%d %H:%M:%S.milliseconds" (padrão), quando `fmt` é omitido.

milissegundos só podem ser transformados no formato padrão, caso contrário, a precisão é de até segundos

parâmetro fmt

A especificação de formato é um texto contendo uma sequência de caracteres especiais chamada especificações de conversão, cada uma das quais é introduzida por um caractere '%' e terminada por algum outro caractere conhecido como especificador de conversão. Todos os outros caracteres são tratados como texto comum.

Especificador	Significado
%a	Nome abreviado do dia da semana
%A	Nome completo do dia da semana
%b	Nome do mês abreviado
%B	Nome completo do mês
%c	Data/Hora no formato da localidade
%C	Número do século [00-99], o ano dividido por 100 e truncado para um número inteiro
%d	Dia do mês [01-31]
%D	Formato de data, igual a %m/%d/%y
%e	O mesmo que %d, exceto que um único dígito é precedido por um espaço [1-31]
%g	Parte do ano de 2 dígitos da data da semana ISO [00,99]

Especificador	Significado
%F	Formato de data ISO, igual a %Y-%m-%d
%G	Parte do ano de 4 dígitos da data da semana ISO
%h	O mesmo que %b
%H	Hora no formato de 24 horas [00-23]
%I	Hora em formato de 12 horas [01-12]
%j	Dia do ano [001-366]
%m	Mês [01-12]
%M	Minuto [00-59]
%n	Caractere de nova linha
%p	Cadeia AM ou PM
%r	Hora no formato AM-PM da localidade
%R	Formato de 24 horas sem segundos, igual a %H:%M
%S	Segundo [00-61], o intervalo de segundos permite um segundo bissexto e um segundo bissexto duplo
%t	Caractere de tabulação
%T	Formato de 24 horas com segundos, igual a %H:%M:%S
%u	Dia da semana [1,7], segunda-feira é 1 e domingo é 7
%U	Número da semana do ano [00-53], domingo é o primeiro dia da semana
%V	Número da semana ISO do ano [01-53]. Segunda-feira é o primeiro dia da semana. Se a semana contendo 1º de janeiro tiver quatro ou mais dias no ano novo, será considerada a semana 1. Caso contrário, será a última semana do ano anterior e o ano seguinte será a semana 1 do ano novo.
%w	Dia da semana [0,6], domingo é 0
%W	Número da semana do ano [00-53], segunda-feira é o primeiro dia da semana
%x	Data no formato da localidade
%X	Hora no formato da localidade
%y	Ano de 2 dígitos [00,99]
%Y	Ano de 4 dígitos (pode ser negativo)
%z	String de deslocamento UTC com formato +HHMM ou -HHMM
%Z	Nome do fuso horário
%%	Caractere %

Sob o capô `format` usa C's strftime e `parse` usa C's strptime, mas a tabela de especificação de formato acima se aplica praticamente nos dois sentidos.

type._

Extensões do protótipo para tipos nativos:

- Void
- Boolean
- Number
- HugeInt
- Text
- Method
- List
- Scope
- Error
- Chunk

> FatScript **não** carrega essas definições automaticamente no escopo global, portanto você deve **explicitamente** importar quando necessário

Importando

Se você quiser disponibilizar todos eles de uma só vez, basta escrever:

```
_ <- fat.type._
```

...ou importe um por um, conforme necessário, por exemplo:

```
_ <- fat.type.List
```

características comum

Todos os tipos neste pacote suportam os seguintes métodos de protótipo:

- apply (construtor)
- isEmpty
- nonEmpty
- size
- toText

Veja também

- Tipos (sintaxe)

Void

Extensões do protótipo Void

Importação

```
_ <- fat.type.Void
```

Construtor

Nome	Assinatura	Breve descrição
Void	(val: Any)	Retorna nulo, apenas ignora o argumento

Membros do protótipo

Nome	Assinatura	Breve descrição
isEmpty	(): Boolean	Retorna verdadeiro, sempre
nonEmpty	(): Boolean	Retorno falso, sempre
size	(): Number	Retorna 0, sempre
toText	(): Text	Retorna 'null' como texto

Exemplo

```
_ <- fat.type.Void
x.isEmpty  # true, já que x não foi declarado
```

Veja também

- Void (sintaxe)
- Pacote type

Boolean

Extensões do protótipo Boolean

Importação

```
_ <- fat.type.Boolean
```

Construtor

Nome	Assinatura	Breve descrição
Boolean	(val: Any)	Coage o valor para booleano

Membros do protótipo

Nome	Assinatura	Breve descrição
isEmpty	(): Boolean	Retorna verdadeiro se falso
nonEmpty	(): Boolean	Retorna falso se verdadeiro
size	(): Number	Retorna 1 se verdadeiro, 0 se falso
toText	(): Text	Retorna 'true' ou 'false' como texto

Exemplos

```
_ <- fat.type.Boolean

x = true
x.isEmpty  # falso, já que x é verdadeiro

Boolean('false')  # retorna true, porque o texto é não-vazio
Boolean('')       # retorna falso, porque é vazio
```

> note que o construtor não tenta converter o valor do texto, o que é consistente com as avaliações de controle de fluxo, e você pode usar um simples case se precisar fazer conversão de texto para booleano

Veja também

- Boolean (sintaxe)
- Pacote type

Number

Extensões do protótipo Number

Importação

```
_ <- fat.type.Number
```

Aliases

- Epoch: tempo de época do unix em milissegundos
- ExitCode: status de saída ou código de retorno
- Millis: duração em milissegundos

Construtor

Nome	Assinatura	Breve descrição
Number	(val: Any)	Texto para número ou tamanho da coleção

realiza a conversão de texto para número assumindo a base decimal

Membros do protótipo

Nome	Assinatura	Breve descrição
isEmpty	(): Boolean	Retorna verdadeiro se zero
nonEmpty	(): Boolean	Retorna verdadeiro se não-zero
size	(): Number	Retorna valor absoluto, igual a math.abs
toText	(): Text	Retorna número como texto
format	(fmt: Text): Text	Retorna número como texto formatado
truncate	(): Number	Retorna número descartando decimais

Exemplo

```
_ <- fat.type.Number
x = Number('52')  # número: 52
x.toText          # texto: '52'
x.format('.2')    # texto: '52.00'
```

format

O método `format` é usado para converter números em strings de várias maneiras. A estrutura básica de um especificador de formato é `%[flags][width][.precision][type]`. Aqui está o que cada um destes componentes significa:

- `flags` são caracteres opcionais que controlam o comportamento específico de formatação. Por exemplo, `0` pode ser usado para preenchimento com zeros e `-` para justificação à esquerda.
- `width` é um número inteiro que especifica o número mínimo de caracteres a serem impressos. Se o valor a ser impresso for mais curto do que este número, o resultado é preenchido com espaços em branco ou zeros, dependendo da flag utilizada.
- `precision` é um número opcional que segue um `.` que especifica o número de dígitos a serem impressos após o ponto decimal.
- `type` é um caractere que especifica como o número deve ser representado. Os tipos comuns são `f` (notação de ponto fixo), `e` (notação exponencial), `g` (fixo ou exponencial dependendo da magnitude do número) e `a` (notação de ponto flutuante hexadecimal).

Exemplos:

- `%5.f`: Isso imprimirá o número com uma largura total de 5 caracteres, sem dígitos após o ponto decimal (porque a precisão é `f`, que significa ponto fixo, mas nenhum número segue o ponto). Será justificado à direita porque nenhuma flag `-` é usada.
- `%05.f`: Semelhante ao anterior, mas como a flag `0` é usada, os espaços vazios serão preenchidos com zeros.
- `%8.2f`: Isso imprimirá o número com uma largura total de 8 caracteres, com 2 dígitos após o ponto decimal.
- `%-8.2f`: Semelhante ao anterior, mas o número será justificado à esquerda por causa da flag `-`.
- `%.2e`: Isso imprimirá o número usando notação exponencial, com 2 dígitos após o ponto decimal.
- `%.2a`: Isso imprimirá o número usando notação de ponto flutuante hexadecimal, com 2 dígitos após o ponto hexadecimal.
- `%.2g`: Isso imprimirá o número em notação de ponto fixo ou exponencial, dependendo da sua magnitude, com no máximo 2 dígitos significativos.

caso o `%` não esteja presente, `fmt` é automaticamente avaliado como `%<fmt>f`

Veja também

- Number (sintaxe)
- Biblioteca math
- Pacote type

HugeInt

Extensões do protótipo HugeInt

Importação

```
_ <- fat.type.HugeInt
```

Construtor

Nome	Assinatura	Breve descrição
HugeInt	(val: Any)	Número ou texto convertido para HugInt

realiza a conversão de texto para número assumindo a representação hexadecimal

Membros do protótipo

Nome	Assinatura	Breve descrição
isEmpty	(): Boolean	Retorna verdadeiro se zero
nonEmpty	(): Boolean	Retorna verdadeiro se não zero
size	(): Number	Retorna número de bits necessários para representar
toText	(): Text	Retorna número em texto hexadecimal
modExp	(exp: HugeInt, mod: HugeInt): HugeInt	Retorna exponenciação modular
toNumber	(): Number	Converte para número (com perda de precisão)

Notas de uso

Ao converter do tipo `Number` para `HugeInt`, o limite é 2^53, que é o valor máximo que pode ser representado com segurança como um inteiro sem perda de precisão. Tentar passar um valor acima deste limite resultará em um `ValueError`.

Por outro lado, ao converter de `HugeInt` para `Number`, valores de até 2^1023 - 1 podem ser convertidos com certo grau de perda de precisão. Tentar converter um valor acima disso resultará em `infinity` (infinito), o que pode ser verificado usando o método `isInf` fornecido pela biblioteca de matemática.

a biblioteca de matemática também fornece o valor `maxInt`, que serve para avaliar a potencial perda de precisão; se um número é menor que `maxInt`, sua conversão de `HugeInt` é considerada segura sem perda de precisão

Veja também

- HugeInt (sintaxe)
- Pacote type

Text

Extensões do protótipo Text

Importação

```
_ <- fat.type.Text
```

Construtor

Nome	Assinatura	Breve descrição
Text	(val: Any)	Coage valor para texto, igual a .toText

Membros do protótipo

Nome	Assinatura	Breve descrição
isEmpty	(): Boolean	Retorna verdadeiro se o comprimento for zero
nonEmpty	(): Boolean	Retorna verdadeiro se comprimento não-zero
size	(): Number	Comprimento do texto de retorno
toText	(): Text	Retorna o próprio valor
replace	(old: Text, new: Text): Text	Substituir old por new (todos)
indexOf	(frag: Text): Number	Obter índice de fragmento, -1 se ausente
contains	(frag: Text): Boolean	Verifica se o texto contém fragmento
count	(frag: Text): Number	Obtém contagem de repetições do fragmento
startsWith	(frag: Text): Boolean	Verifica se começa com o fragmento
endsWith	(frag: Text): Boolean	Check if ends with fragment
split	(sep: Text): List/Text	Divide texto por sep em lista
toLower	(): Text	Retorna a versão minúscula do texto
toUpper	(): Text	Retorna a versão maiúscula do texto
trim	(): Text	Retorna a versão aparada do texto
match	(regex: Text): Boolean	Retorna se o texto corresponde à regex
repeat	(n: Number): Text	Retorna o texto repetido n vezes
overlay	(base: Text, align: Text): Text	Retorna o texto sobreposto a base
patch	(i, n, val: Text): Text	Insere val na posição i, removendo n caracteres

Exemplo

```
_ <- fat.type.Text
x = 'banana'
x.size                          # retorna 6
x.replace('nana', 'nquete');  # produz 'banquete'
```

Regex

Ao definir expressões regulares, prefira usar textos brutos e lembre-se de escapar as barras invertidas conforme necessário, garantindo que as expressões regulares sejam interpretadas corretamente.

No momento, o suporte regex está limitado apenas à correspondência:

```
alphaOnly = "^[[:alpha:]]+$"
'abc'.match(alphaOnly)  # saída: true
```

o dialeto implementado é POSIX regex estendido

Overlay (sobreposição)

O valor padrão de alinhamento (se não fornecido) é 'left' (esquerda). Outros valores possíveis são 'center' (centro) e 'right' (direita):

```
'x'.overlay('___')           # 'x__'
'x'.overlay('___', 'left')   # 'x__'
'x'.overlay('___', 'center') # '_x_'
'x'.overlay('___', 'right')  # '__x'
```

o resultado é sempre do mesmo tamanho que o parâmetro `base`, o texto será cortado se for mais longo

Veja também

- Text (sintaxe)
- Pacote type

Method

Extensões do protótipo Method

Importação

```
_ <- fat.type.Method
```

Construtor

Nome	Assinatura	Breve descrição
Method	(val: Any)	Envolve val em um método

Membros do protótipo

Nome	Assinatura	Breve descrição
isEmpty	(): Boolean	Retorna falso, sempre
nonEmpty	(): Boolean	Retorna verdadeiro, sempre
size	(): Number	Retorna 1, sempre
toText	(): Text	Retorna o literal de texto 'Method'
arity	(): Number	Retorna a aridade do método

Exemplo

```
_ <- fat.type.Method
x = (): Number -> 3
(~ x).toText  # retorna 'Method'
```

note que é preciso optar por não usar chamada automática explicitamente para usar os membros do protótipo

Veja também

- Method (sintaxe)
- Pacote type

List

Extensões do protótipo List

Importação

```
_ <- fat.type.List
```

Construtor

Nome	Assinatura	Breve descrição
List	(val: Any)	Envolver val em uma lista

Membros do protótipo

Nome	Assinatura	Breve descrição
isEmpty	(): Boolean	Retorna verdadeiro se o comprimento for zero
nonEmpty	(): Boolean	Retorna verdadeiro para comprimento não zero
size	(): Number	Retorna o comprimento da lista
toText	(): Text	Retorna o literal de texto 'List'
join	(sep: Text): Text	Junta lista com separador em texto
flatten	(): List	Achata lista de listas em uma plana
find	(p: Method): Any	Retorna a primeira correspondência ou nulo
contains	(p: Method): Boolean	Checa se algum item corresponde ao predicado
filter	(p: Method): List	Retorna sub-lista que corresponde ao predicado
reverse	(): List	Retorna uma cópia invertida da lista
shuffle	(): List	Retorna uma cópia embaralhada da lista
unique	(): List	Retorna itens únicos da lista
sort	(): List	Retorna uma cópia ordenada da lista
sortBy	(key: Any): List	Retorna uma cópia ordenada da lista *
indexOf	(item: Any): Number	Retorna índice do item, -1 se ausente
head	(): Any	Retorna o primeiro item, nulo se vazio
tail	(): List	Retorna todos os itens, exceto o primeiro
map	(m: Method): List	Utilitário funcional (permite encadeamento)
reduce	(m: Method, acc: Any): Any	Utilitário funcional
walk	(m: Method): Void	Aplicar efeitos colaterais para cada item
patch	(i, n, val: List): List	Insere val na posição i, removendo n itens
headOption	(): Option	Retorna o primeiro item, como Option
itemOption	(index: Number): Option	Retorna item por índice, como Option
findOption	(p: Method): Option	Busca item por predicado, como Option

Exemplo

```
_ <- fat.type.List
x = [ 'a', 'b', 'c' ]
x.size  # retorna 3
```

Ordenação

Os métodos `sort` e `sortBy` implementam o algoritmo de ordenação quicksort, aprimorado com a seleção aleatória de pivô. Essa abordagem é conhecida por sua eficiência, oferecendo uma complexidade de tempo médio de caso de O(n log n). Demonstra alto desempenho na maioria dos conjuntos de dados. Para conjuntos de dados que contêm valores ou chaves repetidas uma ordenação estável não pode ser garantida e seu desempenho pode degradar até O(n^2) no pior caso, onde todos os elementos são idênticos ou tem a mesma chave.

`sortBy` aceita um parâmetro textual para `key` se for uma lista de `Scope` ou um parâmetro numérico caso seja uma lista de `List` (matriz), representando o índice

Reduzindo

O método `reduce` no FatScript transforma uma lista em um único valor aplicando um redutor (`m: Method`) a cada elemento em sequência, começando de um valor acumulador inicial (`acc: Any`), ou do primeiro elemento caso nenhum valor seja provido. Este método é útil para operações que envolvem agregar dados de uma lista.

Características

- **Método Redutor:** O redutor deve receber o valor acumulador atual e o item atual da lista, retornando o valor acumulador atualizado.
- **Comportamento com Lista Vazia:** Quando o `reduce` é aplicado a uma lista vazia sem um valor acumulador inicial, ele retorna `null`.

Exemplo Prático

```
_ <- fat.type.List
sumReducer = (acc: Número, item: Número) -> acc + item
sum = [1, 2, 3].reduce(sumReducer)  # resulta em 6
```

para transformações complexas de dados ou ao lidar com listas de escopos, estruture cuidadosamente o redutor para manipular os tipos de dados específicos e o resultado desejado

Veja também

- List (sintaxe)
- Tipo Option
- Pacote type

Scope

Extensões do protótipo Scope

Importação

```
_ <- fat.type.Scope
```

Construtor

Nome	Assinatura	Breve descrição
Scope	(val: Any)	Envolver val em um escopo

Membros do protótipo

Nome	Assinatura	Breve descrição
isEmpty	(): Boolean	Retorna verdadeiro se o tamanho for zero
nonEmpty	(): Boolean	Retorna verdadeiro para tamanho não zero
size	(): Number	Retorna o número de entradas do escopo
toText	(): Text	Retorna o literal de texto 'Scope'
copy	(): Scope	Retorna cópia profunda do escopo
keys	(): List	Retorna lista de chaves do escopo
maybe	(key: Text): Option	Retorna valor dentro de Option

Exemplo

```
_ <- fat.type.Scope
x = { num = 12, prop = 'outra' }
x.size  # retorna 2
```

Veja também

- Scope (sintaxe)
- Tipo Option
- Pacote type

Error

Extensões do protótipo Error

Importação

```
_ <- fat.type.Error
```

Aliases

- AssignError: atribuindo um novo valor a uma entrada imutável
- AsyncError: falha na operação assíncrona
- CallError: uma chamada é feita com argumentos insuficientes
- FileError: falha na operação de arquivo
- IndexError: o índice está fora dos limites da lista/texto
- KeyError: a chave (nome) não é encontrada no escopo
- SyntaxError: erro de sintaxe ou estrutura de código
- TypeError: inconsistência de tipo em chamada, retorno ou atribuição de método
- ValueError: tipo pode estar correto, mas conteúdo não é aceito

Construtor

Nome	Assinatura	Breve descrição
Error	(val: Any)	Retornar val coagido para texto como erro

Membros do protótipo

Nome	Assinatura	Breve descrição
isEmpty	(): Boolean	Retorna verdadeiro, sempre
nonEmpty	(): Boolean	Retorna falso, sempre
size	(): Number	Retorna 0, sempre
toText	(): Text	Retorna o texto do erro

Exemplo

```
_ <- fat.type.Error
x = Error('ops')
x.toText  # retorna "Error: ops"

# ...ou algo inesperado
e = undeclared.item  # gera erro
e.toText             # retorna "can't resolve scope of 'item'"
```

Veja também

- Biblioteca failure
- Error (sintaxe)
- Pacote type

Chunk

Extensões do protótipo Chunk

Importação

```
_ <- fat.type.Chunk
```

Construtor

Nome	Assinatura	Breve descrição
Chunk	(val: Any)	Converte valor para bloco (binário)

Membros do protótipo

Nome	Assinatura	Breve descrição
isEmpty	(): Boolean	Retorna verdadeiro se o tamanho for zero
nonEmpty	(): Boolean	Retorna verdadeiro se tamanho não-zero
size	(): Number	Retorna o tamanho do bloco (em bytes)
toText	(): Text	Converte o bloco para formato de texto
toBytes	(): List/Number	Converte o bloco para uma lista de bytes
seekByte	(byte: Number, offset: Number = 0): Number	Retorna índice da primeira correspondência
patch	(i, n, val: Chunk): Chunk	Insere val na posição i, removendo n bytes

`toText` substitui quaisquer sequências inválidas de UTF-8 por U+FFFD, representado como `&#65533;` em UTF-8

Exemplo

```
_ <- fat.type.Chunk

x = Chunk('example')

x.size     # retorna 7
x.toText   # retorna 'example'
x.toBytes  # retorna [ 101, 120, 97, 109, 112, 108, 101 ]
```

Veja também

- Chunk (sintaxe)
- Pacote type

extra._

Tipos adicionais implementados em FatScript puro:

- Date - Gerenciamento de calendário e datas
- Duration - Construtor de duração em milissegundos
- HashMap - Armazenamento rápido de chave-valor
- Logger - Suporte ao registro de logs
- Memo - Utilitário de memoização genérica
- Option - Encapsulamento de valor opcional
- Param - Verificação de presença e tipo de parâmetro
- Sound - Interface de reprodução de som
- Storable - Armazenamento de dados

Importando

Se você quiser disponibilizar todos eles de uma só vez, basta escrever:

```
_ <- fat.extra._
```

...ou importe um por um, conforme necessário, por exemplo:

```
_ <- fat.Date
```

Nota do desenvolvedor

Atualmente, a maioria desses utilitários não são otimizados para recursos ou desempenho.

A intenção aqui era mais fornecer recursos simples, como modelos básicos que podem ser extraídos via readLib, para que qualquer desenvolvedor com requisitos específicos tenha um ponto de partida para suas próprias implementações.

Date

Gerenciamento de calendário e datas

operações como adição e subtração de dias, meses e anos, garantindo o tratamento preciso de várias complexidades relacionadas a datas, como anos bissextos e cálculos de final de mês

Importação

```
_ <- fat.extra.Date
```

biblioteca time, biblioteca math, tipo Error, tipo Text, tipo List, tipo Number, tipo Duration são automaticamente importados com esta importação

Tipo Date

`Date` oferece uma solução abrangente para o gerenciamento de datas, incluindo anos bissextos e horário do dia.

Propriedades

- `year`: Número - Ano da data
- `month`: Número - Mês da data
- `day`: Número - Dia da data
- `tms`: Milissegundos - Horário do dia em milissegundos

valor padrão aponta para: 1 de janeiro de 1970

Membros do protótipo

Nome	Assinatura	Breve descrição
fromEpoch	(ems: Epoch): Date	Cria uma instância a partir de um epoch
isLeapYear	(year: Número): Boolean	Determina se um ano é bissexto
normalizeMonth	(month: Número): Número	Normaliza o número do mês
daysInMonth	(year: Número, month: Número): Número	Retorna o número de dias no mês de um ano
isValid	(year, month, day, tms): Boolean	Valida os componentes da data
truncate	(): Date	Trunca o horário do dia
toEpoch	(): Epoch	Converte a instância para tempo em epoch
addYears	(yearsToAdd: Número): Date	Adiciona anos à data
addMonths	(monthsToAdd: Número): Date	Adiciona meses à data
addWeeks	(weeksToAdd: Número): Date	Adiciona semanas à data
addDays	(daysToAdd: Número): Date	Adiciona dias à data

Exemplos de uso

```
_ <- fat.extra.Date

# Criar uma instância de Data
myDate = Date(2023, 1, 1)

# Adicionar um ano à data
newDate = myDate.addYears(1)

# Adicionar duas semanas à uma data
datePlusTwoWeeks = myDate.addWeeks(2)

# Criar uma Data a partir de um tempo em epoch (em milissegundos)
# o resultado é influenciado pelo fuso horário atual, veja: time.setZone
epochTime = 1672531200000
dateFromEpoch = Date.fromEpoch(Epoch(epochTime))
```

```
# Converter uma data para tempo em epoch
epochFromDate = myDate.toEpoch
```

Duration

Construtor de duração em milissegundos

No FatScript, o tempo é nativamente expresso em milissegundos, e esse tipo fornece uma maneira simples de expressar diferentes magnitudes de tempo em Millis.

Importação

```
_ <- fat.extra.Duration
```

Construtor

Nome	Assinatura	Breve descrição
Duration	(val: Number)	Cria um conversor de duração em Millis

Membros do protótipo

Nome	Assinatura	Breve descrição
nanos	(): Millis	Interpretar valor como nanossegundos
micros	(): Millis	Interpretar valor como microssegundos
millis	(): Millis	Interpretar valor como milissegundos
seconds	(): Millis	Interpretar valor como segundos
minutes	(): Millis	Interpretar valor como minutos
days	(): Millis	Interpretar valor como dias
weeks	(): Millis	Interpretar valor como semanas
months	(): Millis	Interpretar valor como meses (aprox.)
years	(): Millis	Interpretar valor como anos (aprox.)

Exemplo

```
_    <- fat.extra.Duration
time <- fat.time

cincoSegundos = Duration(5).segundos
time.wait(cincoSegundos)  # pausa a execução do thread por 5 segundos
```

HashMap

Armazenamento otimizado em memória de par chave-valor, servindo como um substituto de melhor desempenho para a implementação padrão do Scope, projetado para lidar eficientemente com grandes conjuntos de dados.

os ganhos de velocidade vem em detrimento de um maior uso de memória

Importação

```
_ <- fat.extra.HashMap
```

Construtor

Nome	Assinatura	Breve descrição
HashMap	(capacity: Number = 97)	Cria um HashMap com uma capacidade especificada

a capacidade padrão de 97 é geralmente eficiente para até 10.000 itens

Otimização de Capacidade

Idealmente, você deve manter no máximo cerca de 100 itens por 'compartimento' na tabela hash. Neste contexto, 'capacidade' refere-se ao número de compartimentos disponíveis para seus dados. Note que esta implementação não ajusta seu tamanho automaticamente, portanto, um dimensionamento inicial adequado é crucial. A seguinte tabla pode ajudar a determinar a capacidade ótima para armazenar `n` itens:

```
n <   5000 => 53
n <  10000 => 97
n <  20000 => 193
n <  40000 => 389
n <  80000 => 769
n < 160000 => 1543
_          => 3079
```

usar números primos pode ajudar a reduzir colisões

Estes valores são baseados em testes empíricos e devem ser ajustados de acordo com suas necessidades específicas de dados e objetivos de desempenho. Tenha em mente que a relação entre capacidade e desempenho não é totalmente linear; à medida que o número de itens aumenta, os benefícios de aumentar ainda mais a capacidade diminuem.

Recomendação

Embora o Scope padrão do FatScript mostre um desempenho mais lento para inserções e seja particularmente lento para deleções (como definir para `null`), ele se destaca na recuperação e atualização de dados, superando a velocidade do `HashMap` para pequenas coleções (menos de ~500 itens). Portanto, os benefícios de usar o `HashMap` são mais notáveis em cenários que envolvem inserções e deleções frequentes em grandes conjuntos de dados.

Membros do protótipo

Nome	Assinatura	Breve descrição
isEmpty	(): Boolean	Retorna verdadeiro se o comprimento for zero
nonEmpty	(): Boolean	Retorna verdadeiro para comprimento não zero
size	(): Number	Retorna o comprimento da tabela hash
toText	(): Text	Retorna o literal de texto 'HashMap/capacity'
set	(key: Text, value: Any): Any	Define um par chave-valor no HashMap
get	(key: Text): Any	Obtém o valor associado a uma chave
keys	(): List/Text	Retorna uma lista de todas as chaves do HashMap

Exemplo

```
_ <- fat.extra.HashMap
```

```
hmap = HashMap()
hmap.set('key1', 'value1')

hmap.get('key1')  # retorna 'value1'
hmap.keys         # retorna [ 'key1' ]
```

Logger

Suporte ao registro de logs

desde logs simples em console a registro baseado em arquivos

Importação

```
_ <- fat.extra.Logger
```

biblioteca console, biblioteca color, biblioteca file, biblioteca time, biblioteca sdk e biblioteca de tipos são automaticamente importadas com esta importação

Tipo Logger

`Logger` oferece capacidades de registro de logs personalizáveis com vários níveis e formatos.

Propriedades

- `level`: Text (padrão 'debug') - Nível de log
- `showTime`: Boolean (padrão verdadeiro) - Indicador para exibir carimbos de hora

níveis válidos: 'debug', 'info', 'warn', 'error'

Membros do protótipo

Nome	Assinatura	Breve descrição
setLevel	(level: Text)	Define o nível de log
setShowTime	(showTime: Boolean)	Alterna a exibição de carimbos de hora nos logs
asMessage	(level: Text, args: Scope): Texto	Formata mensagens de log (pode ser substituído)
log	(msg: Any, fg: Number)	Registra mensagens (pode ser substituído)

Métodos de log

- `debug(_1, _2, _3, _4, _5)`: Registra uma mensagem de debug
- `info(_1, _2, _3, _4, _5)`: Registra uma mensagem informativa
- `warn(_1, _2, _3, _4, _5)`: Registra uma mensagem de aviso
- `error(_1, _2, _3, _4, _5)`: Registra uma mensagem de erro

Subtipos

BoringLogger

- Herda de `Logger`
- Substitui `log` para emitir texto simples sem cor

FileLogger

- Herda de `Logger`
- Propriedades Adicionais:
  - `logfile`: Texto (padrão 'log.txt') - arquivo para registro de logs
- Substitui `log` para anexar mensagens a um arquivo

Exemplo de uso

```
_ <- fat.extra.Logger

# Crie uma instância com configurações personalizadas
myLogger = Logger(level = 'info', showTime = false)

# Registra uma mensagem informativa
myLogger.info('Esta é uma mensagem informativa.')
```

```
# Crie um FileLogger para registrar mensagens em um arquivo
fileLogger = FileLogger('meuLog.txt')
fileLogger.info('Registrado no arquivo.')
```

Memo

Utilitário de memoização genérica (também pode criar valores preguiçosos)

Importação

```
_ <- fat.extra.Memo
```

Construtor

Nome	Assinatura	Breve descrição
Memo	(method: Method)	Cria uma instância Memo para um método

a aridade do método memoizado pode ser 1 ou então 0 (para valores preguiçosos)

Membros do protótipo

Nome	Assinatura	Breve descrição
asMethod	(): Method	Retorna uma versão memoizada do método original
call	(arg: Any): Any	Chamada memoizada; armazena e retorna resultado

Exemplo

Memo é útil para otimizar funções, armazenando os resultados. Ela armazena o resultado das chamadas de função e retorna o resultado armazenado quando as mesmas entradas ocorrem novamente.

```
_ <- fat.extra.Memo

fib = (n: Number) -> {
  n <= 2 => 1
  _      => quickFib(n - 1) + quickFib(n - 2)
}

quickFib = Memo(fib).asMethod

quickFib(50)  # 12586269025
```

Agora você pode chamar `quickFib` como se estivesse chamando `fib`, mas com resultados armazenados em cache para entradas computadas anteriormente.

aviso: pode causar acúmulo na alocação de memória

Option

Encapsulamento de valor opcional

Importação

```
_ <- fat.extra.Option
```

> o tipo Error é automaticamente importado junto com esta importação

Tipos

Esta biblioteca introduz dois principais tipos: `Some` e `None`, que são casos especiais do tipo `Option`, fornecendo uma maneira de representar valores opcionais, encapsulando a presença (`Some`) ou ausência (`None`) de um valor.

Membros do protótipo

Nome	Assinatura	Breve descrição
isEmpty	(): Boolean	Verifica se a opção é None
nonEmpty	(): Boolean	Verifica se a opção é Some
get	(): Any	Retorna valor ou erro NoSuchElement
getOrElse	(default: Any): Any	Retorna valor ou default se for None
map	(fn: Method): Option	Aplica o método ao valor contido
flatMap	(fn: Method/Option): Option	Aplica método que retorna Option
filter	(predicate: Method): Option	Filtra o valor pelo predicado
toList	(): List	Converte a opção para Lista
concrete	(): Option	Resolve a opção para Some ou None

Exemplo de Uso

```
_ <- fat.extra.Option

# Criando opções
x = Some(5)  # equivalente a Option(5).concrete
y = None()   # equivalente a Option().concrete

# Trabalhando com opções
isEmptyX = x.isEmpty   # false
isEmptyY = y.isEmpty   # true
valX = x.getOrElse(0)  # 5
valY = y.getOrElse(0)  # 0

# Aplicando uma transformação
transformedX = x.map(v -> v * 2).getOrElse(0)  # 10
transformedY = y.map(v -> v * 2).getOrElse(0)  # 0

# Elevando valores a opção
label: Text = Option(opVal).concrete >> {
  Some => 'algum valor'   # caso onde opVal não é null
  None => 'nenhum valor'  # caso onde opVal é null
}
```

Option em Programação Funcional

No FatScript, `null` é integrado como um cidadão de primeira classe, permitindo que, na maioria dos cases, tipos nativos manipulem valores ausentes sem a necessidade de construtos adicionais para segurança. Consequentemente, o tipo `Option` está incluído no pacote `extra` como açúcar sintático.

Ele permite a encapsulação explícita de valores opcionais para clareza semântica ou aderência a certos paradigmas de programação funcional. Um exemplo de sua utilidade é demonstrado no tipo `Scope`, que inclui um método `maybe` além da sintaxe padrão de recuperação de valor:

- `myScope('key')` retorna o valor associado a `key` ou `null` se a chave não existir.
- `myScope.maybe('key')` fornece um valor envolto em `Option`, distinguindo explicitamente entre a existência (`Some`) e a ausência (`None`) de um valor.

Manipulação semântica de valores ausentes

Um dos principais benefícios de usar o tipo `Option` é sua capacidade de lidar semanticamente e com segurança com operações que envolvem valores potencialmente ausentes. Esse recurso é particularmente útil em operações primitivas ou transformações de dados onde valores `null` poderiam, de outra forma, levar a erros. Por exemplo, considere um cenário onde você precisa somar um número com um valor que pode não estar presente:

```
# Supondo que ovosComprados esteja definido e tenha um valor
ovosComprados: Number = ...

# geladeira.maybe('ovo') recupera o número de ovos na geladeira como uma Option
# Se 'egg' não estiver presente, o padrão é 0, evitando erros relacionados a null
totalDeOvos: Number = geladeira.maybe('ovo').getOrElse(0) + ovosComprados
```

Considerações de desempenho

O uso de tipos `Option` introduz sobrecarga computacional devido às chamadas de função necessárias para manipular valores e à memória adicional decorrente de sua estrutura subjacente. Embora os benefícios de segurança e expressividade sejam significativos, o custo de desempenho pode se tornar perceptível em loops apertados ou ao processar grandes conjuntos de dados.

Veja também

- Tipo Scope
- Tipo Error

Param

Verificação de presença e tipo de parâmetro

Importação

```
_ <- fat.extra.Param
```

> tipo Text e tipo Error são automaticamente importados com esta importação

Tipos

Esta biblioteca introduz o tipo `Param` e o utilitário `Using` para declaração de parâmetros implícitos.

Construtores

Os construtores de `Param` e `Using` recebem dois argumentos:

- **_exp**: o nome do parâmetro a ser verificado no contexto.
- **_typ**: o tipo esperado do valor avaliado.

Param

O tipo `Param` oferece mecanismos para verificar a presença e o tipo de parâmetros no contexto de execução.

Membros do protótipo

Nome	Assinatura	Resumo
get	(): Any	Recupera o parâmetro se corresponder ao tipo

> o método `get` gera `KeyError` se o parâmetro não estiver definido, e `TypeError` se o tipo não corresponder

Exemplo

```
_ <- fat.extra.Param

currentUser = Param('userId', 'Text')

...

# Assumindo que userId está definido no contexto e é um texto,
# recupere de forma segura seu valor do namespace atual
userId = currentUser.get
```

Using

Aplique `Using` para suprimir dicas de parâmetros implícitos em declarações de métodos para entradas esperadas no escopo.

> alternativamente, para suprimir avisos sobre parâmetros implícitos, nomeie a entrada implícita começando com um sublinhado (_)

Exemplo

```
_ <- fat.extra.Param

printUserIdFromContext = -> {
  Using('userId', 'Text')
  console.log(userId)
}
```

> se o parâmetro implícito estiver faltando ou não corresponder, um erro será gerado em tempo de execução quando o método for chamado

Veja também

- Pacote extra

Sound

Interface de reprodução de som

Wrapper para players de áudio de linha de comando usando fork e kill.

Importação

```
_ <- fat.extra.Sound
```

Construtor

O construtor `Sound` recebe três argumentos:

- **path**: o caminho do arquivo de áudio.
- **duração** (opcional): o tempo de espera (em milissegundos) para poder reproduzir novamente o arquivo. Geralmente, você deseja definir isso como a duração exata do seu áudio.
- **player** (opcional): o player padrão utilizado é `aplay` (utilitário de áudio comum no Linux, apenas dá suporte a wav), mas você pode usar `ffplay` para reproduzir mp3, por exemplo, definindo `ffplay = [ 'ffplay', '-nodisp', '-autoexit', '-loglevel', 'quiet' ]` e fornecendo-o como argumento para sua instância de som. Neste caso o pacote `ffmpeg` precisa estar instalado no sistema.

Membros do protótipo

Nome	Assinatura	Breve descrição
play	(): Void	Inicia reprodução, se ainda não estiver tocando
stop	(): Void	Interrompe reprodução, se ainda estiver tocando

o estado de "estar tocando" é inferido do parâmetro de duração

Exemplo

```
_    <- fat.extra.Sound
time <- fat.time

applause = Sound('applause.wav', 5000);
applause.play
time.wait(5000)
```

note que Sound cria um processo filho para reproduzir o áudio, portanto, a reprodução é assíncrona

Som na Web Build

Ao usar `fry` construído com Emscripten (por exemplo, ao usar FatScript Playground), este protótipo utiliza comandos embutidos `$soundPlay` e `$soundStop`, que são definidos apenas na compilação para web. Portanto, em vez de utilizar um reprodutor de áudio CLI através de fork do processo, há suporte de áudio via SDL2/WebAudio.

Veja também

- Pacote extra

Storable

Armazenamento de dados

Importação

```
_ <- fat.extra.Storable
```

> biblioteca file, biblioteca sdk, biblioteca enigma, tipo Error, tipo Text, tipo Void e tipo Method são automaticamente importados com esta importação

Mixins

Esta biblioteca introduz dois tipos de mixin: `Storable` e `EncryptedStorable`

Storable

O mixin `Storable` fornece métodos para armazenar e recuperar objetos no sistema de arquivos usando serialização JSON.

Membros do protótipo

Nome	Assinatura	Breve descrição
list	(): List/Text	Obtém lista de ids para instâncias armazenadas
load	(id: Text): Any	Carrega um objeto do sistema de arquivos
save	(): Boolean	Salva a instância do objeto atual
erase	(): Boolean	Exclui o arquivo associado ao id

> os métodos `load` e `save` emitem `FileError` em caso de falha

EncryptedStorable

Estende `Storable` com capacidades de criptografia para um armazenamento de dados mais seguro. Requer uma implementação do método `getEncryptionKey`.

Exemplo de uso

```
_ <- fat.extra.Storable

# Defina um tipo que inclua Storable (ou EncryptedStorable)
User = (
  Storable  # Inclui o mixin Storable

  # EncryptedStorable                              # implementação alternativa
  # getEncryptionKey = (): Text -> '3ncryp1ptM3'  # poderia obter via KMS ou
configuração

  ## Argumentos
  name: Text
  email: Text

  # Os setters retornam uma nova cópia imutável da instância com o campo atualizado
  setName = (name: Text) -> self + User * { name }
  setEmail = (email: Text) -> self + User * { email }
)

# Cria uma nova instância de usuário
newUser = User('Jane Doe', 'jane.doe@example.com')

# Salva o novo usuário
newUser.save

# Atualiza as informações do usuário e salva as alterações
```

```
updatedUser = newUser
  .setName('Jane Smith')
  .setEmail('jane.smith@example.com')
updatedUser.save

# Lista todos os usuários salvos
userIds = User.list

# Carrega um usuário do sistema de arquivos
userId = userIds(0)  # ...ou newUser.id
loadedUser = User.load(userId)

# Exclui os dados do usuário do sistema de arquivos
loadedUser.erase  # ...ou User.erase(userId)
```

Storable na Web Build

Ao usar `fry` construído com Emscripten (por exemplo, ao usar FatScript Playground), este protótipo utiliza comandos embutidos `$storableSave`, `$storableLoad`,`$storableList` e `$storableErase` , que são definidos apenas na compilação para web. Portanto, em vez de utilizar o sistema de arquivos convencional para o armazenamento, há suporte especial para utilização do objeto `localStorage` do navegador.

Veja também

- Pacote extra

Comandos embutidos

Comandos embutidos são funções de baixo nível do FatScript que podem ser invocadas com palavras-chave precedidas por um cifrão `$`. Esses comandos estão sempre disponíveis, implementados como código compilado e não requerem importações.

Ao contrário dos métodos, eles não recebem argumentos explícitos, mas podem ler a partir de nomes de entrada específicos no escopo atual ou até mesmo do estado interno do interpretador.

Os mais úteis

Aqui estão alguns comandos embutidos que podem valer a pena conhecer:

- `$break` pausa a execução e carrega o console de depuração
- `$debug` alterna os logs de depuração do interpretador
- `$exit` encerra o programa com o código fornecido
- `$keepDotFry` mantém a config (.fryrc) no escopo após a inicialização
- `$result` alterna a impressão do resultado no final da execução
- `$root` fornece uma referência ao escopo global
- `$self` fornece uma referência própria ao escopo do método/instância
- `$bytesUsage` retorna o total de bytes alocados no momento
- `$nodesUsage` retorna o total de nós alocados no momento
- `$isMain` verifica se o código está sendo executado como principal ou módulo

> as palavras-chave `root` e `self` são automaticamente convertidas em `$root` e `$self`

Você pode chamá-los diretamente no seu código, assim:

```
$exit  # encerra o programa
```

> para usar outros comandos embutidos você deve estudar a implementação C de `fry`, já que a lista completa não está documentada, consulte o arquivo embedded.c

Bibliotecas por trás dos bastidores

As bibliotecas padrão embalam chamadas embutidas como métodos, fornecendo uma interface mais ergonômica. Você não precisa criar um escopo de execução ou carregar argumentos nesse escopo antes de delegar a execução a eles.

Por exemplo, veja como você pode usar o método `floor` da biblioteca math:

```
_ <- fat.math
floor(2.53)
```

Este método é implementado como:

```
floor = (x: Number): Number -> $floor
```

Por trás dos bastidores, o método `floor` cria um escopo de execução e carrega um argumento como `x` nele. O método então delega a execução ao comando embutido `$floor`, que por sua vez, lê o valor de `x` do escopo atual e retorna o menor inteiro que não é maior que este número.

Você pode obter o mesmo resultado que o método acima fazendo o seguinte:

```
x = 2.53
$floor  # lê o valor de x do escopo atual
```

Hackeando

Você pode ver qual comando embutido um método de biblioteca está chamando, olhando para a implementação da biblioteca através do método `readLib` da biblioteca SDK. Tecnicamente, não há nada que o impeça de chamar comandos embutidos diretamente.

Por exemplo, você pode encerrar seu programa chamando `$exit` diretamente, que sairá com o código 0 (padrão) ou, se uma entrada numérica chamada `code` existir no escopo atual, o valor dessa entrada será usado como código de saída. No entanto, seria mais elegante importar a biblioteca `fat.system` e chamar o método `exit` com o código de saída desejado:

```
sys <- fat.system
sys.exit(0)  # exits with code 0
```

Essa abordagem torna seu código mais legível e menos propenso a erros, além de fornecer uma melhor separação de responsabilidades.

É importante ter em mente que os comandos embutidos são caixas pretas e não são destinados à escrita de código FatScript comum. Na maioria dos casos, você precisaria ler a implementação em C subjacente para entender melhor o que um comando está realmente fazendo.

Embora seja possível usar comandos embutidos para obter desempenho adicional em tempo de execução, evitando importações e chamadas de método, isso não é recomendado devido à perda de legibilidade do código. Em geral, é melhor usar as bibliotecas padrão e seguir as melhores práticas para escrever um código claro, fácil de manter.