Hanbit eBook

Realtime 46

루아 프로그래밍 가이드

WOW, 앵그리버드에서 사용한 쉽고 빠른 스크립트 언어

Lua 5.2 Reference Manual

Lua.org 지음 / **권상구** 옮김 / **서진택** 감수

한빛미디어 Hanbit Media, Inc.

WOW, 앵그리버드에서 사용한 쉽고 빠른 스크립트 언어

루아 프로그래밍 가이드

WOW, 앵그리버드에서 사용한 쉽고 빠른 스크립트 언어 **루아 프로그래밍 가이드**

초판발행 2013년 11월 14일

지은이 Lua.org / **옮긴이** 권상구 / **펴낸이** 김태헌
펴낸곳 한빛미디어(주) / **주소** 서울시 마포구 양화로 7길 83 한빛미디어(주) IT출판부
전화 02-325-5544 / **팩스** 02-336-7124
등록 1999년 6월 24일 제10-1779호
ISBN 978-89-6848-651-7 15000 / **정가** 13,000원

책임편집 배용석 / **기획** 이중민 / **편집** 김연숙
디자인 표지 여동일, 내지 스튜디오 [밈], 조판 김정숙
마케팅 박상용

이 책에 대한 의견이나 오탈자 및 잘못된 내용에 대한 수정 정보는 한빛미디어(주)의 홈페이지나 아래 이메일로 알려주십시오.
한빛미디어 홈페이지 www.hanb.co.kr / **이메일** ask@hanb.co.kr

지금 하지 않으면 할 수 없는 일이 있습니다.
책으로 펴내고 싶은 아이디어나 원고를 메일(ebookwriter@hanb.co.kr)로 보내주세요.
한빛미디어(주)는 여러분의 소중한 경험과 지식을 기다리고 있습니다.

지은이_ **Lua.org**

Lua.org는 Lua 언어를 공식 관리하는 조직으로 Lua 언어의 관리와 커뮤니티 운영, 레퍼런스 문서 배포 등 다양한 활동을 하고 있다.

옮긴이_ **권상구**

2004년 경북대학교에 입학하였고, 2006년 게임 회사 KOG에 입사해 병역 특례를 마치고 지금까지 게임 프로그래머로 일하며 게임 '그랜드체이스'를 담당했었다. 현재는 게임 '파이터스클럽' 팀에서 클라이언트 프로그래머로 일하고 있다. 종종 회사 프로그래머를 대상으로 루아 강의를 진행하기도 한다.

견고한 구조를 만들어서 프로그래머의 도움 없이도 게임을 만들어갈 수 있는 환경을 제공하는 것을 선호한다. 루아도 이러한 구조를 만들어 가던 중에 알게 되었고 가벼운 스크립트 언어로서 큰 도움을 얻어 이 책을 번역했다. 평생 프로그래머로서 살아가는 것을 목표로 열심히 공부하고 노력하고 있다.

역자 서문

2006년 게임 회사 KOG에 처음 입사하고 팀에 배속되었을 때 배속된 팀에서는 Lua를 사용하고 있었습니다. 하지만 지금 생각해보면 훨씬 더 나은 방법이 있음에도 불구하고 굉장히 비효율적인 방식으로 Lua를 사용했습니다. 그 이유는 사용법을 잘 모르고 어느 블로그나 웹 사이트에 소개된 예제 코드만 조금 참조해서 도입했기 때문이라고 생각합니다. 그래서 저는 틈틈이 레퍼런스 문서를 읽어가면서 게임 안에 비효율적인 부분들을 찾아 개선해 나가기 시작했습니다.

그러던 2009년, 처음으로 Lua 레퍼런스를 번역하기 시작했습니다. 책으로 내겠다는 거창한 목표가 있었던 것이 아니라 제가 사용하는 기능들 위주로 번역해두고 개발 중간에 막히는 부분이 생겼을 때 얼른 찾아보기 위함이었습니다. 그런데 3년이라는 시간 동안 일을 하면서 점점 더 다양한 API들을 사용하게 되었고 사용하는 기능 위주로 번역해둔 내용은 어느새 레퍼런스 대부분에 해당하는 상황이 되어버렸습니다. 이러한 이유로 번역된 것을 책으로 만들게 되었습니다.

번역하면서 느낀 것은 이 문서가 처음부터 한글이었다면 얼마나 좋았을까 하는 점입니다. 왜냐하면 영어가 모국어인 나라에서는 어린이도 마음만 먹으면 이런 문서를 자유롭게 읽으면서 프로그래밍을 공부할 것이고, 영어를 모르는 사람에 비해서 훨씬 더 쉽게 많은 정보를 얻을 수 있겠다는 생각이 들었기 때문입니다. 실제로 다양한 레퍼런스 문서를 살펴보면 가까운 나라 일본에서는 상당히 많은 문서가 번역된 것을 볼 수 있습니다. 일본어는 지원하지만 한국어를 지원하지 않는 사이트들도 많이 볼 수 있었죠.

어디에서 이런 차이가 생기는지는 잘 모르겠습니다. 하지만 이 많은 문서가 한글로만 되어 있어도, 프로그래머들이 영어라는 벽을 만나지 않고도 훌륭한 결과물을 만들어낼 수 있을 것으로 생각합니다. 번역을 진행하다가 영어 문장을 읽는 것이 점

점 익숙해져서 한 60% 정도 진행했을 때는 굳이 번역해놓고 찾는 것보다 영문 사이트를 직접 살펴보는 게 편한 상황까지 이르렀습니다. 그래서 이렇게 번역한 것도 사실 과거의 추억으로 남겨두고 사라질 뻔 했습니다.

하지만 번역해보면서 많은 것을 배울 수 있었습니다. 저 스스로 문서를 꼼꼼하게 읽지 않는다는 것을 알게 되었습니다. 그래서 이 책을 읽는 여러 프로그래머분에게 부탁하고 싶습니다. 프로그래머는 결국 영어를 익히게 될 수밖에 없습니다. 화가 나지만 그게 현실입니다. 외국인처럼 영어를 유창하게 말할 수 있어야 하는 것이 아니라 수많은 기술 문서를 읽고 이해할 수 있어야 하고, 필요하다면 그러한 문서를 작성할 수 있어야 할 것입니다. 그 하나의 발판으로 이렇게 번역을 해보는 것을 추천하고 싶습니다. 얼마 전에 http://en.cppreference.com에서 C++ 관련 정보를 찾아보다가 언어 메뉴에 또 한국어만 없다는 것을 알게 되어 사이트 관리자에게 한국어 링크도 만들어 달라는 요청 메일을 보내 http://ko.cppreference.com라는 링크를 받았습니다. 그리고 Lua 레퍼런스처럼 시간 날 때마다 한 페이지씩 번역하곤 합니다. 저는 영어를 뛰어나게 잘하는 사람도 아니고 외국인이랑 대화도 못 하는 일반인입니다. 대단한 사람만 번역하는 것이 아니고, 대단한 사람만 책을 내는 것이 아닙니다. 여러분도 언젠가 자신의 이름을 가진 책을 출간하는 프로그래머가 되기를 바랍니다.

끝으로 초보 프로그래머의 번역서를 내기로 해준 한빛미디어에 감사드립니다. 독자를 신뢰하고 PDF로 eBook을 판매하는 마음에 감동해 첫 책은 한빛미디어에서 내겠다고 마음먹었습니다. 그리고 역자의 어색한 번역 부분을 꼼꼼하게 확인해 준 한빛미디어 이중민 대리님께도 감사드립니다. 또한 저와 일면식도 없지만 제가 LuaTinker를 책에 인용하고 싶다는 요청을 수락해 주신 이권일 님과, 같은 직장

동료에게 도움을 주고자 기꺼이 감수를 맡아준 '게임 개발자를 위한 C++'의 저자인 서진택 님께도 감사드립니다. 베타 리더를 자처해 준 이상호 님과 공호원 님께도 감사드립니다. 귀찮은 일을 선뜻하겠다고 이야기해 주셔서 정말 감사합니다.

매주 평일에는 회사에서 일한다고 바쁘고 주말에는 번역한다고 바쁘다고 연락도 자주 하지 못하고 찾아뵙지 못한 부모님께도 이 서문을 빌어 죄송하다는 말씀을 드립니다. 바쁘다고 집안일에 신경을 못 쓰는 점을 이해해 주려고 노력하는 아내 정은미에게 미안하고 사랑한다는 말을 전합니다.

번역을 마치며

역자 권상구

참고 사이트

- Lua 공식 사이트

  : http://www.lua.org

- Lua 5.2 레퍼런스 문서 사이트

  : http://www.lua.org/manual/5.2/

한빛 eBook 리얼타임

한빛 eBook 리얼타임은 IT 개발자를 위한 eBook 입니다.

요즘 IT 업계에는 하루가 멀다 하고 수많은 기술이 나타나고 사라져 갑니다. 인터넷을 아무리 뒤져도 조금이나마 정리된 정보를 찾는 것도 쉽지 않습니다. 또한 잘 정리되어 책으로 나오기까지는 오랜 시간이 걸립니다. 어떻게 하면 조금이라도 더 유용한 정보를 빠르게 얻을 수 있을까요? 어떻게 하면 남보다 조금 더 빨리 경험하고 습득한 지식을 공유하고 발전시켜 나갈 수 있을까요? 세상에는 수많은 종이책이 있습니다. 그리고 그 종이책을 그대로 옮긴 전자책도 많습니다. 전자책에는 전자책에 적합한 콘텐츠와 전자책의 특성을 살린 형식이 있다고 생각합니다.

한빛이 지금 생각하고 추구하는, 개발자를 위한 리얼타임 전자책은 이렇습니다.

1. **eBook Only - 빠르게 변화하는 IT 기술에 대해 핵심적인 정보를 신속하게 제공합니다.**

   500페이지 가까운 분량의 잘 정리된 도서(종이책)가 아니라, 핵심적인 내용을 빠르게 전달하기 위해 조금은 거칠지만 100페이지 내외의 전자책 전용으로 개발한 서비스입니다. 독자에게는 새로운 정보를 빨리 얻을 수 있는 기회가 되고, 자신이 먼저 경험한 지식과 정보를 책으로 펴내고 싶지만 너무 바빠서 엄두를 못 내시는 선배, 전문가, 고수분에게는 보다 쉽게 집필하실 기회가 되리라 생각합니다. 또한 새로운 정보와 지식을 빠르게 전달하기 위해 O'Reilly의 전자책 번역 서비스도 하고 있습니다.

2. **무료로 업데이트되는, 전자책 전용 서비스입니다.**

   종이책으로는 기술의 변화 속도를 따라잡기가 쉽지 않습니다. 책이 일정한 분량 이상으로 집필되고 정리되어 나오는 동안 기술은 이미 변해 있습니다. 전자책으로 출간된 이후에도 버전 업을 통해 중요한 기술적 변화가 있거나, 저자(역자)와 독자가 소통하면서 보완되고 발전된 노하우가 정리되면 구매하신 분께 무료로 업데이트해 드립니다.

3. 독자의 편의를 위하여, DRM-Free로 제공합니다.

구매한 전자책을 다양한 IT기기에서 자유롭게 활용하실 수 있도록 DRM-Free PDF 포맷으로 제공합니다. 이는 독자 여러분과 한빛이 생각하고 추구하는 전자책을 만들어 나가기 위해, 독자 여러분이 언제 어디서 어떤 기기를 사용하시더라도 편리하게 전자책을 보실 수 있도록 하기 위함입니다.

4. 전자책 환경을 고려한 최적의 형태와 디자인에 담고자 노력했습니다.

종이책을 그대로 옮겨 놓아 가독성이 떨어지고 읽기 힘든 전자책이 아니라, 전자책의 환경에 가능한 최적화하여 쾌적한 경험을 드리고자 합니다. 링크 등의 기능을 적극적으로 이용할 수 있음은 물론이고 글자 크기나 행간, 여백 등을 전자책에 가장 최적화된 형태로 새롭게 디자인하였습니다.

앞으로도 독자 여러분의 충고에 귀 기울이며 지속해서 발전시켜 나가도록 하겠습니다.

차례

1 | 소개

Lua는 일반 절차적 프로그래밍과 데이터 표현 편의 기능을 지원하기 위한 확장 언어로 객체 지향 프로그래밍, 함수형 프로그래밍, 데이터 주도 프로그래밍 기법을 지원합니다. Lua는 강력하면서도 가벼운 스크립트 언어를 필요로 하는 모든 프로그램에서 사용할 수 있도록 C와 C++의 공통 분모라고 할 수 있는 순수 C[clean C] 언어로 구현된 라이브러리이기도 합니다.

Lua는 확장 언어이므로 메인 프로그램이라는 개념이 없습니다. 즉, 내장 프로그램 혹은 단순히 호스트라고 부르는 호스트 클라이언트에 내장되어야만 동작합니다. 또한 호스트 프로그램은 Lua의 조각 코드를 실행하기 위해 함수를 호출하거나 Lua의 변수를 읽고 쓸 수 있으며, Lua 코드에 C 함수를 등록해 호출할 수도 있습니다. 이렇게 C 함수를 사용함으로써 Lua를 좀 더 다양한 도메인에 맞추어 확장할 수 있으며, 이런 특징을 잘 이용하면 문법적 기반을 공유하는 사용자화 프로그래밍 언어를 만들 수도 있습니다. 그리고 Lua 배포본에는 lua라고 부르는 샘플 호스트 프로그램이 포함되어 있습니다. 이 프로그램은 Lua 라이브러리를 완전히 지원하는 독립 Lua 인터프리터로 대화형이나 일괄 처리에 사용합니다.

Lua는 무료 소프트웨어로, 라이선스에서 언급한 것처럼 다른 무료 소프트웨어와 동일하게 사용료 없이 무료로 사용할 수 있지만 문제가 발생했을 때 책임을 지지는 않습니다. Lua 공식 웹 사이트인 www.lua.org에서 Lua의 프로그래밍 매뉴얼을 살펴볼 수 있습니다.

다른 레퍼런스 문서와 마찬가지로 이 문서도 원론적이거나, 설명이 부족하거나, 이해가 어려울 수도 있습니다. 양해 부탁드립니다. 또한 Lua의 디자인에 대한 토론이나 기술 문서들을 읽어볼 분은 Lua 웹 사이트를 이용하기 바랍니다. 마지막으로 Lua 프로그래밍에 대한 자세한 소개는 호베르토Roberto가 쓴 『Programming in Lua』[01]를 참고하기 바랍니다.

01 역자주_국내에서는 2007년 인사이트에서 『프로그래밍 루아』라는 책으로 출간되었다. 루아 5.1 기준으로 설명한다.

2 | 기본 개념

2장은 Lua 언어의 기본 개념을 소개합니다.

2.1 값과 타입

Lua는 동적으로 타입이 결정되는 언어입니다. 즉, 변수가 특정한 타입을 가지지 않는다는 의미입니다. Lua는 특정 타입을 정의하지 않으므로 변수는 타입에 대한 제한이 없고 값 스스로가 타입을 가집니다.

Lua의 모든 값들은 1차 값first-class value입니다. 1차 값이란 (어떤) 값이 변수에 저장될 수 있고, 함수의 인자로 전달될 수 있으며 반환 값으로 사용될 수 있다는 의미입니다.

Lua에는 기본적으로 nil, 불리언boolean, 수치number, 문자열string, 함수function, 유저 데이터userdata, 스레드thread, 테이블table이라는 여덟 가지 타입이 있습니다. 각 타입이 갖는 의미는 다음과 같습니다.

- nil은 nil 값을 가리키는 타입이며, 보통 (쓸모 있는) 값의 부재를 나타낼 때 사용합니다. 다른 값들과는 전혀 다른 속성을 가집니다(그래서 전역 변수의 값을 지울 때 타입을 nil로 정의하고, 기본값을 지정하지 않을 경우엔 전역 변수의 형은 nil이 됩니다).
- 불리언은 false와 true를 가리키는 타입입니다. nil과 false는 논리식에서 false로 처리되며 그 외의 다른 값들은 true로 처리됩니다. 따라서 0도 true로 처리됩니다.

- 수치는 배밀도 부동 소수점으로 표현합니다. 수치와 관련된 연산은 C 구현 기반과 동일한 규칙인 IEEE 754 표준을 따릅니다(luaconf.h 파일을 보면 수치를 무엇으로 표현할 것인지에 대한 정의가 있습니다. 이 정의를 바꿔서 단밀도 부동 소수점이나 정수로 타입을 변경할 수 있습니다).[01]

- 문자열은 변경되지 않는 바이트 배열로 표현합니다. Lua는 8비트를 사용하기 때문에 종결 문자('\0')를 포함한 8비트 문자열 값을 포함할 수 있습니다.

Lua에서는 Lua로 만들어진 함수 혹은 C로 작성된 함수를 호출하거나 조작할 수 있습니다('3.4.9 함수 호출' 참조).

유저데이터는 Lua 변수에 저장된 임의의 C 데이터로 제공되며 값은 메모리 블록의 포인터입니다. 유저데이터의 형태로는 풀 유저데이터full userdata와 라이트 유저데이터light userdata 두 가지가 있는데 풀 유저데이터는 Lua에서 메모리를 관리하고, 라이트 유저데이터는 호스트 프로그램에서 메모리를 관리합니다. Lua에는 유저데이터와 관련하여 할당과 식별 검사identity test를 위한 동작 외에는 어떠한 동작도 정의되어 있지 않습니다. 메타테이블을 이용하면 풀 유저데이터에 대한 동작을 프로그래머가 정의할 수 있습니다('2.4 메타테이블과 메타메서드' 참조). 유저데이터 값은 Lua에서 만들거나 수정할 수 없고 C API를 사용해야 합니다. 이런 특징 덕분에 호스트 프로그램이 데이터에 대한 모든 권한을 갖습니다.

스레드 타입은 독립적으로 실행되는 스레드를 의미하며 코루틴coroutine으로 구현하는 데 사용됩니다('2.6 코루틴' 참조). 참고로 운영체제의 스레드와 Lua의 스레드는 전혀 다르며 Lua의 코루틴은 스레드를 지원하지 않는 운영체제에서도 사용할 수 있습니다.

---

01 역자주_luaconf.h에서 Lua_Number에 대한 정의를 바꿀 때는 주의해야 합니다. 정수 타입으로 변경하면 Lua에서 소수점 연산을 사용할 수가 없고, 단밀도 부동 소수점 타입으로 변경하면 6자리가 넘는 수치에 대해 잘못된 결과 값을 얻을 수도 있습니다.

테이블 타입은 연관 배열associative array로 구현되어 있습니다. 즉, 수치 이외의 값도 배열의 인덱스로 사용할 수 있다는 의미입니다. 하지만 nil이나 NaN(Not a Number, 정의되지 않거나 0/0과 같이 표현할 수 없는 결과를 나타낼 때 또는 지칭할 때 쓰이는 특수한 숫자 값) 값은 인덱스로 사용할 수 없습니다. 테이블은 이형적heterogeneous 성질을 가지기 때문에 특정 타입의 값이 아닌 여러 가지 타입의 값들을 포함할 수 있습니다. 따라서 값이 nil인 키는 테이블에 포함되지 않습니다. 반대로 테이블에 포함되지 않는 키에 대한 값은 nil이 됩니다.

테이블은 Lua의 유일한 데이터 구조로서 일반적인 배열, 시퀀스, 심벌 테이블, 셋, 레코드, 그래프, 트리 등의 자료 구조를 표현할 수 있습니다. 예를 들어 레코드를 표현하려면 필드 이름을 인덱스로 사용하면 됩니다. 그리고 사용자 편의를 위해 a["name"]의 형태를 a.name과 같은 표현으로 사용할 수 있습니다. 테이블을 생성할 수 있는 편리한 방법 몇 가지는 '3.4.8 테이블 생성자'를 참조하기 바랍니다.

어떤 정수 n에 대해 {1, ..., n}인 양수 숫자 키를 사용하는 테이블을 나타내려면 시퀀스 이론을 사용합니다. 여기서 n은 시퀀스의 길이입니다('3.4.6 길이 연산자' 참조).

인덱스와 마찬가지로 테이블 필드의 값 역시 어떤 타입이든 사용할 수 있습니다. 특히 함수 또한 1차 값이므로 테이블 필드에 함수도 포함할 수 있습니다. 따라서 테이블은 메서드도 포함할 수 있습니다('3.4.10 함수 정의' 참조).

테이블의 인덱싱은 언어 내부의 동등성 규칙을 따릅니다. a[i]와 a[j]는 i와 j가 근본적으로 동등할 때만 같은 테이블 요소라고 할 수 있습니다(단, 이 규칙은 메타메서드가 없을 때만 만족합니다).

테이블, 함수, 스레드, 유저데이터(풀 유저데이터)는 객체입니다. 변수는 이 객체를 저장하는 것이 아니라 참조합니다. 즉 할당, 파라미터 전달, 함수 반환 등에서 이루어지는 변수 사용은 참조로 이루어집니다. 단, 이러한 과정에서 어떠한 종류의 복사도 수행하지 않습니다.

라이브러리로 제공되는 type() 함수를 통해 해당 값의 타입을 문자열 형태로 얻어낼 수 있습니다('6.1 기본 함수' 참조).

2.2 환경과 전역 환경

'3.2 변수'나 '3.3.3 할당'에서 자세히 설명하겠지만 전역 이름global name var를 참조하는 모든 변수는 문법적으로 _ENV.var로 변환됩니다. 또한 모든 청크chunk는 _ENV라는 외부 지역 변수 안에 컴파일되어 있습니다('3.3.2 청크' 참조). 따라서 청크 안에서는 _ENV 자신을 전역 이름으로 생각하면 안 됩니다.

외부 _ENV 변수와 변환된 전역 이름이 존재함에도 불구하고 _ENV는 완전한 정규 이름regular name입니다. 특히 새로운 변수와 파라미터를 _ENV라는 이름으로 정의할 수도 있습니다. 전역 이름에 대한 각 참조는 프로그램의 해당 위치에서 접근할 수 있는 _ENV를 사용하며, 이는 Lua의 일반적인 가시성 규칙을 따릅니다('3.5 가시성 규칙' 참조).

_ENV의 값으로 사용되는 테이블을 환경environment이라고 합니다.

Lua는 전역 환경을 별도의 환경으로 유지합니다. 이 값은 C 레지스트리의 특별한 인덱스로 유지합니다('4.5 레지스트리' 참조). Lua에서는 변수 _G를 같은 값으로 초기화합니다.

Lua가 청크를 컴파일할 때는 전역 환경의 _ENV를 업밸류upvalue에 초기화합니다('load()' 참조). 따라서 Lua 코드의 전역 변수는 기본적으로 전역 환경을 참조하게 됩니다. 또한 모든 기본 라이브러리는 환경을 다루기 위한 일부 함수와 전역 환경에 로드됩니다. load() 함수나 loadfile() 함수를 사용해 다른 환경에 청크를 로드할 수 있습니다(C에서는 청크를 로드한 다음 첫 번째 업밸류를 바꿔줘야 합니다).

만약 (C 코드나 디버그 라이브러리를 통해) 레지스트리의 전역 환경을 바꾸었다면 변경 이후에 로드되는 모든 청크는 새로운 환경을 사용하게 될 것입니다. 단, 변경 전에 로드된 청크는 영향을 받지 않지만 _ENV 변수에 저장된 각각의 환경을 참조할 것입니다. 게다가 (원본 전역 환경을 저장한) _G 변수는 절대 Lua에 의해 업데이트되지 않습니다.

2.3 에러 처리

Lua는 내장형 확장 언어이기 때문에 모든 동작은 루아 라이브러리의 함수를 호출하는, 호스트 프로그램의 C 코드에서 시작합니다('lua_pcall()' 참조). Lua 청크의 실행이나 컴파일이 진행되는 동안 에러가 발생하면 호출된 함수는 호스트 프로그램에서 다시 제어하게 되고 에러를 확인해볼 수 있습니다(예를 들어 에러 메시지를 출력할 수 있습니다).

이를 위해 Lua 코드에서 error() 함수를 호출해 명시적으로 에러를 생성할 수 있습니다. 만약 Lua에서 에러를 처리해야 한다면 보호 모드에서 특정 함수를 호출하는 pcall() 함수나 xpcall() 함수를 사용하면 됩니다.

에러가 발생할 때마다 에러 객체(또는 에러 메시지라고 부름)에는 에러 정보가 전달됩니다. Lua에서는 에러 객체가 문자열인 것만 생성할 수 있지만 프로그램은 에러 객체를 위한 어떠한 값을 사용해서든 에러를 생성할 수 있습니다.

xpcall() 함수나 lua_pcall() 함수를 사용하면 에러가 발생할 때 처리할 메시지 핸들러를 지정할 수 있습니다. 이 핸들러는 호출될 때 원본 에러 메시지를 전달받아 새로운 에러 메시지를 반환합니다. 또한 호출될 때는 에러가 발생하기 전의 스택으로 에러를 되돌리기 때문에 스택의 내용을 살피거나 스택을 역추적하면서 에러에 대한 좀 더 많은 정보를 수집할 수 있습니다.

그리고 메시지 핸들러는 보호된 호출에 의해 보호 상태가 되므로 메시지 핸들러가 핸들링하는 동안에 핸들러 내부에서 에러가 발생하면 메시지 핸들러를 다시 호출할 것입니다. 이러한 루프가 발생하면 Lua는 해당 루프를 멈추고 적절한 에러 메시지를 반환합니다.

2.4 메타테이블과 메타메서드

Lua 내부의 모든 값은 메타테이블metatable을 가질 수 있습니다. 메타테이블은 일반 Lua 테이블을 말하며 특정 연산자를 사용할 때 원본 값에 대한 동작을 정의합니다. 따라서 메타테이블의 특정 필드를 설정하는 것으로 값에 대한 여러 연산자의 동작을 변경할 수 있습니다. 예를 들어 수치가 아닌 값에 더하기 연산을 수행하려고 하면 Lua는 메타테이블의 "__add" 필드를 확인합니다. 해당 필드를 발견하면 Lua는 더하기 연산에 이 함수를 사용합니다.

메타테이블의 키는 이벤트 이름에서 유래했으며 키와 일치하는 값의 메타메서드metamethod가 호출됩니다. 앞 예에서 이벤트는 "add"이고 메타메서드는 더하기 연산을 수행하는 함수입니다.

getmetatable() 함수를 사용하면 원하는 메타테이블 값을 조회할 수 있고, setmetatable() 함수를 사용하면 테이블에 설정된 메타테이블을 교체할 수 있습니다. 단, 테이블 타입이 아닌 값은 debug 라이브러리를 통하지 않고는 교체할 수 없습니다. 다른 타입의 메타테이블을 교체하려면 C API를 사용해야만 합니다.

테이블과 유저데이터는 메타테이블을 가질 수 있습니다(여러 테이블과 유저데이터는 동일한 메타테이블을 공유할 수 있습니다). 그 이외의 수치, 문자열 등 각 타입 값들은 각각의 타입에 따라 하나의 메타테이블을 공유합니다. 기본적으로 값은 메타테이블을 가지지 않지만 문자열 라이브러리는 문자열 타입을 위한 메타테이블을 설정합니다('6.4 문자열 조작' 참조).

메타테이블은 객체가 수치 연산, 비교 연산, 이어붙이기, 길이 연산, 인덱싱을 어떻게 수행할지를 제어합니다. 또한 유저데이터나 테이블을 가비지 컬렉터가 수집할 때 호출되어야 할 함수를 정의합니다. Lua가 하나의 값에 대해 이러한 연산 중 하나를 수행할 때 이 값이 연산과 일치하는 이벤트를 포함하는 메타테이블을 가졌는지를 확인합니다. 만약 가지고 있다면 값과 연관된 키(메타메서드)는 해당 연산에서 어떻게 동작할지를 제어합니다. 이렇게 재정의되면 키와 연관된 값(메타메서드)은 해당 연산에 대한 Lua의 동작을 제어합니다.

메타테이블은 다음에 소개하는 연산을 제어하며 각각의 연산은 자신의 이름에 해당하는 동작을 수행합니다. 그리고 각 연산의 키는 이름에 해당하는 문자열과 두 개의 언더스코어('__')를 접두어로 사용합니다. 예를 들어 '더하기' 연산의 키는 "__add"입니다.

각 연산의 의미는 인터프리터에서 연산을 실행하는 Lua 함수를 설명하는 것이 이해하기가 더 쉬울 겁니다. 다음에 소개하는 Lua 코드는 설명을 위한 것이고 실제 동작은 인터프리터 안에 코드로 정의되어 있습니다(예제 코드는 글로 설명하는 것보다 더 효율적입니다). 이 설명에 사용되는 모든 함수들(rawget(), tonumber() 등)은 '6.1 기본 함수'에서 설명합니다. 특히 주어진 객체로부터 이벤트에 대한 메타메서드를 얻어오기 위해서는 다음 구문을 임시로 사용합니다.

```
metatable(obj)[event]
```

이 코드를 정식으로 사용하면 다음과 같은 코드가 됩니다.

```
rawget(getmetatable(obj) or {}, event)
```

앞 코드는 메타메서드의 접근이 또 다른 메타메서드의 동작을 일으키지 않음을 의미합니다. 또한 메타테이블이 없는 객체로 접근할 때 nil을 반환한다는 의미이기도 합니다.

단항 연산자인 '-'와 '#'의 메타메서드가 호출될 때는 두 번째 인자가 더미로 주어집니다. 이 여분의 인자는 Lua 내부에서만 사용되는 것으로, 이후 버전에서는 제거됩니다. 따라서 다음에 소개하는 코드부터는 생략했습니다(이 여분의 인자는 대부분 코드 실행과 무관합니다).

"add"
'+'(더하기) 연산.

다음 getbinhandler() 함수는 Lua에서 이항 연산binary operation을 선택할 때 어떠한 동작이 이루어지는지를 보여줍니다. Lua는 첫 번째 피연산자의 핸들러를 먼저 호출합니다. 만약 첫 번째 피연산자의 핸들러가 정의되어 있지 않으면 두 번째 피연산자의 핸들러를 호출합니다.

```
function getbinhandler(op1, op2, event)
  return metatable(op1)[event] or metatable(op2)[event]
end
```

이 getbinhandler() 함수를 사용해 'op1+op2'을 수행합니다.

```
function add_event(op1, op2)
  local o1, o2 = tonumber(op1), tonumber(op2)
  if o1 and o2 then          -- 두 피연산자가 모두 숫자인가?
    return o1 + o2           -- 여기의 '+'는 기본 '더하기' 연산이다
  else                       -- 적어도 피연산자 중 하나는 숫자가 아니다
    local h = getbinhandler(op1, op2, "__add")
```

```
    if h then                  -- 두 피연산자의 핸들러를 호출
      return (h(op1, op2))
    else                       -- 사용할 수 있는 핸들러가 없으면 기본 동작 수행
      error(…)
    end
  end
end
```

"sub"

'-'(빼기) 연산. add 연산과 비슷하게 동작합니다.

"mul"

'*'(곱하기) 연산. add 연산과 비슷하게 동작합니다.

"div"

'/'(나누기) 연산. add 연산과 비슷하게 동작합니다.

"mod"

'%'(나머지) 연산. add 연산과 비슷하게 동작합니다. 'o1-floor(o1/o2)*o2'라는 수식을 계산하는 것이 기본 동작입니다.

"pow"

'^'(지수) 연산. add 연산과 비슷하게 동작합니다. 기본적으로는 C의 math 라이브러리에 포함된 pow 함수를 실행해 동작합니다.

"unm"

'-'(음수) 단항 연산.

```
function unm_event(op)
  local o = tonumber(op)
  if o then        -- op 가 숫자인가?
    return -o      -- '-' 는 기본 음수 표현
  else             -- 피연산자가 숫자가 아니다
    -- 피연산자에서 핸들러를 얻는다
    local h = metatable(op).__unm
    if h then      -- 피연산자의 핸들러 호출
      return (h(op))
    else           -- 사용할 수 있는 핸들러가 없으면 기본 동작 수행
      error(…)
    end
  end
end
```

"concat"

'..'(이어붙이기) 연산.

```
function concat_event(op1, op2)
  if(type(op1) == "string" or type(op1) == "number") and
    (type(op2) == "string" or type(op2) == "number") then
    return op1 .. op2   -- 기본 문자열 연결
  else
    local h = getbinhandler(op1, op2, "__concat")
    if h then
      return (h(op1, op2))
    else
      error(…)
    end
  end
end
```

"len"

'#'(길이) 연산.

```
function len_event(op)
  if type(op) == "string" then
    return strlen(op)          -- 원본 문자열의 길이
  else
    local h = metatable(op).__len
    if h then
      return (h(op))           -- 연산자의 핸들러 호출
    elseif type(op) == "table" then
      return #op               -- 원본 테이블의 길이
    else                       -- 에러는 핸들러를 이용할 수 없다
      error(…)
    end
  end
end
```

'3.4.6 길이 연산자'의 테이블 길이에 대한 설명을 참조하세요.

"eq"

'==' 연산. getequalhandler() 함수는 Lua가 상등성을 확인하기 위해서 어떻게 메타메서드를 선택하는지를 정의합니다. 메타메서드는 두 값이 같은 타입이고 선택된 연산에 대해 같은 메타메서드를 사용할 때 선택됩니다. 또는 두 값이 서로 다른 테이블이거나 풀 유저데이터여야 합니다.

```
function getequalhandler(op1, op2)
  if type(op1) ~= type(op2) or
    (type(op1) ~= "table" and type(op1) ~= "userdata") then
    return nil    -- 별개의 값
```

```
  end
  local mm1 = metatable(op1).__eq
  local mm2 = metatable(op2).__eq
  if mm1 == mm2 then
    return mm1
  else
    return nil
  end
end
```

"eq" 이벤트는 다음처럼 정의합니다.

```
function eq_event(op1, op2)
  if op1 == op2 then   -- 원본이 같은가?
    return true        -- 값이 같다
  end
  -- 메타메서드를 확인한다
  local h = getequalhandler(op1, op2)
  if h then
    return not not h(op1, op2)
  else
    return false
  end
end
```

결과는 언제나 불리언 타입입니다.

"lt"

'<' 연산자.

```
function lt_event(op1, op2)
  if type(op1) == "number" and type(op2) == "number" then
    return op1 < op2   -- 숫자 비교
  elseif type(op1) == "string" and type(op2) == "string" then
    return op1 < op2   -- 사전적 비교
  else
    local h = getbinhandler(op1, op2, "__lt")
    if h then
      return not not h(op1, op2)
    else
      error(…)
    end
  end
end
```

결과는 언제나 불리언 타입입니다.

"le"

'<=' 연산자.

```
function le_event(op1, op2)
  if type(op1) == "number" and type(op2) == "number" then
    return op1 <= op2   -- 숫자 비교
  elseif type(op1) == "string" and type(op2) == "string" then
    return op1 <= op2   -- 사전적 비교
  else
    local h = getbinhandler(op1, op2, "__le")
    if h then
```

```
    return not not h(op1, op2)
  else
    h = getbinhandler(op1, op2, "__lt")
    if h then
      return not h(op2, op1)
    else
      error(…)
    end
  end
 end
end
```

"le" 메타메서드가 없다면 Lua는 "lt"를 사용하여 a <= b 형태를 not (a <= b) 형태로 변환합니다. 다른 비교 연산과 마찬가지로 결과는 항상 불리언 타입입니다.

"index"

table[key]. table의 key 값에 접근할 때 발생하는 이벤트입니다. 이 메타메서드는 테이블 안에 키가 존재하지 않을 때만 호출됩니다(인자 table이 테이블이 아닐 때는 키가 생성된 적이 없으므로 매번 메타메서드가 호출됩니다).

```
function gettable_event(table, key)
  local h
  if type(table) == "table" then
    local v = rawget(table, key)
    -- 키가 존재하면 원본 값을 반환한다
    if v ~= nil then
      return v
    end
    h = metatable(table).__index
    if h == nil then
      return nil
```

```
    end
  else
    h = metatable(table).__index
    if h == nil then
      error(…)
    end
  end
  if type(h) == "function" then
    return (h(table, key))    -- 핸들러를 호출
  else
    return h[key]             -- 혹은 연산을 반복
  end
end
```

"newindex"

table[key] = value. table의 특정 key에 값을 할당할 때 발생하는 이벤트입니다. 테이블에 키가 존재하지 않을 때만 메타메서드가 호출됩니다.

```
function settable_event(table, key, value)
  local h
  if type(table) == "table" then
    local v = rawget(table, key)
    -- 키가 존재하려면 해당 키에 값을 할당한다
    if v ~= nil then
      rawset(table, key, value);
      return
    end
    h = metatable(table).__newindex
    if h == nil then
      rawset(table, key, value);
      return
```

```
    end
  else
    h = metatable(table).__newindex
    if h == nil then
      error(…)
    end
  end
  if type(h) == "function" then
    h(table, key,value)   -- 핸들러를 호출
  else
    h[key] = value        -- 혹은 연산을 반복
  end
end
```

"call"

Lua가 값을 호출할 때 호출됩니다.

```
function function_event(func, ...)
  if type(func) == "function" then
    return func(...)   -- 기본 호출
  else
    local h = metatable(func).__call
    if h then
      return h(func, ...)
    else
      error(…)
    end
  end
end
```

2.5 가비지 컬렉션

Lua에서는 메모리 관리가 자동으로 이루어집니다. 즉, 새로운 객체를 생성하거나 해제하는 데 메모리 할당을 어떻게 할지 고민할 필요가 없다는 의미입니다. Lua의 메모리 관리는 가비지 컬렉터가 쓸모 없어진 객체를 수집하는 방식으로 동작합니다(쓸모가 없어진 객체라는 것은 더 이상 접근할 수 없는 개체라는 의미입니다). 문자열, 테이블, 유저데이터, 함수, 스레드, 내부 구조체 등 Lua에서 사용하는 모든 메모리는 자동으로 관리합니다.

Lua는 점진적 표시-제거mark-and-sweep 컬렉터를 사용하며, 가비지 컬렉터의 주기를 제어하기 위해 두 개의 숫자를 사용합니다. 두 개의 숫자는 각각 가비지 컬렉터의 수집 중지와 가비지 컬렉터 단계의 승수를 의미합니다. 두 값은 %에 기준을 둔 비율로 사용됩니다. 예를 들어 100은 내부적으로 1로 간주합니다.

가비지 컬렉터의 수집 중지를 제어한다는 의미는 새로운 수집 주기가 시작되기 전에 얼마 동안을 기다릴지 제어한다는 뜻입니다. 큰 값을 설정하면 가비지 컬렉터가 소극적으로 객체를 수집합니다. 값이 100보다 작으면 가비지 컬렉터가 새로운 주기가 시작되기 전에 대기 시간을 가지지 않습니다. 값이 200인 것은 이전보다 메모리가 두 배가 되기 전에는 객체 수집이 시작되지 않는다는 의미입니다.

가비지 컬렉터 단계 승수를 제어한다는 의미는 메모리 할당에 대한 상대적인 수집 속도를 제어한다는 의미입니다. 값이 클수록 적극적으로 객체를 수집하지만 증분 단계의 크기가 커집니다. 값이 100보다 작으면 가비지 컬렉터가 굉장히 느려지고 수집 주기도 끝나지 않을 수 있습니다. 기본값은 200이고 메모리 할당 속도보다 2배의 속도로 가비지 컬렉터가 동작함을 의미합니다.

만약 단계 승수를 아주 큰 숫자(프로그램이 사용할 수 있는 최대 바이트의 10% 이상)로 설정하면 가비지 컬렉터는 아마도 세계가 종말하는 날까지 동작하는 컬렉터

가 될 것입니다. 따라서 가비지 컬렉터 중지를 제어하는 숫자를 200으로 설정해 컬렉터는 예전 Lua 버전의 컬렉터와 동일하게 동작하게 해야 합니다. Lua의 메모리 사용량이 두 배가 될 때마다 메모리를 완벽히 수집할 것입니다.

그리고 C에서 lua_gc() 함수를 호출하거나 Lua에서 collectgarbage() 함수를 호출해 이 값을 변경할 수 있습니다. 또한 이 컬렉터를 직접 제어하기 위해 이 함수들을 사용할 수도 있습니다(예를 들어 멈추거나 다시 시작하는 동작을 수행합니다).

Lua 5.2에서는 실험적인 기능으로 가비지 컬렉터를 증분 모드가 아닌 세대 모드generational mode로 설정할 수 있습니다. 세대 컬렉터generational collector는 객체 대부분이 생성된 지 얼마 되지 않아 제거된다고 가정해 생성된 지 얼마 되지 않은 객체들을 우선 검사합니다. 이러한 동작은 가비지 컬렉터가 객체를 검사하는 시간을 최소화할 수 있지만 오래된 객체들이 누적되므로 메모리 사용량이 늘어납니다.

이러한 부작용을 완화하기 위해서 세대 컬렉터는 수시로 전체 수집을 수행합니다. 이 기능은 실험적인 기능이라는 걸 기억해야 합니다. 사용해보는 것은 좋지만 부작용을 고려해야 합니다.

2.5.1 가비지 컬렉션 메타메서드

테이블을 위한 가비지 컬렉션 메타메서드를 설정할 수 있습니다. C API를 사용하면 풀 유저데이터에도 설정할 수 있습니다(‘2.4 메타테이블과 메타메서드’ 참조). 이 메타메서드는 종결자finalizer를 호출해줍니다. 종결자는 Lua의 메모리 수집을 외부 리소스 관리를 통해 수행할 수 있도록 해줍니다(파일 닫기, 네트워크 혹은 데이터베이스 연결, 자체 메모리 해제 등).

객체(테이블, 유저데이터) 수집이 종결되려면 반드시 종결을 위한 표시를 해야 합니다. (객체에) 메타테이블을 설정했고 해당 메타테이블의 인덱스에 문자열 "__gc"인 필드가 있다면 해당 객체의 종결을 위해 표시를 해둡니다. 만약 메타테

이블에 "__gc" 필드가 없고 나중에 해당 필드를 추가하게 되면 객체는 종결을 위한 표시를 하지 않을 것입니다. 그러나 객체에 표시해둔 다음에는 "__gc" 필드를 자유롭게 변경할 수 있습니다.

표시된 객체가 쓸모 없게 되어도 가비지 컬렉터에 바로 수집되지는 않습니다. 대신 Lua는 해당 객체를 수집 목록에 넣습니다. 수집이 끝나면 Lua는 해당 목록를 순회하면서 다음 함수와 같은 동작을 수행합니다.

```
function gc_event(obj)
  local h = metatable(obj).__gc
  if type(h) == "function" then
    h(obj)
  end
end
```

실제로 매번 메모리 수집 주기가 끝날 때 해당 객체가 수집하려고 표시해둔 순서의 역순으로 객체의 종결자를 호출합니다. 이렇게 역순으로 호출하는 이유는 수집 과정이 진행되는 동안 처음으로 호출되는 종결자가 남아 있는, 다른 표시해둔 객체들과 연관되어 있을 수도 있기 때문입니다. 각각의 종결자의 호출은 표준 코드가 실행되는 도중 어떤 위치에서도 일어날 수 있습니다.

따라서 수집이 진행 중인 객체는 종결자가 사용하는 중이므로 해당 객체(해당 객체를 통해서만 접근할 수 있는 다른 객체)는 Lua에 의해 되살려질 것입니다. 일반적으로 이렇게 되살려지는 객체는 짧은 시간 동안만 유지되며, 객체의 메모리는 다음 가비지 컬렉션 주기에 해제됩니다. 하지만 만약 종결자가 객체를 어떤 전역 공간에 저장한다면(예를 들어 전역 변수 같은 곳) 객체는 영구적으로 되살릴 수 있습니다. 어쨌든 객체의 메모리는 더 이상 접근할 방법이 없을 때 해제되며 종결자는 절대 두 번 호출되지 않습니다.

Lua 상태가 닫힐 때('lua_close()' 참조) Lua는 표시해둔 모든 객체의 종결자를 표시해둔 순서의 반대로 호출합니다. 만약 종결자가 이 단계에 새로운 객체를 표시한다면 이 새 객체는 종결되지 않을 것입니다.

2.5.2 약한 참조 테이블

약한 참조 테이블은 모든 요소가 약한 참조인 테이블입니다. 그리고 약한 참조 상태는 가비지 컬렉터가 객체를 참조한다고 간주하지 않습니다. 다시 말해 객체에 대한 참조가 약한 참조뿐이면 가비지 컬렉터는 해당 객체를 참조하는 것이 없다고 간주하고 해당 객체를 수집할 것입니다. 즉, 약한 참조는 객체의 참조 카운팅에 포함되지 않습니다.

약한 참조 테이블은 약한 키와 약한 값을 가집니다. 약한 키를 가진 테이블은 해당 키의 수집을 허용하지만 값을 수집하는 일은 방지합니다. 또한 약한 키와 약한 값으로 이루어진 테이블은 키와 값이 수집되는 것을 허용합니다. 키나 값이 수집되면 모든 쌍이 테이블에서 제거됩니다.

테이블의 약한 참조 설정은 메타테이블의 "__mode" 필드로 제어합니다. 만약 "__mode" 필드가 문자 k를 포함하고 있으면 테이블의 키key가 약한 참조가 됩니다. "__mode" 필드가 문자 v를 포함한다면 테이블의 값이 약한 참조가 됩니다.

약한 키와 강한 값을 가지는 테이블은 하루살이ephemeron 테이블이라고 부릅니다. 하루살이 테이블의 값에 접근할 수 있으려면 키에 접근할 수 있어야 합니다. 특히 값을 참조하는 유일한 통로가 키뿐일 때는 해당 쌍이 제거됩니다.

테이블의 약함 설정을 변경하면 효과는 다음 메모리 수집 주기에 영향을 줍니다. 특히 강한 참조로 변경된 경우에 Lua의 수집은 해당 변경이 효과를 발휘하기 전까지는 계속 수집을 수행할 것입니다.

객체가 명시적으로 생성자를 가진 경우에만 약한 테이블에서 제거될 수 있습니다. 수치나 간단한 C 함수 같은 값은 메모리 수집 대상이 되지 않습니다. 따라서 연관된 값이 수집되기 전까지는 약한 테이블에서 제거되지 않습니다. 문자열은 메모리 수집 대상이 되지만 명시적인 생성자를 가지지 않기 때문에 약한 테이블에서 제거되지 않습니다.

소멸 중인 객체나 소멸 중인 객체를 통해 접근하는 객체 같은 되살려진 객체들은 약한 테이블 안에서 특수한 동작을 하게 되는데, 이런 객체들은 소멸자가 실행되기 전에 테이블에서 제거됩니다. 하지만 약한 키 때문에 제거되는 객체는 소멸자가 실행된 이후여야만 다음 수집에서 제거됩니다. 이는 소멸자가 약한 테이블을 통해 객체에 접근하는 것을 가능하게 합니다.

만약 수집 주기가 진행되는 동안 약한 테이블에서 객체를 되살리면 다음 주기가 올 때까지 객체는 제거되지 않습니다.

2.6 코루틴

Lua는 코루틴coroutine을 지원합니다(코루틴은 협력적인 멀티스레딩이라 불리기도 합니다). Lua의 코루틴은 스레드와는 독립적으로 실행됩니다. 따라서 시스템에서 사용하는 스레드와는 달리 코루틴은 명시적으로 yield() 함수를 호출할 때만 실행이 중단됩니다.

코루틴을 생성하려면 coroutine.create() 함수를 호출해야 하며 인자는 코루틴의 메인 함수로 사용될 함수 하나만 받습니다. coroutine.create() 함수는 하나의 새로운 코루틴을 생성하고 생성된 코루틴의 핸들(스레드 타입의 객체)을 반환합니다. 단, 생성과 동시에 코루틴이 시작되지는 않습니다.

코루틴을 실행하려면 coroutine.resume() 함수를 호출하면 됩니다. 함수를 호출하면 처음으로 coroutine.resume() 함수를 호출할 때 coroutine.create() 함수

에서 반환한 스레드 객체를 첫 번째 인자로 전달합니다. 그러면 코루틴은 자신의 메인 함수 첫 번째 줄부터 실행을 시작합니다. coroutine.resume() 함수로 전달된 나머지 인자들은 메인 함수의 인자로 전달됩니다. 코루틴이 시작되면 실행이 종료되거나 coroutine.yield() 함수가 호출될 때까지 멈추지 않고 실행됩니다.

코루틴을 종료하는 방법은 두 가지입니다. 첫 번째(일반적)는 메인 함수에서 (마지막 명령을 처리한 후 명시적이거나 묵시적으로) 반환이 일어날 때고, 두 번째는 내부에서 보호되지 않은 에러가 발생하는 상황입니다. 첫 번째는 coroutine.resume() 함수가 true라는 값을 반환함과 동시에 코루틴 메인 함수가 값들을 전달합니다. 두 번째는 coroutine.resume() 함수가 false와 함께 에러 메시지를 반환합니다.

코루틴에서 coroutine.yield() 함수를 호출하는 것으로 양도yield를 할 수 있습니다. 코루틴이 양도를 하게 되면 코루틴을 실행했던 coroutine.resume() 함수는 즉시 반환을 수행하게 됩니다. 심지어 내장 함수 안에서 양도가 일어나더라도 똑같은 동작을 수행합니다(내장 함수란 코루틴의 메인 함수가 아닌 다른 메인 함수에서 호출하거나 호출된 함수 내부에서 호출한 함수를 말합니다). 양도가 일어나면 coroutine.resume() 함수는 true 값을 반환하면서 coroutine.yield() 함수로 전달된 값들을 전달합니다. 같은 코루틴을 다음에 다시 시작할 때는 마지막으로 양도가 일어났던 지점에서 실행이 시작되고 coroutine.resume() 함수를 호출할 때 전달된 추가 인자들이 coroutine.yield() 함수에서 반환됩니다.

coroutine.create() 함수처럼 coroutine.wrap() 함수 또한 코루틴을 생성합니다. 하지만 코루틴의 핸들을 반환하는 것이 아니라 coroutine.resume() 함수를 호출했을 때 반환되는 값을 반환합니다. 또한 coroutine.wrap() 함수를 호출할 때 전달된 인자들은 coroutine.resume() 함수를 호출할 때의 추가 인자로 전달되며 coroutine.wrap() 함수는 coroutine.resume() 함수가 반환하는 true, false 값

을 제외한 나머지 인자들을 모두 반환합니다. 즉, coroutine.resume() 함수와는 달리 coroutine.wrap() 함수는 에러를 받지 않고 호출자에게 에러를 전달합니다.

그럼 코루틴이 어떻게 동작하는지에 대한 예제 코드를 살펴보겠습니다.

```
function foo(a)
  print("foo", a)
  return coroutine.yield(2*a)
end

co = coroutine.create(function (a,b)
  print("co-body", a, b)
  local r = foo(a+1)
  print("co-body", r)
  local r, s = coroutine.yield(a+b, a-b)
  print("co-body", r, s)
  return b, "end"
end)

print("main", coroutine.resume(co, 1, 10))
print("main", coroutine.resume(co, "r"))
print("main", coroutine.resume(co, "x", "y"))
print("main", coroutine.resume(co, "x", "y"))
```

위 코드를 실행하면 다음과 같은 출력 결과를 확인할 수 있습니다.

```
co-body 1       10
foo     2
main    true    4
co-body r
main    true    11      -9
co-body x       y
```

```
main    true    10      end
main    false   cannot resume dead coroutine
```

마지막으로 C API를 이용하면 코루틴을 생성하거나 조작이 가능하다는 사실을 기억하기 바랍니다('lua_newthread(), lua_resume(), lua_yield()'를 참조하세요).