파일

2015년 10월 21일에 한국어로 옮겨짐

원문: https://github.com/torch/torch7/blob/master/doc/file.md

목차

이것은 한 *추상* 클래스입니다. 파일은 디스크파일, 메모리파일, 파이프파일 같은 그것의 자식 클래스들에 의해 구현된 대부분의 메소드들을 정의합니다.

여기에 정의된 메소드들은 기본적인 읽기/쓰기 기능을 위해 만들어졌습니다. 읽기/쓰기 메소드들은 아스키 또는 바이너리 모드로 쓸 수 있습니다.

아스키 모드에서, 숫자는 사람이 읽을 수 있는 포맷(문자)으로 변환됩니다. 불리언(boolean)들은 `0`(false) 또는 `1`(true)로 변환됩니다. 바이너리 모드에서, 숫자와 불리언은 그 컴퓨터의 레지스터에서 표현되는 것처럼 인코드됩니다. 바이너리 모드는 비록 사람이 읽을 수 없고 이식성은 떨어지지만 뚜렷하게 빠릅니다.

아스키 모드에서, 만약 기본 옵션 autoSpacing()이 선택되면, 쓰여진 각 숫자 또는 불리언 뒤에 스페이스(space) 하나가 생깁니다. 각 쓰기 메소드의 호출 뒤에는 캐리지 리턴도 추가됩니다. 이 옵션에서는, 스페이스들이 읽는 동안에도 존재하는 것으로 가정됩니다. 이 옵션은 noAutoSpacing()으로 비활성화될 수 있습니다.

파일에 읽기/쓰기 에러나 문제가 있을 경우, 루아 에러가 생기거나 생기지 않을 수도 있습니다. 이는 quiet()와 pedantic() 옵션 선택에 따라 달라집니다. hasError()을 호출함으로써 마지막 연산에서 에러가 발생했는지 알아볼 수 있습니다.

읽기 메소드

읽기 메소드에는 세 종류가 있습니다.

- `[number] readTYPE()`
- `[TYPEStorage] readTYPE(n)`

- `[number] readTYPE(TYPEStorage)`

여기서 `TYPE`은 `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, 또는 `Double` 일 수 있습니다.

불리언 타입을 위한 편의 메소드도 존재합니다. `[boolean] readBool()`이 그것입니다. 이 메소드는 `readInt()`로 그 파일에서 값 하나를 읽습니다. 그리고 만약 그 값이 1이면 `true`를 리턴합니다. 불리언들로 구성된 스토리지들을 읽는 것은 불가능합니다.

이 모든 메소드는 인코딩 선택(아스키 또는 바이너리 모드)에 따라 달라집니다. 아스키 모드에서는, 옵션 autoSpacing()과 noAutoSpacing() 또한 이 메소드들에 영향을 미칩니다.

만약 파라미터들이 주어지지 않으면, 한 요소가 리턴됩니다. 이 요소는 읽을 때 루아 수 하나로 변환됩니다.

만약 `n`이 주어지면, 특정된 타입의 `n` 개 값이 읽히고 그 특정 타입의 새 스토리지 하나가 리턴됩니다. 그 스토리지의 크기는 실제로 읽은 요소 개수에 맞춰 정해집니다.

만약 한 스토리지가 인자로 입력되면, 그 메소드는 주어진 스토리지의 크기와 같은 수의 요소들을 읽으려 할 것입니다. 그리고 이 요소들로 그 스토리지를 가득 채울 것입니다. 리턴 값은 실제로 읽은 요소 개수입니다.

읽기 에러가 생길 경우, 이 메소드들은 루아 에러 함수를 호출할 것입니다. 기본 옵션은 pedantic입니다. 이 옵션은 에러 메시지들이 출력되게 설정합니다. 또는 quiet 옵션을 사용하여 에러 메시지가 출력되지 않게 설정할 수도 있습니다. quiet 옵션 사용 시, hasError()를 사용하여 에러가 생겼는지 확인할 수 있습니다.

쓰기 메소드

쓰기 메소드에는 두 종류가 있습니다.

- `[number] writeTYPE(number)`
- `[number] writeTYPE(TYPEStorage)`

여기서 `TYPE`은 `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, 또는 `Double`일 수 있습니다.

불리언 타입을 위한 편의 메소드도 존재합니다. `writeBool(value)`이 그것입니다. 만약 `value`가 `nil`이거나 `true`가 아니면 그 편의 메소드는 `writeInt(0)`을 호출하는 것과 같습니다. 그렇지 않으면, 그 편의 메소드는 `writeInt(1)`을 호출하는 것과 같습니다. 불리언들로 구성된 스토리지들에 쓰는 것은 불가능합니다.

이 모든 메소드는 인코딩 선택(아스키 또는 바이너리 모드)에 따라 달라집니다. 아스키 모드에서, 옵션 autoSpacing()과 noAutoSpacing() 또한 이 메소드들에 영향을 미칩니다.

만약 한 루아 수가 인자로 주어지면, 그 수는 쓸 때 그 메소드의 이름에 따라 변환됩니다 (예를 들어, `writeInt(3.14)`는 `3`을 쓸 것입니다).

만약 한 스토리지가 인자로 주어지면, 그 메소드는 그 스토리지에 담긴 모든 요소들을 쓰려고 할 것입니다.

이 메소드들은 실제로 쓴 요소 개수를 리턴합니다.

쓰기 에러가 생길 경우, 이 메소드들은 루아 에러 함수를 호출합니다. 기본 옵션은 pedantic입니다. 이 옵션은 에러 메시지들이 출력되게 설정합니다. 또는 quiet 옵션을 사용하여 에러 메시지들이 출력되지 않게 설정할 수도 있습니다. quiet 옵션 사용 시, hasError()를 사용하여 에러가 생겼는지 확인할 수 있습니다.

직렬화 메소드

이 메소드들은 사용자가 직렬화 가능 객체들을 디스크에 저장할 수 있게 합니다. 그리고 그것을 원래 상태대로 나중에 다시 로드할 수 있게 합니다. 다른 말로, 이 메소드들은 한 주어진 파일로 한 객체의 *깊은 복사*를 수행할 수 있습니다.

직렬화 가능한 객체들은 `read()`와 `write()` 메소드를 가진 토치(`Torch`) 객체들입니다. `table`, `number`, 또는 `string` 같은 *순수한* 루아 함수들 또한 직렬화 할 수 있습니다.

만약 저장할 객체가 몇 개의 다른 객체들을 포함하면 (그것이 객체들의 트리라고 합시다), 그 트리에서 여러 번 나타나는 객체들은 *오직 한 번만* 저장될 것입니다. 이는 디스크 공간을 절약하고, 저장과 로딩을 가속하며, 객체들 사이 의존성을 준수합니다.

흥미롭게도, 만약 그 파일이 메모리파일이면, 직렬화 메소드는 사용자가 직렬화 가능 객체의 클론을 쉽게 만들 수 있게 하는 데 쓰일 수 있습니다.

```
file = torch.MemoryFile() -- 메모리에 한 파일을 만듭니다.
file:writeObject(object) -- 그 객체를 파일에 씁니다.
file:seek(1) -- 그 파일의 처음으로 돌아옵니다.
objectClone = file:readObject() -- 객체의 한 클론을 얻습니다.
```

readObject()

그 파일에 writeObject()로 먼저 저장된, 다음 직렬화 가능 객체를 리턴합니다.

유념하십시오. 같은 참조를 가지고 쓰여진 객체들은 로딩 후에도 여전히 같은 참조를 가집니다.

예:

```
-- 같은 텐서 두 개를 가진 한 배열을 만듭니다.
array = {}
x = torch.Tensor(1)
table.insert(array, x)
table.insert(array, x)

-- array[1]과 array[2]는 같은 주소를 참조합니다.
-- x[1] == array[1][1] == array[2][1] == 3.14
array[1][1] = 3.14

-- 그 배열을 디스크에 씁니다.
file = torch.DiskFile('foo.asc', 'w')
file:writeObject(array)
file:close() -- 그 데이터가 쓰여졌는지 확인합니다.

-- 그 배열을 다시 로드합니다.
file = torch.DiskFile('foo.asc', 'r')
arrayNew = file:readObject()

-- arrayNew[1]와 arrayNew[2]는 같은 주소를 참조합니다!
-- arrayNew[1][1] == arrayNew[2][1] == 3.14
-- 그래서 만약 우리가 지금 다음과 같이 입력하면:
```

```
arrayNew[1][1] = 2.72
-- arrayNew[1][1] == arrayNew[2][1] == 2.72 !
```

writeObject(object)

`object`를 그 파일에 씁니다. 이 객체는 readObject()를 사용하여 나중에 읽힐 수 있습니다. 직렬화 가능 객체들은 `read()`와 `write()` 메소드를 가진 토치(`Torch`) 객체들입니다. `table`, `number`, 또는 `string` 같은 루아 객체들 또는 순수한 루아 함수들도 직렬화 가능합니다.

만약 객체가 이미 그 파일에 쓰여 있다면, 그 이미 저장된 객체를 가리키는 한 *참조*만 쓰여질 것입니다. 이는 공간을 아끼고 쓰기 속도를 높입니다. 또한, 이는 객체들 사이 의존성이 온전히 유지되게 합니다.

리턴에서, 만약 누군가 한 객체를 쓰고, 그것의 멤버를 수정하여, 그 객체를 같은 파일에 다시 쓰면, 그 수정은 그 파일에 기록되지 않을 것입니다. 오직 원본에 대한 참조만 쓰여질 것이기 때문입니다. 그 예는 readObject()를 침고하십시오.

[string] readString(format)

만약 `format`이 ""*l"로 시작하면, 그 파일에서 다음 줄을 리턴합니다. end-of-line 문자는 생략됩니다.

만약 `format`이 ""*a"로 시작하면, 그 파일의 남은 모든 내용들을 리턴합니다.

(그 파일이 quiet 모드인 경우를 제외하고) 만약 데이터를 읽을 수 없으면, 에러가 발생합니다. 이 때, 이 메소드는 빈 문자열 `''`을 리턴합니다. 그리고 그다음 your_file:hasError()로 end of file 때문에 마지막 읽기가 실패했다는 것을 볼 수 있을 것입니다.

숫자 타이핑(typing)은 토치가 더 정확하므로, 루아 포맷 ""*n""은 지원되지 않습니다. 대신 숫자 읽기 메소드 중 하나가 사용됩니다.

[number] writeString(str)

문자열 `str`을 파일에 씁니다. (그 파일이 quiet 모드인 경우를 제외하고) 만약 그 문자열이 완전히 쓰여질 수 없으면 에러가 발생합니다. 이 메소드는 실제로 파일에 쓰여진 문자의 개수를 리턴합니다.

일반적인 접근과 제어 메소드

---

ascii() [default]

데이터가 아스키 모드로 읽히거나 쓰여집니다. 모든 숫자가 (사람이 읽을 수 있는) 문자로 변환됩니다. 그리고 불리언은 `0`(false) 또는 `1`(true)로 변환됩니다. 이 모드에서의 입력-출력 포맷은 autoSpacing()과 noAutoSpacing() 옵션 선택에 따라 달라집니다.

autoSpacing() [default]

아스키 모드에서, 디스크 상에 쓰여지는 요소들 주변에 추가적인 스페이스들을 씁니다. 만약 한 스토리지에 쓴다면, 각 *요소* 사이에 스페이스 하나가 그리고 마지막 요소 뒤에 *리턴 라인* 하나가 생길 것입니다. 만약 요소 하나만 쓴다면, 그 요소 뒤에 *리턴 라인* 하나가 생길 것입니다.

이 모드에서 그 스페이스들은 읽는 동안에도 존재하는 것으로 가정됩니다.

이 옵션은 기본값입니다. 이 옵션은 noAutoSpacing() 메소드로 비활성화 될 수 있습니다.

binary()

데이터가 바이너리 모드로 읽히거나 쓰여집니다. 그 파일 내부 표현은 (사람이 읽을 수 없는) 그 컴퓨터의 메모리/레지스터에서의 표현과 같습니다. 이 모드는 아스키 보다 빠르지만 이식성은 떨어집니다.

clearError()

에러를 초기화합니다. 플래그는 hasError()로 리턴됩니다.

close()

그 파일을 닫습니다. 이 이후의 어떤 연산도 루아 에러를 발생시킵니다.

noAutoSpacing()

아스키 모드에서, 디스크 상에 쓰여지는 요소 사이에 추가 스페이스를 넣지 않습니다. 이는 autoSpacing() 옵션과 반대입니다.

synchronize()

만약 쓰는 동안 자식 클래스가 데이터를 버퍼화하면, 그 데이터가 실제로 쓰여지는 것을 보증합니다.

pedantic() [default]

만약 (기본 값인) 이 모드가 선택되면, 에러가 생길 때 루아 에러가 발생될 것입니다 (그 에러는 프로그램을 멈출게 할 수도 있습니다).

quiet()를 사용하여 루아 에러 발생을 피하고 대신 플래그가 설정되게 할 수도 있습니다.

[number] position()

파일에서의 현재 위치를 리턴합니다 (바이트 단위). 첫 위치는 (루아의 표준 인덱싱을 따라) 1입니다.

quiet()

만약 이 모드가 pedantic()대신 선택되면, 읽기/쓰기 에러가 생겨도 루아 에러는 발생하지 않습니다. 대신, hasError()로 읽을 수 있는 한 플래가 발생됩니다. 이 플래그는 clearError()로 초기화될 수 있습니다.

isQuiet()을 사용하여 한 파일 quiet인지 확인할 수 있습니다.

seek(position)

그 파일의 주어진 `position`으로 점프합니다 (바이트 단위). 문제가 생길 경우 에러가 발생할 수도 있습니다. 첫 위치는 (루아의 표준 인덱싱을 따라) 1입니다.

seekEnd()

그 파일의 끝으로 점프합니다. 문제가 생길 경우 에러가 발생할 수도 있습니다.

파일 상태 문의

이 메소드들은 사용자가 주어진 `File`의 상태를 알아볼 수 있게 합니다.

[boolean] hasError()

만약 마지막 clearError() 호출 이후 에러가 발생하면 리턴합니다. 또는 만약 그 파일이 열린 이후 `clearError()`가 한 번도 호출되지 않았으면 리턴합니다.

[boolean] isQuiet()

그 파일이 quiet 모드인지 아닌지 말해주는 한 불리언을 리턴합니다.

[boolean] isReadable()

그 파일을 읽을 수 있는지 아닌지 말해주는 한 불리언을 리턴합니다.

[boolean] isWritable()

그 파일에 쓸 수 있는지 아닌지 말해주는 한 불리언을 리턴합니다.

[boolean] isAutoSpacing()

만약 autoSpacing이 선택되어 있으면 `true`를 리턴합니다.

referenced(ref)

파일의 referenced 속성을 `ref`로 설정합니다. `ref`는 반드시 `true` 또는 `false`여야 합니다.

`ref`의 기본값은 `true`입니다. 이는 한 파일 객체가 (writeObject 메소드를 사용하여) 쓰여지거나 (readObject 메소드를 사용하여) 읽어지는 객체들을 계속 파악하고 있음을 의미합니다. 같은 주소를 가진 객체들은 오직 한 번만 쓰이거나 읽힐 것입니다. 이 접근은 공유된 메모리 구조를 유지합니다.

참조들을 계속 파악하는 데는 비용이 듭니다. 파일에서 직렬화되는 매 객체들은 (누군가 읽기/쓰기 뒤에 그 객체를 파기하더라도) 살아 있는 채로 유지됩니다. 파일이 그 포인터를 계속 추적할 필요가 있기 때문입니다. 이는 특히 큰 자료 구조를 다룰 때 썩 바람직하지 않습니다.

또다른 참조 추적이 바람직하지 않은 전형적 예는 누군가 같은 텐서 하나를 반복적으로 한 파일에 넣고자 하는데 그 내용은 매번 바뀔 때입니다. `referenced(false)` 호출은 우리가 바라는대로 동작하도록 확실히 합니다.

isReferenced()

referenced에 의해 설정된 상태를 리턴합니다.

목차

- 읽기 메소드
- 쓰기 메소드
- 직렬화 메소드
  - readObject()
  - writeObject(object)
  - [string] readString(format)
  - [number] writeString(str)
- 일반적인 접근과 제어 메소드
  - ascii() [default]
  - autoSpacing() [default]
  - binary()
  - clearError()
  - close()
  - noAutoSpacing()
  - synchronize()
  - pedantic() [default]
  - [number] position()
  - quiet()
  - seek(position)
  - seekEnd()
- 파일 상태 문의
  - [boolean] hasError()
  - [boolean] isQuiet()
  - [boolean] isReadable()
  - [boolean] isWritable()
  - [boolean] isAutoSpacing()
  - referenced(ref)
  - isReferenced()

목차

❖ 틀렸거나 보완할 점을 본문에 댓글로 또는 저에게 이메일로 알려 주시면 감사하겠습니다.