자주 묻는 질문(FAQ)

목차

출처

프로젝트의 목적은 무엇인가요?

불과 10년 전인 Go가 처음 시작될 당시 프로그래밍 세계는 지금과는 달랐습니다. 프로덕션 소프트웨어는 보통 C++ 또는 Java로 작성되었고, GitHub는 존재하지 않았으며, 대부분의 컴퓨터는 아직 멀티프로세서가 아니었고, 비주얼 스튜디오와 이클립스 외에는 인터넷에서 무료로 사용할 수 있는 IDE나 기타 고급 도구가 거의 없었을 때였습니다.

한편, 서버 소프트웨어를 개발하기 위해 사용하는 언어가 지나치게 복잡하다는 사실에 좌절감을 느꼈습니다. C, C++, Java와 같은 언어가 처음 개발된 이후 컴퓨터는 엄청나게 빨라졌지만 프로그래밍 자체는 그다지 발전하지 못했습니다. 또한 멀티프로세서가 보편화되고 있는 것은 분명했지만 대부분의 언어는 이를 효율적이고 안전하게 프로그래밍하는 데 거의 도움을 주지 못했습니다.

저희는 한 발 물러서서 기술 발전에 따라 앞으로 몇 년 동안 소프트웨어 엔지니어링을 지배할 주요 이슈가 무엇인지, 그리고 새로운 언어가 이러한 문제를 해결하는 데 어떻게 도움이 될 수 있을지 생각해보기로 했습니다. 예를 들어, 멀티코어 CPU의 등장으로 언어가 일종의 동시성 또는 병렬성을 최고 수준으로 지원해야 한다는 주장이 제기되었습니다. 또한 대규모 동시 프로그램에서 리소스 관리를 추적 가능하게 만들려면 가비지 컬렉션 또는 최소한 안전한 자동 메모리 관리가 필요했습니다.

이러한 고려 사항들은 일련의 토론으로 이어졌고, 그 결과 Go는 처음에는 아이디어와 요구사항의 집합으로, 그다음에는 언어로서 발전했습니다. 가장 중요한 목표는 도구를 활성화하고, 코드 서식 지정과 같은 일상적인 작업을 자동화하고, 대규모 코드 베이스 작업의 장애물을 제거하여 프로그래머가 작업하는 데 더 많은 도움을 주는 것이었습니다.

Go의 목표와 그 목표에 도달하거나 최소한 접근하는 방법에 대한 훨씬 더 광범위한 설명은 Go at Google 문서에서 확인할 수 있습니다: 소프트웨어 엔지니어링 서비스에서의 언어 설계.

프로젝트의 역사는 어떻게 되나요?

로버트 그리세머, 롭 파이크, 켄 톰슨은 2007년 9월 21일 화이트보드에 새로운 언어에 대한 목표를 스케치하기 시작했습니다. 며칠이 지나지 않아 목표는 무언가를 하겠다는 계획과 그것이 무엇인지에 대한 대략적인 아이디어로 정리되었습니다. 디자인은 관련 없는 업무와 병행하여 파트타임으로 계속했습니다. 2008년 1월, Ken은 아이디어를 탐색할 수 있는 컴파일러 작업을 시작했고, 그 결과물로 C 코드를 생성했습니다. 그해 중반에는 이 언어가 풀타임 프로젝트가 되었고, 프로덕션 컴파일러를 시도할 수 있을 만큼 안정화되었습니다. 2008년 5월, 이안 테일러는 초안 사양을 사용하여 Go용 GCC 프론트엔드 개발을 독립적으로 시작했습니다. 2008년 말에는 러스 콕스가 합류하여 언어와 라이브러리를 프로토타입에서 현실로 옮기는 데 도움을 주었습니다.

Go는 2009년 11월 10일에 공개 오픈 소스 프로젝트가 되었습니다. 커뮤니티의 수많은 사람들이 아이디어, 토론, 코드에 기여해 왔습니다.

현재 전 세계에는 수백만 명의 바둑 프로그래머, 즉 고퍼가 존재하며 매일 그 수가 늘어나고 있습니다. 바둑의 성공은 우리의 예상을 훨씬 뛰어넘었습니다.

고퍼 마스코트의 유래는 무엇인가요?

마스코트와 로고는 플랜 9 토끼 글렌다를 디자인한 르네 프렌치가 디자인했습니다. 고퍼에 대한 블로그 게시물에서는 몇 년 전 그녀가 WFMU 티셔츠 디자인에 사용했던 고퍼에서 고퍼가 어떻게 파생되었는지 설명합니다. 로고와 마스코트는 크리에이티브 커먼즈 저작자표시 4.0 라이선스의 적용을 받습니다.

고퍼의 특징과 이를 올바르게 표현하는 방법을 설명하는 모델 시트가 있습니다. 이 모델 시트는

2016년 고퍼콘에서 르네의 강연에서 처음 공개되었습니다. 고퍼는 일반적인 고퍼가 아닌 *고퍼* 고퍼만의 독특한 특징을 가지고 있습니다.

언어가 Go인가요, 골랑인가요?

이 언어는 Go라고 불립니다. "golang"이라는 명칭은 원래 웹사이트가 *golang.org였기* 때문에 생겨났습니다. (당시에는 *.dev* 도메인이 없었습니다.) 하지만 많은 사람들이 골랑이라는 이름을 사용하고 있으며, 이는 라벨로 사용하기에 편리합니다. 예를 들어, 이 언어의 트위터 태그는 "*#golang*"입니다. 언어의 이름은 이와 상관없이 그냥 Go입니다.

참고: 공식 로고에는 대문자 두 개가 있지만, 언어 이름은 GO가 아닌 Go로 표기되어 있습니다.

새로운 언어를 만든 이유는 무엇인가요?

Go는 Google에서 수행하던 업무에 대한 기존 언어와 환경에 대한 불만에서 탄생했습니다. 프로그래밍이 너무 어려워졌고 언어 선택이 부분적으로 원인이었습니다. 효율적인 컴파일, 효율적인 실행, 프로그래밍의 용이성 중 하나를 선택해야 했는데, 이 세 가지 모두 동일한 주류 언어로 제공되지 않았습니다.
프로그래머들은 안전성과 효율성보다는 편의성을 선택해 C++나 Java 대신 Python, JavaScript와 같은 동적 유형 언어로 전환했습니다.

저희만의 우려는 아니었습니다. 수년 동안 프로그래밍 언어에 대한 조용한 환경이 지속되던 중, 고(Go)는 러스트, 엘릭서, 스위프트 등 여러 새로운 언어 중 가장 먼저 프로그래밍 언어 개발이 다시 활발해지고 거의 주류가 된 분야 중 하나였습니다.

Go는 해석되고 동적으로 타입이 지정된 언어의 프로그래밍 용이성과 정적으로 타입이 지정된 컴파일된 언어의 효율성 및 안전성을 결합하여 이러한 문제를 해결하고자 했습니다. 또한 네트워크 및 멀티코어 컴퓨팅을 지원하여 현대성을 지향했습니다. 마지막으로, Go는 한 대의 컴퓨터에서 대용량 실행 파일을 빌드하는 데 최대 몇 초밖에 걸리지 않아야 하는 *빠른* 작업 속도를 목표로 합니다. 이러한 목표를 달성하려면 표현력이 풍부하면서도 가벼운 타입 시스템, 동시성 및 가비지 컬렉션, 엄격한 종속성 사양 등 여러 가지 언어적 문제를 해결해야 했습니다. 라이브러리나 도구로는 이러한 문제를 해결할 수 없었기 때문에 새로운 언어가 필요했습니다.

Google의 Go 문서에서는 Go 언어의 설계 배경과 동기에 대해 설명하며, 이 FAQ에 제시된 많은 답변에 대해 자세히 설명합니다.

바둑의 조상은 무엇인가요?

Go는 대부분 C 계열(기본 구문)에 속하며, 파스칼/모듈라/오베론 계열(선언, 패키지)에서 상당한 영향을 받았으며, 뉴스케이크와 림보(동시성)와 같이 토니 호어의 CSP에서 영감을 받은 언어에서 일부 아이디어를 얻었습니다. 그러나 전반적으로 새로운 언어입니다. 모든 면에서 이 언어는 프로그래머가 하는 일과 적어도 우리가 하는 프로그래밍을 더 효과적으로, 즉 더 재미있게 만들 수 있는 방법에 대해 고민하여 설계되었습니다.

디자인의 기본 원칙은 무엇인가요?

Go가 설계되었을 때, 적어도 Google에서는 서버를 작성하는 데 가장 일반적으로 사용되는 언어가 Java와 C++였습니다. 이러한 언어는 너무 많은 장부와 반복이 필요하다고 생각했습니다. 일부 프로그래머는 효율성과 유형 안전성을 희생하는 대신 Python과 같은 보다 역동적이고 유동적인 언어로 전환했습니다. 저희는 단일 언어에서 효율성, 안전성, 유동성을 모두 갖출 수 있어야 한다고 생각했습니다.

Go는 두 가지 의미에서 타이핑의 양을 줄이려고 노력합니다. 디자인 전반에 걸쳐 군더더기와 복잡성을 줄이려고 노력했습니다. 포워드 선언이나 헤더 파일이 없으며 모든 것이 정확히 한 번만 선언됩니다. 초기화는 표현력이 풍부하고 자동적이며 사용하기 쉽습니다. 구문이 깔끔하고 키워드 사용량이 적습니다. 반복(`foo.Foo* myFoo = new(foo.Foo)`)은 `:=` 선언 및-를 사용한 간단한 유형 파생으로 감소합니다.

를 초기화합니다. 그리고 가장 근본적으로는 타입 계층 구조가 없다는 점입니다. 타입은 그냥 있는 *그대로* 존재하며, 관계를 알릴 필요가 없습니다. 이러한 단순화 덕분에 Go는 정교함을 희생하지 않고도 표현력이 풍부하면서도 이해하기 쉽습니다.

또 다른 중요한 원칙은 개념을 직교로 유지하는 것입니다. 메서드는 모든 유형에 대해 구현할 수 있으며, 구조체는 데이터를 나타내고 인터페이스는 추상화를 나타내는 등 다양한 방식으로 구현할 수 있습니다. 직교성을 유지하면 사물이 결합할 때 어떤 일이 일어나는지 더 쉽게 이해할 수 있습니다.

사용법

Google은 내부적으로 Go를 사용하나요?

예. Go는 Google 내부 프로덕션에서 널리 사용됩니다. 한 가지 쉬운 예로 golang.org 뒤에 있는 서버를 들 수 있습니다. 이 서버는 구글 앱 엔진에서 프로덕션 구성으로 실행되는 `godoc` 문서 서버일 뿐입니다.

더 중요한 사례는 Chrome 바이너리와 `apt-get` 패키지와 같은 기타 대용량 설치 파일을 제공하는 Google의 다운로드 서버인 `dl.google.com`입니다.

Go가 Google에서 사용되는 유일한 언어는 아니지만, 사이트 안정성 엔지니어링(SRE) 및 대규모 데이터 처리를 비롯한 여러 분야에서 핵심적인 언어입니다.

다른 어떤 회사에서도 Go를 사용하나요?

Go의 사용은 전 세계적으로 증가하고 있으며, 특히 클라우드 컴퓨팅 공간에서만 사용되는 것은 아닙니다. Go로 작성된 주요 클라우드 인프라 프로젝트로는 Docker와 Kubernetes가 있지만, 그 외에도 더 많은 프로젝트가 있습니다.

하지만 클라우드뿐만이 아닙니다. Go 위키에는 정기적으로 업데이트되는 Go를 사용하는 많은 회사의 목록이 있는 페이지가 있습니다.

위키에는 이 언어를 사용하는 회사 및 프로젝트의 성공 사례에 대한 링크가 있는 페이지도 있습니

다.

바둑 프로그램은 C/C++ 프로그램과 연동되나요?

동일한 주소 공간에서 C와 Go를 함께 사용할 수는 있지만, 이는 자연스럽지 않으며 특별한 인터페이스 소프트웨어가 필요할 수 있습니다. 또한 C를 Go 코드와 연결하면 Go가 제공하는 메모리 안전성과 스택 관리 속성을 포기해야 합니다. 때때로 문제를 해결하기 위해 C 라이브러리를 사용해야 하는 경우가 있지만, 그렇게 하면 항상 순수한 Go 코드에는 없는 위험 요소가 발생하므로 신중하게 사용해야 합니다.

Go에서 C를 사용해야 하는 경우 진행 방법은 Go 컴파일러 구현에 따라 다릅니다. Go 팀에서 지원하는 Go 컴파일러 구현은 세 가지가 있습니다. 기본 컴파일러인 gc, GCC 백엔드를 사용하는 gccgo, LLVM 인프라를 사용하는 다소 덜 성숙한 gollvm이 그것입니다.

Gc는 C와 다른 호출 규칙과 링커를 사용하므로 C 프로그램에서 직접 호출할 수 없으며 그 반대의 경우도 마찬가지입니다. cgo 프로그램은

"외부 함수 인터페이스"를 통해 Go 코드에서 C 라이브러리를 안전하게 호출할 수 있습니다. SWIG는 이 기능을 C++ 라이브러리로 확장합니다.

`cgo` 및 SWIG를 `Gccgo` 및 `gollvm`과 함께 사용할 수도 있습니다. 이러한 컴파일러는 기존 API를 사용하기 때문에 세심한 주의를 기울이면 이러한 컴파일러의 코드를 GCC/LLVM으로 컴파일된 C 또는 C++ 프로그램과 직접 연결하는 것도 가능합니다. 하지만 이렇게 안전하게 연결하려면 관련 언어의 호출 규칙을 모두 이해하고 있어야 하며, Go에서 C 또는 C++를 호출할 때 스택 제한에 대한 고려가 필요합니다.

Go는 어떤 IDE를 지원하나요?

Go 프로젝트에는 사용자 지정 IDE가 포함되어 있지 않지만, 언어와 라이브러리는 소스 코드를 쉽게 분석할 수 있도록 설계되었습니다. 그 결과, 대부분의 유명 편집기와 IDE는 직접 또는 플러그인을 통해 Go를 잘 지원합니다.

Go를 잘 지원하는 잘 알려진 IDE 및 편집기 목록에는 Emacs, Vim, VSCode, Atom, 이클립스, Sublime, IntelliJ(Goland라는 사용자 지정 변형을 통해) 등이 있습니다. 여러분이 가장 선호하는 환경이 Go 프로그래밍을 위한 생산적인 환경일 가능성이 높습니다.

Go는 Google의 프로토콜 버퍼를 지원하나요?

별도의 오픈 소스 프로젝트에서 필요한 컴파일러 플러그인과 라이브러리를 제공합니다. github.com/golang/protobuf/에서 확인할 수 있습니다.

Go 홈 페이지를 다른 언어로 번역할 수 있나요?

물론이죠. Google은 개발자가 자신의 언어로 Go 언어 사이트를 만들 것을 권장합니다. 그러나 사이트에 Google 로고나 브랜딩을 추가하기로 선택한 경우(golang.org에는 표시되지 않음) www.google.com/permissions/guidelines.html 의 가이드라인을 준수해야 합니다.

디자인

Go에 런타임이 있나요?

Go에는 모든 Go 프로그램의 일부인 *런타임이라고 하는* 광범위한 라이브러리가 있습니다. 런타임 라이브러리는 가비지 컬렉션, 동시성, 스택 관리 및 기타 Go 언어의 중요한 기능을 구현합니다. 언어의 중심이라고 할 수 있지만, Go의 런타임은 C 라이브러리인 `libc`와 유사합니다.

그러나 Go의 런타임에는 Java 런타임에서 제공하는 것과 같은 가상 머신이 포함되어 있지 않다는 점을 이해하는 것이 중요합니다. Go 프로그램은 네이티브 머신 코드(또는 일부 변형 구현의 경우 JavaScript 또는 WebAssembly)로 미리 컴파일됩니다. 따라서 이 용어는 프로그램이 실행되는 가상 환경을 설명하는 데 자주 사용되지만, Go에서 "런타임"이라는 단어는 중요한 언어 서비스를 제공하는 라이브러리에 부여된 이름일 뿐입니다.

유니코드 식별자란 무엇인가요?

Go를 설계할 때 저희는 지나치게 ASCII 중심적이지 않도록 하고 싶었고, 이는 7비트 ASCII의 한계에서 식별자의 공간을 확장하는 것을 의미했습니다. 식별자는 유니코드에 정의된 문자 또는 숫자여야 한다는 Go의 규칙은 이해하고 구현하기는 쉽지만 제한이 있습니다. 예를 들어, 문자를 결합하는 것은 설계상 제외되어 있으며 데바나가리와 같은 일부 언어는 제외됩니다.

이 규칙에는 또 한 가지 안타까운 결과가 있습니다. 내보낸 식별자는 대문자로 시작해야 하므로 일부 언어의 문자로 만든 식별자는 다음과 같이 될 수 있습니다.
정의를 내보낼 수 없습니다. 현재로서는 X日本語와 같은 것을 사용하는 것이 유일한 해결책입니다.
는 분명 만족스럽지 않습니다.

언어의 초기 버전부터 다른 모국어를 사용하는 프로그래머를 수용하기 위해 식별자 공간을 확장하는 최선의 방법에 대해 많은 고민이 있었습니다. 정확히 어떻게 해야 하는지는 여전히 활발한 논의의 주제이며, 향후 버전의 언어에서는 식별자의 정의가 더 자유로워질 수 있습니다. 예를 들어, 식별자에 대한 유니코드 조직의 권장 사항 중 일부 아이디어를 채택할 수도 있습니다.
어떤 일이 발생하든 대소문자가 식별자의 가시성을 결정하는 방식을 유지(또는 확장)하면서 호환 가능하게 수행해야 하며, 이는 Go에서 가장 좋아하는 기능 중 하나입니다.

당분간은 모호한 식별자를 허용하는 규칙으로 인해 발생할 수 있는 버그를 방지하면서 프로그램을 손상시키지 않고 나중에 확장할 수 있는 간단한 규칙을 마련했습니다.

Go에 기능 X가 없는 이유는 무엇인가요?

모든 언어에는 새로운 기능이 포함되어 있고 누군가가 좋아하는 기능은 생략되어 있습니다. Go는 프로그래밍의 용이성, 컴파일 속도, 개념의 직교성, 동시성 및 가비지 컬렉션과 같은 기능 지원의 필요성을 염두에 두고 설계되었습니다. 컴파일 속도나 설계의 명확성에 영향을 미치거나 기본 시스템 모델을 너무 어렵게 만들 수 있기 때문에 선호하는 기능이 누락될 수 있습니다.

Go에 *X* 기능이 없는 것이 마음에 걸리신다면, 저희를 용서해 주시고 Go에 있는 기능들을 살펴보세요. *X의* 부재를 흥미로운 방식으로 보완하는 기능을 발견할 수도 있습니다.

Go는 언제 일반 유형이 생겼나요?

Go 1.18 릴리스에서는 언어에 유형 매개변수가 추가되었습니다. 이를 통해 다형성 또는 일반 프로

그래밍의 한 형태가 가능해졌습니다. 자세한 내용은 언어 사양 및 제안서를 참조하세요.

Go가 처음에 일반 유형 없이 출시된 이유는 무엇인가요?

Go는 시간이 지나도 유지보수가 용이한 서버 프로그램을 작성하기 위한 언어로 고안되었습니다. (자세한 배경은 이 문서를 참조하세요.) 확장성, 가독성, 동시성 같은 요소에 집중하여 설계되었습니다. 다형성 프로그래밍은 당시 언어의 목표에 필수적이지 않아 보였기 때문에 처음에는 단순성을 위해 제외되었습니다.

제네릭은 편리하지만 유형 시스템과 런타임이 복잡해지는 대가가 따릅니다. 복잡성에 비례하는 가치를 제공하는 디자인을 개발하는 데 시간이 오래 걸렸습니다.

Go에 예외가 없는 이유는 무엇인가요?

'try-catch-finally'라는 관용구처럼 예외를 제어 구조에 연결하면 코드가 복잡해집니다. 또한 프로그래머가 파일 열기 실패와 같은 일반적인 오류를 너무 많이 예외로 분류하는 경향이 있습니다.

Go는 다른 접근 방식을 취합니다. 일반적인 오류 처리의 경우, Go의 다중 값 반환을 사용하면 반환값에 과부하를 주지 않고도 오류를 쉽게 보고할 수 있습니다. 표준 에러 유형은 Go의 다른 기능과 결합되어 에러 처리를 편리하게 만들지만 다른 언어의 에러 처리와는 상당히 다릅니다.

또한 Go에는 정말 예외적인 상황에서 신호를 보내고 복구할 수 있는 몇 가지 내장 함수가 있습니다. 복구 메커니즘은 오류 발생 후 함수의 상태가 해체될 때만 실행되며, 이는 재앙을 처리하기에 충분하지만 추가 제어 구조가 필요하지 않고 잘 사용하면 깔끔한 오류 처리 코드를 생성할 수 있습니다.

자세한 내용은 지연, 패닉 및 복구 문서를 참조하세요. 또한 오류는 값입니다 블로그 게시물에서는 오류는 값에 불과하기 때문에 오류 처리에 Go의 모든 기능을 활용할 수 있다는 것을 보여줌으로써 Go에서 오류를 깔끔하게 처리하는 한 가지 접근 방식을 설명합니다.

Go에 어설션이 없는 이유는 무엇인가요?

Go는 어설션을 제공하지 않습니다. 이는 분명 편리한 기능이지만, 프로그래머가 적절한 오류 처리 및 보고에 대해 생각하지 않고 버팀목으로 사용하는 경우가 많았습니다. 적절한 오류 처리란 치명적이지 않은 오류가 발생한 후에도 서버가 중단되지 않고 계속 작동하는 것을 의미합니다. 적절한 오류 보고는 오류가 직접적이고 요점만 전달되므로 프로그래머가 대규모 크래시 추적을 해석하는 수고를 덜 수 있습니다. 정확한 오류는 오류를 발견한 프로그래머가 코드에 익숙하지 않은 경우 특히 중요합니다.

이것이 논쟁의 여지가 있다는 것을 잘 알고 있습니다. 바둑 언어와 라이브러리에는 현대의 관행과 다른 점이 많으며, 때로는 다른 접근 방식을 시도해 볼 가치가 있다고 생각하기 때문입니다.

CSP의 아이디어로 동시성을 구축하는 이유는 무엇인가요?

동시성 및 다중 스레드 프로그래밍은 시간이 지남에 따라 어렵다는 평판을 얻었습니다. 이는 부분적으로는 pthread와 같은 복잡한 설계와 뮤텍스, 조건 변수, 메모리 배리어와 같은 저수준 세부 사항을 지나치게 강조했기 때문이라고 생각합니다. 상위 레벨 인터페이스를 사용하면 뮤텍스 등이 숨겨져 있더라도 훨씬 더 간단한 코드를 작성할 수 있습니다.

동시성을 위한 높은 수준의 언어 지원을 제공하는 가장 성공적인 모델 중 하나는 Hoare의 순차적 프로세스(CSP)에서 비롯되었습니다. Occam과 Erlang은 CSP에서 파생된 잘 알려진 두 가지 언어입니다. Go의 동시성 프리미티브는 가계도의 다른 부분에서 파생되며, 주요 기여는 다음과 같은 강력한 개념입니다.

채널을 첫 번째 클래스 객체로 사용할 수 있습니다. 이전의 여러 언어에 대한 경험에 따르면 CSP 모델은 절차적 언어 프레임워크에 잘 맞는 것으로 나타났습니다.

왜 스레드 대신 고루틴을 사용하나요?

고루틴은 동시성을 쉽게 사용할 수 있게 해주는 기능 중 하나입니다. 한동안 사용되어 온 이 아이디어는 독립적으로 실행되는 함수, 즉 코루틴을 스레드 집합에 다중화하는 것입니다. 코루틴이 차단 시스템 호출을 호출하는 등의 이유로 차단되면 런타임은 동일한 운영 체제 스레드에 있는 다른 코루틴을 자동으로 실행 가능한 다른 스레드로 이동하여 차단되지 않도록 합니다. 프로그래머는 이 사실을 전혀 알 수 없다는 것이 핵심입니다. 고루틴이라고 부르는 고루틴은 스택 메모리를 제외하고는 몇 킬로바이트에 불과한 오버헤드가 거의 없기 때문에 매우 저렴할 수 있습니다.

스택을 작게 만들기 위해 Go의 런타임은 크기 조정이 가능한 경계 스택을 사용합니다. 새로 생성되는 고루틴에는 몇 킬로바이트가 주어지는데, 이는 거의 항상 충분합니다. 그렇지 않은 경우 런타임은 스택을 저장하기 위한 메모리를 자동으로 늘리거나 줄여서 많은 고루틴이 적당한 양의 메모리에서 살 수 있도록 합니다. CPU 오버헤드는 함수 호출당 평균적으로 약 3개의 저렴한 명령어를 사용합니다. 동일한 주소 공간에 수십만 개의 고루틴을 생성하는 것이 실용적입니다. 고루틴이 스레드에 불과하다면 시스템 리소스는 훨씬 적은 수로 부족할 것입니다.

지도 연산이 원자적으로 정의되지 않은 이유는 무엇인가요?

오랜 논의 끝에 지도의 일반적인 사용에는 여러 고루틴의 안전한 액세스가 필요하지 않으며, 필요한 경우에도 지도는 이미 동기화된 더 큰 데이터 구조 또는 계산의 일부일 가능성이 높다는 결론을 내렸습니다. 따라서 모든 맵 연산에 뮤텍스를 가져오도록 요구하면 대부분의 프로그램 속도가 느려지고 일부 프로그램에만 안전성이 추가됩니다. 그러나 제어되지 않은 맵 액세스가 프로그램을 중단시킬 수 있기 때문에 이는 쉬운 결정이 아니었습니다.

이 언어는 원자 지도 업데이트를 배제하지 않습니다. 신뢰할 수 없는 프로그램을 호스팅할 때와 같이 필요한 경우 구현에서 맵 액세스를 연동할 수 있습니다.

맵 액세스는 업데이트가 발생할 때만 안전하지 않습니다. 모든 고루틴이 범위 루프를 사용하여 맵의 요소를 반복하는 등 읽기만 하고 요소에 할당하거나 삭제를 수행하여 맵을 변경하지 않는 한, 동

기화 없이 동시에 맵에 액세스해도 안전합니다.

올바른 맵 사용을 돕기 위해 일부 언어 구현에는 동시 실행으로 인해 맵이 안전하지 않게 수정될 경우 런타임에 자동으로 보고하는 특수 검사가 포함되어 있습니다.

언어 변경을 수락하시겠습니까?

사람들은 종종 언어 개선을 제안하지만(메일링 리스트에는 이러한 논의의 풍부한 역사가 담겨 있습니다), 이러한 변경 사항 중 수용된 것은 거의 없습니다.

Go는 오픈 소스 프로젝트이지만, 언어와 라이브러리는 적어도 기존 프로그램을 깨는 변경을 방지하는 호환성 약속에 의해 보호됩니다.

소스 코드 수준(프로그램을 최신 상태로 유지하기 위해 때때로 다시 컴파일해야 할 수도 있음). 제안이 Go 1 사양을 위반하는 경우, 제안의 장점과 상관없이 해당 아이디어를 검토할 수 없습니다. 향후 Go의 주요 릴리스가 Go 1과 호환되지 않을 수도 있지만, 이 주제에 대한 논의는 이제 막 시작되었으며 한 가지 확실한 것은 그 과정에서 이러한 비호환성이 도입되는 경우는 거의 없을 것이라는 점입니다. 또한, 호환성 약속은 이러한 상황이 발생할 경우 이전 프로그램이 적응할 수 있는 자동 경로를 제공하도록 장려합니다.

제안서가 Go 1 사양과 호환되더라도 Go의 디자인 목표에 부합하지 않을 수 있습니다. *Google의 Go: 소프트웨어 엔지니어링을 위한 언어 설계라는* 글에서 Go의 기원과 설계 동기를 설명합니다.

유형

바둑은 객체 지향 언어인가요?

예, 아니요. Go에는 타입과 메서드가 있고 객체 지향 프로그래밍 스타일을 허용하지만, 타입 계층 구조는 없습니다. Go의 "인터페이스"라는 개념은 사용하기 쉽고 어떤 면에서는 더 일반적이라고 생각되는 다른 접근 방식을 제공합니다. 다른 유형에 유형을 포함시켜 유사하지만 동일하지는 않은 것을 제공하는 방법도 있습니다.
-를 서브클래싱할 수 있습니다. 또한, Go의 메서드는 C++나 Java보다 더 일반적입니다. 모든 종류의 데이터에 대해 정의할 수 있으며, 심지어 일반 "언박스" 정수와 같은 기본 제공 유형도 정의할 수 있습니다. 메서드는 구조체(클래스)에만 국한되지 않습니다.

또한 유형 계층 구조가 없기 때문에 Go의 '객체'는 C++나 Java와 같은 언어에 비해 훨씬 더 가볍게 느껴집니다.

메서드의 동적 디스패치를 받으려면 어떻게 해야 하나요?

메서드를 동적으로 파견할 수 있는 유일한 방법은 인터페이스를 이용하는 것입니다. 구조체나 다른 구체적인 유형의 메서드는 항상 정적으로 해결됩니다.

유형 상속이 없는 이유는 무엇인가요?

객체 지향 프로그래밍은 적어도 가장 잘 알려진 언어에서는 유형 간의 관계, 종종 자동으로

도출될 수 있는 관계에 대해 너무 많은 논의를 필요로 합니다. Go는 다른 접근 방식을 취합니다.

프로그래머가 두 타입이 서로 관련되어 있다고 미리 선언해야 하는 대신, Go에서는 한 타입이 해당 메서드의 하위 집합을 지정하는 모든 인터페이스를 자동으로 만족합니다.
이 접근 방식은 부기 작업을 줄이는 것 외에도 여러 가지 장점이 있습니다. 유형은 기존의 다중 상속의 복잡성 없이 한 번에 여러 인터페이스를 충족할 수 있습니다.
인터페이스는 매우 가벼울 수 있습니다. 메서드가 하나 또는 전혀 없는 인터페이스도 유용한 개념을 표현할 수 있습니다. 새로운 아이디어가 떠오르거나 테스트할 때 원래 유형에 주석을 달지 않고도 사후에 인터페이스를 추가할 수 있습니다. 유형과 인터페이스 사이에는 명시적인 관계가 없으므로 관리하거나 논의할 유형 계층 구조가 없습니다.

이러한 아이디어를 사용하여 유형 안전 유닉스 파이프와 유사한 것을 구성할 수 있습니다. 예를 들어, `fmt.Fprintf`가 파일뿐만 아니라 모든 출력물에 형식화된 인쇄를 가능하게 하는 방법을 참조하세요,

또는 `bufio` 패키지가 파일 I/O와 완전히 분리될 수 있는 방법이나 `이미지` 패키지가 압축된 이미지 파일을 생성하는 방법 등이 있습니다. 이 모든 아이디어는 단일 메서드(`Write`)를 나타내는 단일 인터페이스(`io.Writer`)에서 비롯됩니다. 그리고 이것은 표면적인 것에 불과합니다. Go의 인터페이스는 프로그램 구조에 지대한 영향을 미칩니다.

익숙해지려면 시간이 좀 걸리겠지만, 이러한 암묵적인 유형 종속성은 바둑에서 가장 생산적인 요소 중 하나입니다.

왜 `len`은 메서드가 아닌 함수인가요?

이 문제에 대해 논의했지만 렌과 친구를 함수로 구현하는 것이 실제로는 괜찮고 기본 유형의 인터페이스(바둑 유형의 의미에서)에 대한 질문을 복잡하게 만들지 않는다고 결정했습니다.

Go가 메서드와 연산자의 오버로드를 지원하지 않는 이유는 무엇인가요?

타입 매칭을 수행할 필요가 없는 경우 메서드 디스패치가 간소화됩니다. 다른 언어에 대한 경험에 따르면 이름은 같지만 서명이 다른 다양한 메서드를 사용하는 것이 때때로 유용할 수 있지만 실제로는 혼란스럽고 취약할 수 있다고 합니다. 이름만 일치시키고 유형에 일관성을 요구하는 것은 Go의 유형 시스템에서 중요한 단순화 결정이었습니다.

작업자 과부하에 관해서는 절대적인 요구 사항이라기보다는 편의성 측면이 더 큰 것 같습니다. 다시 말하지만, 이 기능이 없으면 상황이 더 간단합니다.

Go에 '구현' 선언이 없는 이유는 무엇인가요?

Go 유형은 해당 인터페이스의 메서드를 구현하는 것만으로 인터페이스를 충족합니다. 이 속성을 사용하면 기존 코드를 수정할 필요 없이 인터페이스를 정의하고 사용할 수 있습니다. 이는 일종의 구조적 타이핑을 가능하게 하여 관심사 분리를 촉진하고 코드 재사용을 개선하며 코드가 발전함에 따라 나타나는 패턴을 기반으로 더 쉽게 빌드할 수 있게 해줍니다. 인터페이스의 의미론은 Go가 민첩하고 가벼운 느낌을 주는 주된 이유 중 하나입니다.

자세한 내용은 유형 상속 관련 질문을 참조하세요.

내 유형이 인터페이스를 만족하는지 어떻게 확인할 수 있나요?

컴파일러에 T의 0 값 또는 T에 대한 포인터를 적절히 사용하여 할당을 시도하여 T 유형이 인터페이스 I를 구현하는지 확인하도록 요청할 수 있습니다:

```
유형 T 구조체{}
I = T{}                    // T가 I를 구현하는지 확인합니다. var_
I = (*T)(nil) // *T가 I를 구현하는지 확인합니다.
```

T(또는 *T)가 I를 구현하지 않으면 컴파일 타임에 실수가 포착됩니다.

인터페이스 사용자가 인터페이스가 구현되었음을 명시적으로 선언하도록 하려면 인터페이스의 메서드 집합에 설명이 포함된 이름을 가진 메서드를 추가하면 됩니다. 예를 들어

```
유형 Fooer 인터페이스 {
    Foo()
    ImplementsFooer()
}
```

그런 다음 유형은 Fooer가 되기 위해 ImplementsFooer 메서드를 구현해야 하며, 이 사실을 명확하게 문서화하여 go 문서의 출력에 알려야 합니다.

```
유형 바 구조체{}
func (b Bar) ImplementsFooer() {}
func (b Bar) Foo() {}
```

대부분의 코드는 인터페이스 아이디어의 유용성을 제한하기 때문에 이러한 제약 조건을 사용하지 않습니다. 하지만 때로는 유사한 인터페이스 간의 모호함을 해결하기 위해 필요한 경우도 있습니다.

왜 T 유형이 동등한 인터페이스를 충족하지 않나요?

다른 값과 비교할 수 있는 객체를 나타내는 이 간단한 인터페이스를 생각해 보겠습니다:

```
유형 이퀄러 인터페이스 { 이퀄(
    이퀄러) bool
}
```

그리고 이 유형은 T:

```
유형 T int
func (t T) Equal(u T) bool { return t == u } // 이퀄러를 만족하지 않음
```

일부 다형성 유형 시스템의 유사한 상황과 달리 T는 다음을 구현하지 않습니다. Equaler. T.Equal의 인자 유형은 문자 그대로 필수 유형인 Equaler가 아니라 T입니다.

바둑에서 유형 시스템은 같음의 인수를 장려하지 않습니다. 이는 프로그래머의 책임이며, 같음을 구현하는 유형 T2에서 볼 수 있듯이 프로그래머의 책임입니다:

```
유형 T2 int
func (t T2) Equal(u Equaler) bool { return t == u.(T2) } // 이퀄러 만족
```

하지만 이것도 다른 타입 시스템과는 다릅니다. Go에서는 Equaler를 만족하는 *모든* 타입을 T2.Equal의 인자로 전달할 수 있으며, 런타임에 인자가 T2 타입인지 확인해야 하기 때문입니다. 일부 언어에서는 컴파일 시 이를 보장하도록 준비합니다.

이와 관련된 예는 다른 방향입니다:

```
타입 오프너 인터페이스 {
    Open() 리더
}
```

```
func (t T3) Open() *os.File
```

바둑에서 T3는 다른 언어에서는 가능하지만 오프너를 만족시키지 못합니다.

이러한 경우 Go의 타입 시스템이 프로그래머에게 덜 도움이 되는 것은 사실이지만, 서브타입이 없기 때문에 인터페이스 만족도에 대한 규칙(함수의 이름과 서명이 인터페이스의 이름과 정확히 일치하는가?)을 매우 쉽게 명시할 수 있습니다. Go의 규칙은 또한 효율적으로 구현하기 쉽습니다. 저희는 이러한 장점들이 자동 유형 승격의 부족함을 상쇄한다고 생각합니다. 언젠가 Go가 어떤 형태의 다형성 타이핑을 채택하게 된다면, 이러한 예제의 아이디어를 표현하고 정적으로 검사할 수 있는 방법이 있을 것으로 예상합니다.

T를 []인터페이스{}로 변환할 수 있나요?

직접적으로는 아닙니다. 두 유형이 메모리에서 동일한 표현을 갖지 않기 때문에 언어 사양에서 허용되지 않습니다. 요소를 대상 슬라이스로 개별적으로 복사해야 합니다. 이 예는 `int` 슬라이스를 `interface{}` 슬라이스로 변환합니다:

```
t := []int{1, 2, 3, 4}
s := make([]interface{}, len(t))
for i, v := range t {
    s[i] = v
}
```

T1과 T2의 기본 유형이 동일한 경우 []T1을 []T2로 변환할 수 있나요?

이 코드 샘플의 마지막 줄은 컴파일되지 않습니다.

```
유형  T1  int

유형  T2  int

var t1 T1
var x = T2(t1) // OK
var st1 []T1
var sx = ([]T2)(st1) // NOT OK
```

Go에서 유형은 메서드와 밀접하게 연관되어 있는데, 모든 명명된 유형은 (비어 있을 수 있는) 메서드 집합을 가지고 있습니다. 일반적으로 변환되는 타입의 이름은 변경할 수 있지만(따라서 메서드 집합도 변경할 수 있지만) 복합 타입의 요소의 이름(및 메서드 집합)은 변경할 수 없습니다. Go

에서는 유형 변환에 대해 명시적으로 설명해야 합니다.

오류 값이 0이 아닌 이유는 무엇인가요?

내부적으로 인터페이스는 유형 T와 값 V라는 두 가지 요소로 구현됩니다. V는 인터페이스 자체가 아닌 int, 구조체 또는 포인터와 같은 구체적인 값이며, 유형 T를 갖습니다. 예를 들어, 인터페이스에 int 값 3을 저장하면 결과 인터페이스 값은 도식적으로 (T=int, V=3)이 됩니다. 주어진 인터페이스 변수가 프로그램 실행 중에 다른 값 V(및 해당 유형 T)를 보유할 수 있으므로 값 V를 인터페이스의 *동적*값이라고도 합니다.

인터페이스 값은 V와 T가 모두 설정되지 않은 경우(T=nil, V가 설정되지 않은 경우)에만 nil이며, 특히 nil 인터페이스는 항상 nil 타입을 보유합니다. 인터페이스 값 안에 *int 타입의 nil 포인터를 저장하면 포인터의 값에 관계없이 내부 타입은 *int가 됩니다(T=*int, V=nil). 따라서 이러한 인터페이스 값은 *내부의 포인터 값 V가 nil인 경우에도* nil이 아닙니다.

이러한 상황은 혼란스러울 수 있으며, 오류 반환과 같은 인터페이스 값 내부에 nil 값이 저장될 때 발생합니다:

```
함수 returnsError() error {
    var p *MyError = nil
    if bad() {
        p = ErrBad
    }
    반환 p // 항상 null이 아닌 오류를 반환합니다.
}
```

모든 것이 정상적으로 진행되면 함수는 nil p를 반환하므로 반환값은 오류 인터페이스 값(T=*MyError, V=nil)을 보유하게 됩니다. 즉, 호출자가 반환된 에러를 nil과 비교하면 아무 문제가 없더라도 항상 에러가 발생한 것처럼 보입니다. 호출자에게 적절한 nil 오류를 반환하려면 함수가 명시적으로 nil을 반환해야 합니다:

```
함수 returnsError() error {
    if bad() {
        반환 ErrBad
    }
    반환 없음
}
```

오류를 항상 반환하는 함수는 오류가 올바르게 생성되도록 하기 위해 *MyError와 같은 구체적인 유형이 아닌 서명에서 오류 유형을 사용하는 것이 좋습니다(위에서 한 것처럼). 예를 들어, os.Open은 nil이 아닌 경우에도 오류를 반환하지만 항상 구체적인 유형인 *os.PathError를 사용합니다.

인터페이스를 사용할 때마다 여기에 설명된 것과 비슷한 상황이 발생할 수 있습니다. 인터페이스에 구체적인 값이 저장되어 있으면 인터페이스가 0이 되지 않는다는 점만 기억하세요. 자세한

내용은 반사의 법칙을 참조하세요.

C에서와 같이 태그가 지정되지 않은 유니온이 없는 이유는 무엇인가요?

태그가 지정되지 않은 유니온은 Go의 메모리 안전 보장을 위반하게 됩니다.

Go에 변형 유형이 없는 이유는 무엇인가요?

대수 유형이라고도 하는 변형 유형은 값이 다른 유형 집합 중 하나를 취할 수 있지만 해당 유형만 취할 수 있도록 지정하는 방법을 제공합니다. 시스템 프로그래밍의 일반적인 예는 오류가 네트워크 오류, 보안 오류 또는 애플리케이션 오류임을 지정하고 호출자가 다음을 검사하여 문제의 원인을 구별할 수 있도록 하는 것입니다.

오류의 유형입니다. 또 다른 예로는 선언, 문, 할당 등 각 노드가 다른 유형이 될 수 있는 구문 트리가 있습니다.

Go에 변형 유형을 추가하는 것을 고려했지만, 논의 끝에 인터페이스와 혼란스러운 방식으로 겹치기 때문에 제외하기로 결정했습니다. 변형 유형의 요소 자체가 인터페이스라면 어떻게 될까요?

또한 변형 유형이 처리하는 것 중 일부는 이미 언어에서 다루고 있습니다. 오류 예제는 오류를 유지하는 인터페이스 값과 대소문자를 구분하는 유형 스위치를 사용하여 쉽게 표현할 수 있습니다. 구문 트리 예시도 우아하지는 않지만 가능합니다.

Go에 가변 결과 유형이 없는 이유는 무엇인가요?

가변 결과 유형은 다음과 같은 인터페이스를 의미합니다.

```
유형 복사 가능 인터페이스 {
    Copy() 인터페이스{}
}
```

메서드에 만족할 것입니다.

```
함수 (v 값) 복사() 값
```

가 빈 인터페이스를 구현하기 때문입니다. Go에서 메서드 유형은 정확히 일치해야 하므로 Value는 `Copyable`을 구현하지 않습니다. Go는 타입이 하는 일, 즉 메서드에 대한 개념과 타입의 구현을 분리합니다. 두 메서드가 서로 다른 타입을 반환하면 같은 일을 하는 것이 아닙니다. 가변 결과 유형을 원하는 프로그래머는 인터페이스를 통해 유형 계층 구조를 표현하려고 하는 경우가 많습니다. Go에서는 인터페이스와 구현을 깔끔하게 분리하는 것이 더 자연스럽습니다.

값

Go에서 암시적 숫자 변환을 제공하지 않는 이유는 무엇인가요?

C에서 숫자 유형 간 자동 변환의 편리함은 이로 인한 혼란보다 훨씬 큽니다. 표현식은 언제 부호화되지 않나요? 값이 얼마나 큰가요? 오버플로되나요? 결과가 실행되는 컴퓨터와 무관하게 이식 가

능한가? 또한 '일반적인 산술 변환'은 구현하기 쉽지 않고 아키텍처에 따라 일관성이 떨어지므로 컴파일러를 복잡하게 만듭니다. 이식성을 위해 코드에서 일부 명시적인 변환을 희생하는 대신 명확하고 간단하게 만들기로 결정했습니다. 하지만 부호 및 크기 주석이 없는 임의의 정밀도 값인 Go의 상수를 정의하면 문제가 상당히 개선됩니다.

이와 관련된 세부 사항은 C와 달리 `int`가 64비트 유형이더라도 int와 `int64`는 별개의 유형이라는 점입니다. `int` 유형은 일반적이며, 정수가 몇 비트를 갖고 있는지 궁금하다면 Go에서는 명시적으로 표현하도록 권장합니다.

바둑에서 상수는 어떻게 작동하나요?

바둑은 서로 다른 숫자 유형의 변수 간의 변환에 대해 엄격하지만, 언어의 상수는 훨씬 더 유연합니다. 23, 3.14159, math.Pi와 같은 리터럴 상수는 임의의 정밀도와 오버플로 또는 언더플로우가 없는 일종의 이상적인 숫자 공간을 차지합니다. 예를 들어, math.Pi의 값은 소스 코드에서 63번째 자리로 지정되어 있으며, 이 값을 포함하는 상수 표현식은 float64가 보유할 수 있는 정밀도를 넘어서는 정밀도를 유지합니다. 상수 또는 상수 표현식이 변수(프로그램의 메모리 위치)에 할당될 때만 일반적인 부동소수점 속성과 정밀도를 가진 "컴퓨터" 숫자가 됩니다.

또한 상수는 입력된 값이 아닌 숫자에 불과하기 때문에 변수보다 더 자유롭게 사용할 수 있어 엄격한 변환 규칙에 대한 어색함을 완화할 수 있습니다. 다음과 같은 표현식을 작성할 수 있습니다.

```
sqrt2 := math.Sqrt(2)
```

를 컴파일러의 불만 없이 사용할 수 있습니다. 이상적인 숫자 2를 안전하고 정확하게 float64로 변환하여 math.Sqrt를 호출할 수 있기 때문입니다.

상수라는 제목의 블로그 게시물에서 이 주제에 대해 자세히 살펴봅니다.

지도가 내장된 이유는 무엇인가요?

문자열과 같은 이유로, 문자열은 매우 강력하고 중요한 데이터 구조이므로 구문 지원을 통해 하나의 우수한 구현을 제공하면 프로그래밍이 더 즐거워집니다. Go의 맵 구현은 대부분의 용도에 적합할 만큼 충분히 강력하다고 생각합니다. 특정 애플리케이션이 사용자 정의 구현을 통해 이점을 얻을 수 있는 경우, 사용자 정의 구현을 작성할 수는 있지만 구문상으로는 편리하지 않으므로 이는 합리적인 절충안으로 보입니다.

맵에서 슬라이스를 키로 허용하지 않는 이유는 무엇인가요?

맵 조회에는 등호 연산자가 필요한데, 슬라이스에서는 이 연산자를 구현하지 않습니다. 얕은 비교와 깊은 비교, 포인터와 값 비교, 재귀적 유형을 처리하는 방법 등 여러 가지 고려해야 할 사항이 많지만, 이러한 유형에 대해 등식이 잘 정의되어 있지 않기 때문에 등식을 구현하지 않습니다. 이 문제는 나중에 다시 검토할 수 있지만, 슬라이스에 동등성을 구현한다고 해서 기존 프로그램이 무효화되는 것은 아니지만 슬라이스 동등성이 무엇을 의미하는지에 대한 명확한 아이디어가 없다면

지금은 이 문제를 제외하는 것이 더 간단합니다.

Go 1에서는 이전 릴리스와 달리 구조체와 배열에 대해 동등성이 정의되어 있으므로 이러한 유형을 맵 키로 사용할 수 있습니다. 하지만 슬라이스에는 여전히 동등성에 대한 정의가 없습니다.

배열은 값인 반면 맵, 슬라이스 및 채널은 참조인 이유는 무엇인가요?

이 주제에는 많은 역사가 있습니다. 초기에는 맵과 채널이 구문상 포인터였기 때문에 포인터가 아닌 인스턴스를 선언하거나 사용하는 것이 불가능했습니다. 또한 배열이 어떻게 작동해야 하는지에 대해서도 고민했습니다. 결국 포인터와 값을 엄격하게 분리하는 것이 언어를 사용하기 어렵게 만든다고 판단했습니다. 이러한 유형을 관련 공유 데이터 구조에 대한 참조로 작동하도록 변경하면 이러한 문제가 해결되었습니다. 이 변경으로 일부

언어의 복잡성은 아쉬웠지만 사용성에는 큰 영향을 미쳤습니다: 바둑은 도입 후 더욱 생산적이고 편안한 언어가 되었습니다.

코드 작성

라이브러리는 어떻게 문서화되나요?

소스 코드에서 패키지 문서를 추출하여 선언, 파일 등에 대한 링크가 있는 웹 페이지로 제공하는 Go로 작성된 `godoc`이라는 프로그램이 있습니다. 인스턴스는 golang.org/pkg/에서 실행 중입니다. 실제로 godoc은 golang.org/에서 전체 사이트를 구현합니다.

`godoc` 인스턴스는 표시되는 프로그램의 심볼에 대한 풍부한 대화형 정적 분석을 제공하도록 구성할 수 있으며, 자세한 내용은 여기에 나열되어 있습니다.

명령줄에서 문서에 액세스할 수 있도록 go 도구에는 동일한 정보에 대한 텍스트 인터페이스를 제공하는 doc 하위 명령이 있습니다.

바둑 프로그래밍 스타일 가이드가 있나요?

명시적인 스타일 가이드는 없지만, 확실히 알아볼 수 있는 '바둑 스타일'이 있습니다.

Go에는 이름 지정, 레이아웃, 파일 구성에 관한 결정을 안내하는 규칙이 정해져 있습니다. Effective Go 문서에는 이러한 주제에 대한 몇 가지 조언이 포함되어 있습니다. 좀 더 직접적으로 설명하자면, `gofmt` 프로그램은 레이아웃 규칙을 적용하는 데 목적이 있는 예쁜 프린터로, 해석을 허용하는 일반적인 해야 할 일과 하지 말아야 할 일의 개요를 대체합니다. 리포지토리에 있는 모든 Go 코드와 오픈 소스 세계의 대다수는 `gofmt`를 통해 실행됩니다.

Go 코드 리뷰 코멘트라는 제목의 문서는 프로그래머가 종종 놓치는 Go 관용구의 세부 사항에 대한 매우 짧은 에세이 모음입니다. Go 프로젝트에 대한 코드 리뷰를 수행하는 사람들에게 유용한 참고 자료입니다.

Go 라이브러리에 패치를 제출하려면 어떻게 하나요?

라이브러리 소스는 저장소의 `src` 디렉토리에 있습니다. 중요한 변경을 하고자 하는 경우 시작하

기 전에 메일링 리스트에서 논의하세요.

진행 방법에 대한 자세한 내용은 Go 프로젝트에 기여하기 문서를 참조하세요.

리포지토리를 복제할 때 "go get"이 HTTPS를 사용하는 이유는 무엇인가요?

기업에서는 표준 TCP 포트 80(HTTP) 및 443(HTTPS)에서만 발신 트래픽을 허용하고 TCP 포트 9418(git) 및 TCP 포트 22(SSH)를 포함한 다른 포트의 발신 트래픽을 차단하는 경우가 많습니다. HTTP 대신 HTTPS를 사용하는 경우, `git`은 기본적으로 인증서 유효성 검사를 적용하여 중간자 공격, 도청 및 변조 공격에 대한 보호 기능을 제공합니다. 따라서 `go get` 명령은 안전을 위해 HTTPS를 사용합니다.

Git은 HTTPS를 통해 인증하거나 HTTPS 대신 SSH를 사용하도록 구성할 수 있다. HTTPS를 통해 인증하려면 `$HOME/.netrc` 파일에 git이 참조하는 줄을 추가하면 됩니다:

```
기계 github.com 로그인 사용자 이름 비밀번호 APIKEY
```

GitHub 계정의 경우 비밀번호는 개인 액세스 토큰일 수 있습니다.

특정 접두사와 일치하는 URL에 대해 HTTPS 대신 SSH를 사용하도록 Git을 구성할 수도 있습니다. 예를 들어 모든 GitHub 액세스에 SSH를 사용하려면 `~/.gitconfig`에 다음 줄을 추가하세요:

```
[url "ssh://git@github.com/"]
        대신 = https://github.com/
```

"get"을 사용하여 패키지 버전을 관리하려면 어떻게 해야 하나요?

Go 툴체인에는 *모듈이라고 하는* 관련 패키지의 버전별 집합을 관리하기 위한 시스템이 내장되어 있습니다. 모듈은 Go 1.11에 도입되었으며, 1.14부터 프로덕션 환경에서 사용할 수 있게 되었습니다.

모듈을 사용하여 프로젝트를 만들려면 `go mod init`을 실행합니다. 이 명령은 종속성 버전을 추적하는 `go.mod` 파일을 생성합니다.

```
GO MOD 초기화 예제/프로젝트
```

종속성을 추가, 업그레이드 또는 다운그레이드하려면 `get`을 실행하세요:

```
바로가기 golang.org/x/text@v0.3.5
```

튜토리얼을 참조하세요: 시작하는 방법에 대한 자세한 내용은 튜토리얼: 모듈 만들기를 참조하세요.

모듈을 사용한 종속성 관리에 대한 가이드는 모듈 개발을 참조하세요.

모듈 내의 패키지는 가져오기 호환성 규칙에 따라 발전함에 따라 이전 버전과의 호환성을 유지해야 합니다:

> 이전 패키지와 새 패키지의 가져오기 경로가 동일한 경우,
> 새 패키지는 이전 패키지와 하위 호환이 가능해야 합니다.

내보낸 이름을 제거하지 말고, 태그가 지정된 복합 리터럴을 권장하는 등 Go 1 호환성 가이드라인을 참고하세요. 다른 기능이 필요한 경우 기존 이름을 변경하는 대신 새 이름을 추가하세요.

모듈은 시맨틱 버전 관리 및 시맨틱 가져오기 버전 관리로 이를 코드화합니다. 호환성 중단이 필요한 경우 새로운 주 버전으로 모듈을 릴리스하세요. 메이저 버전 2 이상의 모듈은 경로의 일부로 메이저 버전 접미사(예: `/v2`)가 필요합니다. 이렇게 하면 가져오기 호환성 규칙이 유지되므로 모듈의 서로 다른 주요 버전에 있는 패키지는 서로 다른 경로를 갖게 됩니다.

포인터 및 할당

함수 파라미터는 언제 값으로 전달되나요?

C 계열의 모든 언어와 마찬가지로 Go의 모든 것은 값으로 전달됩니다. 즉, 함수는 매개변수에 값을 할당하는 할당문이 있는 것처럼 항상 전달되는 값의 복사본을 가져옵니다. 예를 들어, 함수에 정수 값을 전달하면 정수의 복사본이 만들어지고, 포인터 값을 전달하면 포인터의 복사본이 만들어지지만 포인터가 가리키는 데이터는 복사되지 않습니다. (이것이 메서드 수신자에 어떤 영향을 미치는지에 대한 설명은 뒷부분을 참조하세요.)

맵 및 슬라이스 값은 포인터처럼 작동하며, 기본 맵 또는 슬라이스 데이터에 대한 포인터를 포함하는 설명자입니다. 맵 또는 슬라이스 값을 복사해도 해당 값이 가리키는 데이터는 복사되지 않습니다. 인터페이스 값을 복사하면 인터페이스 값에 저장된 항목의 복사본이 만들어집니다. 인터페이스 값이 구조체를 포함하는 경우 인터페이스 값을 복사하면 구조체의 복사본이 만들어집니다. 인터페이스 값이 포인터를 포함하는 경우 인터페이스 값을 복사하면 포인터의 복사본이 만들어지지만 포인터가 가리키는 데이터는 복사되지 않습니다.

이 논의는 연산의 의미론에 관한 것입니다. 실제 구현에서는 최적화가 의미를 변경하지 않는 한 복사를 피하기 위해 최적화를 적용할 수 있습니다.

인터페이스에 대한 포인터는 언제 사용해야 하나요?

거의 없습니다. 인터페이스 값에 대한 포인터는 지연된 평가를 위해 인터페이스 값의 유형을 위장해야 하는 드물고 까다로운 상황에서만 발생합니다.

인터페이스를 기대하는 함수에 인터페이스 값에 대한 포인터를 전달하는 것은 흔한 실수입니다. 컴파일러는 이 오류에 대해 불만을 표시하지만 인터페이스를 만족시키기 위해 포인터가 필요한 경우도 있기 때문에 상황이 여전히 혼란스러울 수 있습니다. 인사이트는 구체적인 유형에 대한 포인터는 인터페이스를 만족시킬 수 있지만, 한 가지 예외를 제외하고는 *인터페이스에 대한 포인터는 인터페이스를 결코 만족시킬 수 없다는* 것입니다.

변수 선언을 생각해 보세요,

```
var w io.Writer
```

인쇄 함수 fmt.Fprintf는 첫 번째 인자로 다음 조건을 만족하는 값을 받습니다.

io.Writer - 표준 Write 메서드를 구현하는 메서드입니다. 따라서 다음과 같이 작성할 수 있습니다.

```
fmt.Fprintf(w, "안녕하세요, 세계\n")
```

그러나 w의 주소를 전달하면 프로그램이 컴파일되지 않습니다.

```
fmt.Fprintf(&w, "hello, world\n") // 컴파일 타임 오류.
```

한 가지 예외는 인터페이스에 대한 포인터를 포함한 모든 값을 빈 인터페이스 유형 (interface{})의 변수에 할당할 수 있다는 것입니다. 그럼에도 불구하고 값이 인터페이스에 대한 포인터인 경우 거의 확실하게 실수이며, 그 결과는 혼란을 야기할 수 있습니다.

값이나 포인터에 메서드를 정의해야 하나요?

```
func (s *MyStruct) pointerMethod() { } // 메서드 온 포인터
func (s MyStruct)valueMethod()     } // 값에 대한 메서드
```

포인터에 익숙하지 않은 프로그래머에게는 이 두 가지 예제의 구분이 혼란스러울 수 있지만, 실제로는 매우 간단합니다. 타입에 메서드를 정의할 때 위의 예제에서 수신자는 마치 메서드의 인자처럼 동작합니다. 수신자를 값으로 정의할지 포인터로 정의할지는 함수 인자가 값이어야 하는지 포인터이어야 하는지와 같은 문제입니다. 몇 가지 고려 사항이 있습니다.

첫째, 가장 중요한 것은 메서드가 수신자를 수정해야 하는가 하는 점입니다. 그렇다면 수신자는 포인터여야 합니다. (슬라이스와 맵은 참조 역할을 하므로 조금 더 미묘하지만, 예를 들어 메서드에서 슬라이스의 길이를 변경하려면 수신자가 여전히 포인터여야 합니다.) 위의 예에서 `pointerMethod`가 `s`의 필드를 수정하면 호출자는 이러한 변경 사항을 볼 수 있지만, valueMethod는 호출자의 인자 복사본(즉, 값을 전달하는 정의)과 함께 호출되므로 호출자에게는 변경 사항이 보이지 않습니다.

그건 그렇고, Java에서 메서드 수신자는 포인터 특성이 다소 위장되어 있지만 항상 포인터입니다 (언어에 값 수신자를 추가하는 제안이 있습니다). 특이한 것은 Go의 값 수신자입니다.

두 번째는 효율성에 대한 고려입니다. 예를 들어 수신기가 큰 구조물이라면 포인터 수신기를 사용하는 것이 훨씬 저렴할 것입니다.

다음은 일관성입니다. 타입의 일부 메서드에 포인터 리시버가 있어야 하는 경우 나머지 메서드도 포인터 리시버가 있어야 하므로 타입이 사용되는 방식에 관계없이 메서드 집합이 일관성을 유지합니다. 자세한 내용은 메서드 세트 섹션을 참조하십시오.

기본 타입, 슬라이스, 작은 구조체와 같은 타입의 경우 값 리시버는 매우 저렴하므로 메서드의 의미론에 포인터가 필요하지 않는 한 값 리시버가 효율적이고 명확합니다.

신규와 만들기의 차이점은 무엇인가요?

간단히 말해, new는 메모리를 할당하고 make는 슬라이스, 맵 및 채널 유형을 초기화합니다.

자세한 내용은 효과적인 이동의 관련 섹션을 참조하세요.

64비트 컴퓨터에서 **int의** 크기는 얼마인가요?

int와 uint의 크기는 구현에 따라 다르지만 특정 플랫폼에서는 서로 동일합니다. 이식성을 위해 특정 크기의 값에 의존하는 코드는 int64와 같이 명시적으로 크기가 지정된 유형을 사용해야 합니다. 32비트 시스템에서는 컴파일러가 기본적으로 32비트 정수를 사용하는 반면, 64비트 시스템에서는 정수가 64비트를 갖습니다. (역사적으로 항상 그렇지는 않았습니다.)

반면에 부동 소수점 스칼라와 복소수 타입은 항상 크기가 지정됩니다(
또는 복잡한 기본 유형), 프로그래머는 다음과 같은 경우 정밀도를 고려해야 합니다.

부동 소수점 숫자를 사용합니다. (유형화되지 않은) 부동 소수점 상수에 사용되는 기본 유형은 `float64`입니다. 따라서 `foo := 3.0`은 `float64` 타입의 변수 `foo`를 선언합니다. (유형이 지정되지 않은) 상수로 초기화되는 `float32` 변수의 경우 변수 선언에 변수 유형을 명시적으로 지정해야 합니다:

```
var foo float32 = 3.0
```

또는 상수에 `foo := float32(3.0)`과 같이 변환이 있는 유형을 지정해야 합니다.

변수가 힙에 할당되었는지 스택에 할당되었는지 어떻게 알 수 있나요?

정확성의 관점에서는 알 필요가 없습니다. Go의 각 변수는 그에 대한 참조가 있는 한 존재합니다. 구현에서 선택한 저장 위치는 언어의 의미론과 무관합니다.

저장 위치는 효율적인 프로그램 작성에 영향을 미칩니다. 가능한 경우, Go 컴파일러는 함수에 로컬인 변수를 해당 함수의 스택 프레임에 할당합니다. 그러나 컴파일러가 함수가 반환된 후 해당 변수가 참조되지 않았음을 증명할 수 없는 경우 컴파일러는 가비지 수집 힙에 변수를 할당하여 댕글 포인터 오류를 방지해야 합니다. 또한 로컬 변수가 매우 큰 경우 스택이 아닌 힙에 저장하는 것이 더 합리적일 수 있습니다.

현재 컴파일러에서는 변수에 주소를 가져온 경우 해당 변수가 힙에 할당할 후보가 됩니다. 그러나 기본 *이스케이프 분석은* 이러한 변수가 함수에서 반환된 후에도 스택에 남아있을 수 있는 몇 가지 경우를 인식합니다.

내 Go 프로세스가 왜 이렇게 많은 가상 메모리를 사용하나요?

Go 메모리 할당자는 할당을 위한 경기장으로 가상 메모리의 넓은 영역을 예약합니다. 이 가상 메모리는 특정 Go 프로세스에 로컬이며, 예약으로 인해 다른 프로세스의 메모리가 박탈되지 않습니다.

Go 프로세스에 할당된 실제 메모리 양을 확인하려면 Unix의 `top` 명령을 사용하여 RES(Linux) 또는 RSIZE(macOS) 열을 참조하세요.

동시성

어떤 연산이 원자 연산인가요? 뮤텍스는 무엇인가요?

바둑에서 연산의 원자성에 대한 설명은 바둑 메모리 모델 문서에서 확인할 수 있습니다.

저수준 동기화 및 원자 프리미티브는 동기화 및 동기화/아토믹 패키지에서 사용할 수 있습니다. 이 패키지는 참조 개수를 늘리거나 소규모 상호 제외를 보장하는 등의 간단한 작업에 적합합니다.

동시 서버 간의 조정과 같은 더 높은 수준의 작업의 경우, 더 높은 수준의 기술이 더 좋은 프로그램으로 이어질 수 있으며 Go는 다음을 통해 이러한 접근 방식을 지원합니다.

고루틴 및 채널. 예를 들어, 한 번에 하나의 고루틴만 특정 데이터를 담당하도록 프로그램을 구성할 수 있습니다. 이러한 접근 방식은 원래 바둑 속담에 요약되어 있습니다,

메모리를 공유하여 통신하지 마십시오. 대신 통신을 통해 메모리를 공유하세요.

이 개념에 대한 자세한 내용은 통신을 통한 메모리 공유 코드 워크 및 관련 문서를 참조하세요.

대규모 동시 실행 프로그램은 이 두 가지 툴킷을 모두 차용할 가능성이 높습니다.

더 많은 CPU를 사용하면 프로그램이 더 빨리 실행되지 않는 이유는 무엇인가요?

프로그램이 더 많은 CPU로 더 빠르게 실행되는지 여부는 해결하려는 문제에 따라 달라집니다. Go 언어는 고루틴과 채널과 같은 동시성 프리미티브를 제공하지만, 동시성은 기본 문제가 본질적으로 병렬인 경우에만 병렬 처리를 가능하게 합니다.
본질적으로 순차적인 문제는 CPU를 더 추가해도 속도를 높일 수 없지만, 병렬로 실행할 수 있는 조각으로 나눌 수 있는 문제는 때로는 획기적으로 속도를 높일 수 있습니다.

때때로 CPU를 더 추가하면 프로그램 속도가 느려질 수 있습니다. 실제로 유용한 연산을 수행하는 것보다 동기화나 통신에 더 많은 시간을 소비하는 프로그램은 여러 OS 스레드를 사용할 때 성능 저하를 경험할 수 있습니다. 스레드 간에 데이터를 전달하려면 컨텍스트를 전환해야 하므로 상당한 비용이 발생하고 CPU가 많아질수록 비용이 증가할 수 있기 때문입니다. 예를 들어, Go 사양의 프라임 체 예제는 많은 고루틴을 실행하지만 병렬성이 크지 않으며, 스레드(CPU) 수를 늘리면 속도가 빨라지기보다는 느려질 가능성이 더 높습니다.

이 주제에 대한 자세한 내용은 동시성은 병렬 처리가 아닙니다라는 제목의 강연을 참조하세요.

CPU 수를 제어하려면 어떻게 해야 하나요?

고루틴을 실행하는 데 동시에 사용할 수 있는 CPU의 수는 기본값이 사용 가능한 CPU 코어 수인 `GOMAXPROCS` 셸 환경 변수에 의해 제어됩니다. 따라서 병렬 실행 가능성이 있는 프로그램은 다중 CPU 시스템에서 기본적으로 병렬 실행을 수행해야 합니다. 사용할 병렬 CPU 수를 변경하려면 환경 변수를 설정하거나 런타임 패키지의 유사한 이름의 함수를 사용하여 다른 스레드 수를 활용하도록 런타임 지원을 구성합니다. 이 값을 1로 설정하면 진정한 병렬 처리가 불

가능해져 독립적인 고루틴이 번갈아 실행됩니다.

런타임은 여러 개의 미결 I/O 요청을 처리하기 위해 GOMAXPROCS 값보다 더 많은 스레드를 할당할 수 있습니다. GOMAXPROCS는 실제로 한 번에 실행할 수 있는 고루틴 수에만 영향을 미치며, 임의로 더 많은 고루틴이 시스템 호출에서 차단될 수 있습니다.

Go의 고루틴 스케줄러는 시간이 지남에 따라 개선되긴 했지만, 아직까지 그다지 좋은 편은 아닙니다. 향후에는 OS 스레드 사용을 더 최적화할 수 있습니다. 현재로서는 성능 문제가 있는 경우 애플리케이션별로 GOMAXPROCS를 설정하는 것이 도움이 될 수 있습니다.

고루틴 ID가 없는 이유는 무엇인가요?

고루틴은 이름이 없으며 익명의 워커일 뿐입니다. 고루틴은 프로그래머에게 고유 식별자, 이름, 데이터 구조를 노출하지 않습니다. 어떤 사람들은 이 사실에 놀라며, `go` 문이 나중에 고루틴에 액세스하고 제어하는 데 사용할 수 있는 어떤 항목을 반환할 것으로 기대합니다.

고루틴이 익명으로 처리되는 근본적인 이유는 동시 코드를 프로그래밍할 때 전체 Go 언어를 사용할 수 있도록 하기 위해서입니다. 반면, 스레드와 고루틴의 이름이 지정될 때 발생하는 사용 패턴은 이를 사용하는 라이브러리가 수행할 수 있는 작업을 제한할 수 있습니다.

다음은 그 어려움을 보여주는 예시입니다. 고루틴에 이름을 붙이고 이를 중심으로 모델을 구성하면 고루틴은 특별해지며, 처리를 위해 여러 개의 공유 고루틴을 사용할 가능성을 무시하고 모든 연산을 해당 고루틴과 연결하려는 유혹에 빠지게 됩니다. `net/http` 패키지가 요청별 상태를 고루틴과 연결하면 클라이언트는 요청을 처리할 때 더 많은 고루틴을 사용할 수 없게 됩니다.

또한 모든 처리가 '메인 스레드'에서 이루어져야 하는 그래픽 시스템용 라이브러리와 같은 라이브러리를 사용한 경험은 동시 언어로 배포할 때 이러한 접근 방식이 얼마나 어색하고 제한적일 수 있는지 보여주었습니다. 특수 스레드 또는 고루틴이 존재하기 때문에 프로그래머는 실수로 잘못된 스레드에서 작동하여 발생하는 충돌 및 기타 문제를 피하기 위해 프로그램을 왜곡해야 합니다.

특정 고루틴이 정말 특별한 경우, 언어에서는 유연한 방식으로 고루틴과 상호작용할 수 있는 채널과 같은 기능을 제공합니다.

기능 및 방법

T와 *T의 메서드 세트가 다른 이유는 무엇인가요?

Go 사양에 따르면, 타입 `T`의 메서드 집합은 수신자 타입 `T`를 가진 모든 메서드로 구성되고, 해당 포인터 타입 `*T`의 메서드 집합은 수신자 `*T` 또는 `T`를 가진 모든 메서드로 구성됩니다. 즉, `*T`의 메서드 집합에는 `T`의 메서드가 포함되지만 그 반대의 경우는 포함되지 않습니다.

인터페이스 값에 포인터 `*T`가 포함되어 있으면 메서드 호출이 포인터를 역참조하여 값을 얻을

수 있지만, 인터페이스 값에 값 T가 포함되어 있으면 메서드 호출이 포인터를 얻을 수 있는 안전한 방법이 없기 때문에 이러한 구분이 생깁니다. (그렇게 하면 메서드가 인터페이스 내부의 값 내용을 수정할 수 있으며, 이는 언어 사양에서 허용되지 않습니다.)

컴파일러가 메서드에 전달할 값의 주소를 가져올 수 있는 경우에도 메서드가 값을 수정하면 호출자에서 변경 사항이 손실됩니다. 예를 들어, `bytes.Buffer`의 `Write` 메서드가 포인터가 아닌 값 수신자를 사용하는 경우 이 코드는 다음과 같습니다:

```
var buf bytes.Buffer
io.Copy(buf, os.Stdin)
```

는 표준 입력을 buf 자체에 복사하는 것이 아니라 buf의 *복사본에* 복사합니다. 이는 원하는 동작이 거의 아닙니다.

고루틴으로 실행되는 클로저는 어떻게 되나요?

동시성과 함께 클로저를 사용할 때 약간의 혼란이 발생할 수 있습니다. 다음 프로그램을 고려해 보세요:

```
함수 main() {
    done := make(chan bool)

    values := []string{"a", "b", "c"}
    for _, v := 범위 값 {
        go func() {
            fmt.Println(v)
            done <- true
        }()
    }

    // 범위 값에 대해 _ = 범위 값을 종료하기 전에 모든 고루틴이 완료될
    때까지 기다립니다.
        <-done
    }
}
```

출력으로 a, b, c가 표시될 것으로 잘못 예상할 수 있습니다. 루프의 각 반복이 동일한 변수 v의 인스턴스를 사용하므로 각 클로저가 해당 단일 변수를 공유하기 때문입니다. 클로저가 실행되면 fmt.Println이 실행되는 시점의 v 값을 인쇄하지만, 고루틴이 실행된 이후 v가 수정되었을 수 있습니다. 이 문제 및 기타 문제가 발생하기 전에 이를 감지하려면 go vet을 실행하세요.

클로저가 실행될 때마다 현재 값인 v를 각 클로저에 바인딩하려면 반복할 때마다 새 변수를 생성하도록 내부 루프를 수정해야 합니다. 한 가지 방법은 변수를 클로저에 인수로 전달하는 것입니다:

```
    for _, v := 범위 값 { go
        func(u 문자열) {
            fmt.Println(u)
            done <- true
        }(v)
    }
```

이 예제에서는 익명 함수에 인자로 v 값을 전달합니다. 그러면 해당 값은 함수 내에서 변수 u로 액세스할 수 있습니다.

더 쉬운 방법은 이상해 보이지만 바둑에서 잘 작동하는 선언 스타일을 사용하여 새 변수를 만드는 것입니다:

```
for _, v := 범위 값 {
    v := v // 새 'v'를 만듭니다. go
    func() {
```

```
                fmt.Println(v)
                done <- true
            }()
        }
```

각 반복마다 새 변수를 정의하지 않는 언어의 이러한 동작은 돌이켜보면 실수였을 수 있습니다. 이 문제는 이후 버전에서 해결될 수 있지만, 호환성을 위해 Go 버전 1에서는 변경할 수 없습니다.

제어 흐름

Go에 `?:` 연산자가 없는 이유는 무엇인가요?

바둑에는 삼원 테스트 연산이 없습니다. 다음을 사용하여 동일한 결과를 얻을 수 있습니다:

```
if expr {
    n = trueVal
} else {
    n = falseVal
}
```

Go에 `?:` 연산이 없는 이유는 언어 설계자들이 이 연산이 너무 자주 사용되어 매우 복잡한 표현을 만들어내는 것을 보았기 때문입니다. `if-else` 형식은 더 길지만 의심할 여지없이 더 명확합니다. 언어에는 조건부 제어 흐름 구조가 하나만 필요합니다.

유형 매개변수

Go에 유형 매개변수가 있는 이유는 무엇인가요?

유형 매개변수를 사용하면 함수와 데이터 구조가 나중에 해당 함수와 데이터 구조가 사용될 때 지정되는 유형으로 정의되는 일반 프로그래밍을 사용할 수 있습니다. 예를 들어, 유형 매개변수를 사용하면 가능한 각 유형에 대해 별도의 버전을 작성할 필요 없이 정렬된 모든 유형의 최소값 두 개를 반환하는 함수를 작성할 수 있습니다. 예제와 함께 더 자세한 설명은 블로그 게시물 왜 제네릭인가를 참조하세요.

Go에서 제네릭은 어떻게 구현되나요?

컴파일러는 각 인스턴스화를 개별적으로 컴파일할지 아니면 상당히 유사한 인스턴스화를 단일 구현으로 컴파일할지 선택할 수 있습니다. 단일 구현 접근 방식은 인터페이스 매개변수가 있는 함수와 유사합니다. 컴파일러마다 경우에 따라 다른 선택을 할 것입니다. 표준 Go 1.18 컴파일러는 일반적으로 동일한 모양을 가진 모든 타입 인자에 대해 단일 인스턴스화를 생성하며, 여기서 모양은 포함 된 포인터의 크기 및 위치와 같은 타입의 속성에 의해 결정됩니다. 향후 릴리스에서는 컴파일 시간, 런타임 효율성, 코드 크기 간의 절충점을 실험할 예정입니다.

Go의 제네릭은 다른 언어의 제네릭과 어떻게 비교되나요?

모든 언어의 기본 기능은 비슷합니다. 나중에 지정한 유형을 사용하여 유형과 함수를 작성할 수 있습니다. 하지만 몇 가지 차이점이 있습니다.

Java

Java에서 컴파일러는 컴파일 시 일반 유형을 확인하지만 런타임에는 해당 유형을 제거합니다. 이를 유형 삭제라고 합니다. 예를 들어, 컴파일 시 `List<Integer>`로 알려진 Java 유형은 런타임 시 비일반 유형인 `List`가 됩니다. 이는 예를 들어 Java 형식의 타입 리플렉션을 사용할 때 `List<Integer>` 타입의 값과 `List<Float>` 타입의 값을 구별할 수 없음을 의미합니다. Go에서 일반 유형에 대한 리플렉션 정보에는 전체 컴파일 타임 유형 정보가 포함됩니다.

C++

Java는 일반 공분산 및 공분산을 구현하기 위해 `List<? extends Number>` 또는 `List<? super Number>`와 같은 유형 와일드카드를 사용합니다. Go에는 이러한 개념이 없으므로 Go의 일반 유형은 훨씬 더 간단합니다.

C++20은 개념을 통해 선택적 제약 조건을 지원하지만, 전통적으로 C++ 템플릿은 유형 인자에 대해 어떠한 제약 조건도 적용하지 않습니다. Go에서 제약 조건은 모든 타입 매개변수에 대해 필수입니다. C++20 개념은 유형 인자와 함께 컴파일되어야 하는 작은 코드 조각으로 표현됩니다. Go 제약 조건은 허용되는 모든 유형 인자 집합을 정의하는 인터페이스 유형입니다.

C++는 템플릿 메타프로그래밍을 지원하지만 Go는 지원하지 않습니다. 실제로 모든 C++ 컴파일러는 각 템플릿이 인스턴스화된 지점에서 컴파일하지만, 위에서 언급했듯이 Go는 인스턴스화마다 다른 접근 방식을 사용할 수 있고 실제로 사용합니다.

Rust

제약 조건의 Rust 버전은 특성 바운드라고 합니다. Rust에서 특성 바운드와 유형 간의 연결은 특성 바운드를 정의하는 상자 또는 유형을 정의하는 상자에서 명시적으로 정의해야 합니다. Go 유형이 인터페이스 유형을 암시적으로 구현하는 것과 마찬가지로 Go에서 유형 인자는 암시적으로 제약 조건을 충족합니다. Rust 표준 라이브러리는 비교 또는 덧셈과 같은 연

산에 대한 표준 특성을 정의하지만, Go 표준 라이브러리는 인터페이스 유형을 통해 사용자 코드에서 표현할 수 있기 때문에 그렇지 않습니다.

Python

파이썬은 정적으로 타입이 지정된 언어가 아니므로 모든 파이썬 함수는 기본적으로 항상 모든 타입의 값으로 호출할 수 있고 런타임에 모든 타입 오류가 감지되는 등 항상 일반적이라고 할 수 있습니다.

Go에서 유형 매개변수 목록에 대괄호를 사용하는 이유는 무엇인가요?

Java와 C++는 Java `List<Integer>` 및 C++ `std::vector<int>`에서와 같이 유형 매개변수 목록에 꺾쇠 괄호를 사용합니다. 그러나 이 옵션은 구문 문제가 발생하기 때문에 Go에서는 사용할 수 없었습니다. `v := F<T>`와 같이 함수 내에서 코드를 구문 분석할 때

를 보면 인스턴스화인지 < 연산자를 사용한 표현식인지 모호합니다. 유형 정보 없이는 이 문제를 해결하기가 매우 어렵습니다.

예를 들어 다음과 같은 문장을 생각해 보겠습니다.

```
a, b = w < x, y > (z)
```

유형 정보가 없으면 할당의 오른쪽이 한 쌍의 표현식(`w < x` 및 `y > z`)인지, 아니면 두 개의 결과 값(`(w<x, y>)(z)`을 반환하는 일반 함수 인스턴스화 및 호출인지 여부를 판단할 수 없습니다.)

제네릭에 꺾쇠 괄호를 사용할 때는 불가능해 보이는 유형 정보 없이 구문 분석이 가능하도록 하는 것은 Go의 핵심 설계 결정입니다.

대괄호를 사용하는 데 있어 대괄호를 사용하는 언어가 고유하거나 독창적인 것은 아니며, 일반 코드에도 대괄호를 사용하는 Scala와 같은 다른 언어가 있습니다.

Go에서 유형 매개변수가 있는 메서드를 지원하지 않는 이유는 무엇인가요?

Go는 일반 타입이 메서드를 가질 수 있도록 허용하지만, 수신자를 제외하고 해당 메서드의 인수는 매개변수화된 타입을 사용할 수 없습니다. 타입의 메서드는 타입이 구현하는 인터페이스를 결정하지만, 이것이 제네릭 타입의 메서드에 대한 매개변수화된 인자에서 어떻게 작동하는지는 명확하지 않습니다. 이를 위해서는 런타임에 함수를 인스턴스화하거나 가능한 모든 타입 인자에 대해 모든 제네릭 함수를 인스턴스화해야 합니다. 두 가지 접근 방식 모두 실현 가능하지 않은 것 같습니다. 예제를 포함한 자세한 내용은 제안을 참조하세요. 타입 매개변수가 있는 메서드 대신 타입 매개변수가 있는 최상위 함수를 사용하거나 수신자 타입에 타입 매개변수를 추가하세요.

매개변수화된 유형의 수신자에 대해 더 구체적인 유형을 사용할 수 없는 이유는 무엇인가요?

일반 타입의 메서드 선언은 타입 매개변수 이름을 포함하는 리시버를 사용하여 작성됩니다. 어떤 사람들은 특정 타입을 사용할 수 있다고 생각하여 특정 타입 인자에 대해서만 작동하는 메서드를 생성합니다:

```
유형 S[T any] 구조체 { f T }
```

```
함수 (s S[문자열]) Add(t 문자열) 문자열 { 반환 s.f +
    t
}
```

물론 + 연산자가 미리 선언된 유형 문자열에서 작동하더라도 + 연산자가 s.f에 정의되지 않음(any로 제약된 문자열 유형 변수)과 같은 컴파일러 오류가 발생하면 실패합니다.

이는 Add 메서드의 선언에서 문자열을 사용하는 것은 단순히 유형 매개 변수의 이름을 도입하는 것이며 그 이름이 문자열이기 때문입니다. 이것은 이상하지만 유효한 일입니다. s.f 필드의 유형은 문자열이며, 일반적인 사전 선언된 유형 문자열이 아니라 이 메서드에서 이름이 문자열인 S의 유형 매개변수입니다. 유형 매개 변수의 제약 조건은 임의의 것이므로 + 연산자는 허용되지 않습니다.

컴파일러가 내 프로그램에서 유형 인수를 유추할 수 없는 이유는 무엇인가요?

프로그래머가 일반 타입이나 함수의 타입 인자가 무엇인지 쉽게 알 수 있지만 컴파일러가 이를 유추할 수 없는 경우가 많이 있습니다. 유형 추론은 어떤 유형이 추론되는지 혼동하지 않도록 의도적으로 제한되어 있습니다. 다른 언어에 대한 경험에 따르면 예기치 않은 유형 추론은 프로그램을 읽고 디버깅할 때 상당한 혼란을 초래할 수 있습니다. 호출에 사용할 명시적 유형 인수를 항상 지정할 수 있습니다. 향후에는 규칙이 간단하고 명확하게 유지되는 한 새로운 형태의 추론이 지원될 수 있습니다.

패키지 및 테스트

다중 파일 패키지를 만들려면 어떻게 하나요?

패키지의 모든 소스 파일을 디렉토리에 단독으로 넣으세요. 소스 파일은 다른 파일의 항목을 마음대로 참조할 수 있으므로 포워드 선언이나 헤더 파일이 필요하지 않습니다.

여러 파일로 분할되는 것 외에 패키지는 단일 파일 패키지처럼 컴파일 및 테스트됩니다.

단위 테스트는 어떻게 작성하나요?

패키지 소스와 같은 디렉터리에 `_test.go`로 끝나는 새 파일을 만듭니다. 해당 파일에서 `"testing"`을 임포트하고 다음과 같은 형식의 함수를 작성합니다.

```
함수 TestFoo(t *testing.T) {
    ...
}
```

해당 디렉토리에서 `go 테스트`를 실행합니다. 이 스크립트는 `Test` 함수를 찾아서 테스트 바이너리를 빌드하고 실행합니다.

Go 코드 작성 방법 문서, 테스트 패키지 및 Go 테스트를 참조하세요.
하위 명령에서 자세한 내용을 확인할 수 있습니다.

테스트할 때 가장 좋아하는 도우미 기능은 어디에 있나요?

Go의 표준 테스트 패키지를 사용하면 단위 테스트를 쉽게 작성할 수 있지만, 어설션 함수와 같은 다른 언어의 테스트 프레임워크에서 제공하는 기능이 부족합니다. 이 문서의 이전 섹션에서 Go에 어설션이 없는 이유를 설명했으며, 테스트에서 어설션을 사용하는 데에도 동일한 논거가 적용됩니다. 적절한 오류 처리는 한 테스트가 실패한 후 다른 테스트를 실행하여 실패를 디버깅하는 사람이 무엇이 잘못되었는지 완전히 파악할 수 있도록 하는 것을 의미합니다. 테스트가 2, 3, 5, 7(또는 2, 4, 8, 16)에 대해 isPrime이 잘못된 답을 제공한다고 보고하는 것이 2에 대해 잘못된 답을 제공하므로 더 이상 테스트가 실행되지 않았다고 보고하는 것보다 더 유용합니다. 테스트 실패를 트리거한 프로그래머는 실패한 코드에 익숙하지 않을 수 있습니다. 좋은 오류 메시지를 작성하는 데 투자한 시간이 나중에 테스트가 실패할 때 빛을 발합니다.

이와 관련하여 테스트 프레임워크는 조건문과 제어, 인쇄 메커니즘을 갖춘 자체적인 미니 언어로 발전하는 경향이 있지만, Go에는 이미 이러한 기능이 모두 포함되어 있는데 왜 다시 만들까요? 학습해야 할 언어가 하나 더 적고 접근 방식이 간단하고 이해하기 쉬우므로 차라리 Go로 테스트를 작성하는 편이 낫습니다.

좋은 오류를 작성하는 데 필요한 추가 코드의 양이 반복적이고 부담스러워 보인다면, 데이터 구조에 정의된 입력 및 출력 목록을 반복하는 테이블 기반 테스트가 더 효과적일 수 있습니다(Go는 데이터 구조 리터럴에 대한 탁월한 지원을 제공합니다). 그러면 좋은 테스트와 좋은 오류 메시지를 작성하기 위한 작업이 많은 테스트 케이스에 걸쳐 분할됩니다. 표준 Go 라이브러리에는 `fmt` 패키지의 서식 지정 테스트와 같은 예시가 가득합니다.

표준 라이브러리에 *X가* 없는 이유는 무엇인가요?

표준 라이브러리의 목적은 런타임을 지원하고, 운영 체제에 연결하며, 형식화된 입출력 및 네트워킹과 같이 많은 Go 프로그램에 필요한 주요 기능을 제공하는 것입니다. 또한 암호화와 HTTP, JSON, XML과 같은 표준 지원 등 웹 프로그래밍에 중요한 요소도 포함되어 있습니다.

오랫동안 이것이 *유일한* Go 라이브러리였기 때문에 무엇이 포함되는지 정의하는 명확한 기준은 없습니다. 하지만 오늘날 추가되는 항목을 정의하는 기준이 있습니다.

표준 라이브러리에 새로 추가되는 경우는 드물고 포함 기준이 매우 높습니다. 표준 라이브러리에 포함된 코드는 지속적인 유지보수 비용이 많이 들고(원저자가 아닌 다른 사람이 부담하는 경우가 많음), Go 1 호환성 약속(API의 모든 결함에 대한 수정 차단)의 적용을 받으며, Go 릴리스 일정의 영향을 받기 때문에 사용자에게 버그 수정이 신속하게 제공되지 않을 수 있습니다.

대부분의 새 코드는 표준 라이브러리 외부에 있어야 하며 `go` 도구의 `get` 명령을 통해 액세스할 수 있어야 합니다. 이러한 코드에는 자체 유지관리자, 릴리스 주기 및 호환성 보장이 있을 수 있습니다. 사용자는 godoc.org에서 패키지를 찾고 해당 문서를 읽을 수 있습니다.

표준 라이브러리에는 `로그/시스템` 로그와 같이 실제로는 속하지 않는 부분이 있지만, Go 1 호환성 약속 때문에 라이브러리의 모든 것을 계속 유지 관리하고 있습니다. 하지만 대부분의 새로운 코드는 다른 곳에 보관하도록 권장합니다.

구현

컴파일러를 빌드하는 데 어떤 컴파일러 기술이 사용되나요?

바둑용 프로덕션 컴파일러는 여러 가지가 있으며, 다양한 플랫폼을 위해 개발 중인 컴파일러도 많습니다.

기본 컴파일러인 gc는 `Go` 명령에 대한 지원의 일환으로 Go 배포에 포함되어 있습니다. `Gc`는 원래 부트스트랩의 어려움 때문에 C로 작성되었습니다. Go 환경을 설정하려면 Go 컴파일러가 필요하기 때문입니다. 하지만 상황이 발전하여 Go 1.5 릴리스부터 컴파일러는 Go 프로그램이 되었습니다. 컴파일러는 이 설계 문서와 강연에서 설명한 대로 자동 번역 도구를 사용하여 C에서 Go로 변환되었습니다. 따라서 컴파일러는 이제 "자체 호스팅"이 되었으며, 이는 부트스트래핑에 직면해야 한다는 것을 의미합니다.

문제가 있습니다. 해결책은 일반적으로 작동하는 C 설치와 마찬가지로 작동하는 Go 설치가 이미 준비되어 있는 것입니다. 소스에서 새로운 Go 환경을 불러오는 방법에 대한 이야기는 여기와 여기에 설명되어 있습니다.

Gc는 재귀적 하강 구문 분석기를 사용하여 Go로 작성되며, 역시 Go로 작성되었지만 Plan 9 로더를 기반으로 하는 사용자 정의 로더를 사용하여 ELF/Mach-O/PE 바이너리를 생성합니다.

`Gccgo` 컴파일러는 재귀적 하강 구문 분석기가 표준 GCC 백엔드에 결합된 C++로 작성된 프론트엔드입니다. 실험적인 LLVM 백엔드는 동일한 프론트엔드를 사용하고 있습니다.

프로젝트 초기에는 `gc`에 LLVM을 사용하는 것을 고려했지만, 너무 크고 느려서 성능 목표를 달성하기 어렵다고 판단했습니다. 돌이켜보면 LLVM으로 시작했다면 Go에 필요하지만 표준 C 설정에 포함되지 않는 일부 ABI 및 스택 관리와 같은 관련 변경 사항을 도입하기가 더 어려웠을 것입니다.

Go는 원래의 목표는 아니었지만 Go 컴파일러를 구현하기에 훌륭한 언어임이 밝혀졌습니다. 처음부터 자체 호스팅이 아니었기 때문에 Go의 설계는 네트워크 서버라는 원래 사용 사례에 집중할 수 있었습니다. 초기에 Go가 자체적으로 컴파일되어야 한다고 결정했다면 컴파일러 구축에 더 초점을 맞춘 언어가 되었을 수도 있는데, 이는 가치 있는 목표이기는 하지만 처음의 목표가 아니었습니다.

아직은 사용하지 않지만(아직은?) 네이티브 렉서 및 구문 분석기는 이동 중에도 사용할 수 있습니다. 패키지와 기본 유형 검사기도 있습니다.

런타임 지원은 어떻게 구현되나요?

부트스트랩 문제로 인해 런타임 코드는 원래 대부분 C(약간의 어셈블러 포함)로 작성되었지만 이후 Go로 번역되었습니다(일부 어셈블러 비트 제외). Gccgo의 런타임 지원은 `glibc`를 사용합니다. `gccgo` 컴파일러는 최근 골드 링커의 수정으로 지원되는 세그먼트 스택이라는 기술을 사용하여 고루틴을 구현합니다. `Gollvm`도 마찬가지로 해당 LLVM 인프라를 기반으로 구축됩니다.

제 사소한 프로그램이 왜 이렇게 큰 바이너리인가요?

`gc` 툴체인의 링커는 기본적으로 정적으로 링크된 바이너리를 생성합니다. 따라서 모든 Go 바이너리에는 동적 유형 검사, 리플렉션, 심지어 패닉 타임 스택 추적을 지원하는 데 필요한 런타임 유형 정보와 함께 Go 런타임이 포함됩니다.

Linux에서 gcc를 사용하여 정적으로 컴파일되고 링크된 간단한 C "hello, world" 프로그램은 `printf` 구현을 포함하여 약 750kB입니다. `fmt.Printf`를 사용하는 동등한 Go 프로그램의 무게는 몇 메가바이트이지만, 여기에는 더 강력한 런타임 지원과 유형 및 디버깅 정보가 포함되어 있습니다.

`gc`로 컴파일된 Go 프로그램을 `-ldflags=-w` 플래그와 연결하여 DWARF 생성을 비활성화하면 바이너리에서 디버깅 정보를 제거하지만 다른 기능의 손실은 없습니다. 이렇게 하면 바이너리 크기를 크게 줄일 수 있습니다.

사용하지 않는 변수/임포트에 대한 이러한 불만을 중지할 수 있나요?

사용되지 않는 변수가 있으면 버그를 나타낼 수 있고, 사용되지 않는 임포트는 컴파일 속도를 늦출 뿐이며, 시간이 지남에 따라 코드와 프로그래머가 누적되면 그 영향은 상당히 커질 수 있습니다. 이러한 이유로 Go는 사용하지 않는 변수나 임포트가 있는 프로그램의 컴파일을 거부하여 단기적인 편의성과 장기적인 빌드 속도 및 프로그램 명확성을 맞바꾸고 있습니다.

하지만 코드를 개발할 때 이러한 상황을 일시적으로 생성하는 것이 일반적이며 프로그램이 컴파일되기 전에 편집해야 하는 것은 번거로울 수 있습니다.

일부 사용자는 이러한 검사를 해제하거나 최소한 경고로 축소하는 컴파일러 옵션을 요청했습니다. 하지만 컴파일러 옵션은 언어의 의미론에 영향을 미쳐서는 안 되며, Go 컴파일러는 경고를 보고하지 않고 컴파일을 방해하는 오류만 보고하기 때문에 이러한 옵션은 추가되지 않았습니다.

경고가 없는 데에는 두 가지 이유가 있습니다. 첫째, 불평할 가치가 있다면 코드에서 수정할 가치가 있습니다. (고칠 가치가 없다면 언급할 가치가 없습니다.) 둘째, 컴파일러가 경고를 생성하도록 하면 컴파일을 시끄럽게 만들 수 있는 약한 경우에 대해 경고하여 수정해야 하는 실제 오류를 가리는 구현을 장려할 수 있습니다.

하지만 상황을 해결하는 방법은 간단합니다. 빈 식별자를 사용하여 개발하는 동안 사용하지 않는 항목을 계속 유지하세요.

```
"사용되지 않음" 가져오기

// 이 선언은 임포트를 참조하여 사용된 것으로 표시합니다.
// 항목으로 이동합니다.
var _ = unused.Item // TODO: 커밋하기 전에 삭제하세요!

함수 main() {
    debugData := debug.Profile()
    _ = debugData // 디버깅 중에만 사용됩니다.
    ....
}
```

요즘 대부분의 Go 프로그래머는 Go 소스 파일을 올바른 임포트를 갖도록 자동으로 다시 작성하여 실제로 사용하지 않는 임포트 문제를 없애주는 도구인 goimports를 사용합니다. 이 프로그램은 대부분의 편집기에 쉽게 연결하여 Go 소스 파일을 작성할 때 자동으로 실행됩니다.

바이러스 검사 소프트웨어가 내 Go 배포판 또는 컴파일된 바이너리가 감염되었다고 생각하는 이유는 무엇인가요?

이는 특히 Windows 시스템에서 흔히 발생하며 거의 항상 오탐지입니다. 상용 바이러스 검사 프로그램은 다른 언어로 컴파일된 바이너리만큼 자주 볼 수 없는 Go 바이너리의 구조로 인해 혼동하는 경우가 많습니다.

방금 Go 배포판을 설치했는데 시스템에서 감염되었다고 보고한다면, 이는 분명 실수입니다. 정말 철저하게 확인하려면 다운로드 페이지의 체크섬과 비교하여 다운로드를 확인할 수 있습니다.

어떤 경우든 보고서에 오류가 있다고 생각되면 바이러스 스캐너 공급업체에 버그를 신고하세요. 시간이 지나면 바이러스 스캐너가 바둑 프로그램을 이해하는 방법을 배울 수 있을지도 모릅니다.

성능

벤치마크 X에서 Go의 성능이 좋지 않은 이유는 무엇인가요?

Go의 설계 목표 중 하나는 비슷한 프로그램의 경우 C의 성능에 근접하는 것이지만, 일부 벤치마크에서는 상당히 저조한 성능을 보이고 있으며, 그 중 몇 가지 벤치마크는 golang.org/x/exp/shootout에 포함되어 있습니다.
가장 느린 것은 Go에서 비슷한 성능의 버전을 사용할 수 없는 라이브러리에 의존하는 경우입니다. 예를 들어, pidigits.go는 다중 정밀도 수학 패키지에 의존하며, C 버전은 Go와 달리 최적화된 어셈블러로 작성된 GMP를 사용합니다. 정규식에 의존하는 벤치마크(예: regex-dna.go)는 기본적으로 Go의 기본 정규식 패키지와 PCRE와 같이 고도로 최적화된 성숙한 정규식 라이브러리를 비교하는 것입니다.

벤치마크 게임은 광범위한 튜닝을 통해 승리하며, 대부분의 벤치마크의 Go 버전은 주의가 필요합니다. 비슷한 C와 Go 프로그램을 측정해 보면(reverse- complement.go가 한 예입니다), 이 제품군이 나타내는 것보다 두 언어의 원시 성능이 훨씬 더 가깝다는 것을 알 수 있습니다.

여전히 개선의 여지가 있습니다. 컴파일러는 훌륭하지만 더 개선될 여지가 있고, 많은 라이브러리가 대대적인 성능 개선 작업이 필요하며, 가비지 수집기는 아직 충분히 빠르지 않습니다. (빠르다고 해도 불필요한 가비지를 생성하지 않도록 주의하는 것만으로도 큰 효과를 볼 수 있습니다.)

어쨌든 바둑은 종종 경쟁이 매우 치열할 수 있습니다. 언어와 도구가 발전함에 따라 많은 프로그램의 성능이 크게 향상되었습니다. 유익한 예는 바둑 프로그램 프로파일링에 관한 블로그 게시물을 참조하세요.

C에서 변경된 사항

구문이 C와 다른 이유는 무엇인가요?

선언 구문 외에는 큰 차이가 없으며 두 가지 욕구에서 비롯된 것입니다. 첫째, 구문이 너무 많은

필수 키워드, 반복 또는 아르카나 없이 가볍게 느껴져야 합니다. 둘째, 이 언어는 분석하기 쉽도록 설계되었으며 기호 테이블 없이도 구문 분석이 가능합니다. 따라서 디버거, 의존성 분석기, 자동화된 문서 추출기, IDE 플러그인 등의 도구를 훨씬 쉽게 구축할 수 있습니다. C와 그 자손은 이 점에서 매우 어렵기로 악명이 높습니다.

선언이 왜 거꾸로 되나요?

C에서는 변수를 유형을 나타내는 표현식처럼 선언한다는 개념이 있는데, 이는 좋은 생각이지만 유형과 표현식 문법이 잘 섞이지 않고 결과가 혼란스러울 수 있으므로 함수 포인터를 생각해 보세요. Go는 대부분 표현식과 타입 문법을 분리하고 있으며, 이는 일을 단순화합니다(포인터에 접두사 * 를 사용하는 것은 규칙을 증명하는 예외입니다). C에서 선언

```
int* a, b;
```

는 a를 포인터로 선언하지만 b는 선언하지 않습니다.

```
var a, b *int
```

는 둘 다 포인터로 선언합니다. 이것이 더 명확하고 규칙적입니다. 또한 `:=` 짧은 선언 형식은 전체 변수 선언이 `:=`와 같은 순서로 표시되어야 한다고 주장합니다.

```
var a uint64 = 1
```

와 동일한 효과가 있습니다.

```
a := uint64(1)
```

또한 표현 문법뿐만 아니라 유형에 대한 고유한 문법을 사용하여 구문 분석이 간소화되며, `func`, `chan`과 같은 키워드를 통해 명확하게 구분할 수 있습니다.

자세한 내용은 Go의 선언 구문에 대한 문서를 참조하세요.

포인터 연산이 없는 이유는 무엇인가요?

안전. 포인터 연산이 없으면 잘못 성공하는 잘못된 주소를 절대 도출할 수 없는 언어를 만들 수 있습니다. 컴파일러와 하드웨어 기술은 배열 인덱스를 사용하는 루프가 포인터 연산을 사용하는 루프만큼 효율적일 수 있을 정도로 발전했습니다. 또한 포인터 연산이 없기 때문에 가비지 컬렉터의 구현이 단순화될 수 있습니다.

표현식이 아닌 `++`와 `--` 문은 왜 사용하나요? 그리고 왜 접두사가 아닌 접미사인가요?

포인터 산술이 없으면 접두사 전후 증분 연산자의 편의성이 떨어집니다. 표현식 계층 구조에서 이들을 완전히 제거하면 표현식 구문이 단순화되고 `++`와 `--`의 평가 순서와 관련된 복잡한 문제(`f(i++)`와 `p[i] = q[++i]`를 생각해 보세요)도 제거됩니다. 이러한 단순화는 매우 중요합니다. 포스트픽스와 프리픽스 중 어느 쪽이든 잘 작동하지만 포스트픽스 버전이 더 전통적이며, 프리픽스에 대한 주장은 아이러니하게도 이름에 포스트픽스 증분이 포함된 언어용 라이브러

리인 STL에서 비롯되었습니다.

중괄호는 있는데 세미콜론이 없는 이유는 무엇인가요? 그리고 왜 다음 줄에 여는 중괄호를 넣을 수 없나요?

Go는 C 계열 언어를 다뤄본 프로그래머라면 익숙한 구문인 중괄호를 사용해 문을 그룹화합니다. 그러나 세미콜론은 사람이 아닌 파서를 위한 것이므로 가능한 한 세미콜론을 없애고 싶었습니다. 이 목표를 달성하기 위해 Go는 BCPL의 트릭을 차용했습니다. 문장을 구분하는 세미콜론은 형식 문법에 있지만, 구문 분석기가 미리 보지 않고 자동으로

줄을 추가하여 문장의 끝이 될 수 있습니다. 이 방법은 실제로는 매우 잘 작동하지만 중괄호 스타일을 강제하는 효과가 있습니다. 예를 들어 함수의 여는 중괄호는 한 줄에 단독으로 표시할 수 없습니다.

어떤 사람들은 다음 줄에 버팀대가 살 수 있도록 렉서가 미리 살펴봐야 한다고 주장합니다. 저희는 동의하지 않습니다. Go 코드는 `gofmt`에 의해 자동으로 형식이 지정되도록 되어 있기 때문에 *어떤* 스타일을 선택해야 합니다. 그 스타일은 C나 Java에서 사용했던 스타일과 다를 수 있지만, Go는 다른 언어이고 gofmt의 스타일은 다른 어떤 언어와도 비슷합니다. 더 중요한 것은, 아니 훨씬 더 중요한 것은 모든 Go 프로그램에 대해 프로그래밍적으로 강제된 단일 형식의 이점이 특정 스타일의 단점보다 훨씬 더 크다는 것입니다. 또한 Go의 스타일은 Go의 대화형 구현에서 특별한 규칙 없이 한 번에 한 줄씩 표준 구문을 사용할 수 있다는 것을 의미합니다.

왜 쓰레기 수거를 하나요? 비용이 너무 많이 들지 않을까요?

시스템 프로그램에서 부기 작업의 가장 큰 원인 중 하나는 할당된 객체의 수명을 관리하는 것입니다. 이 작업을 수동으로 수행하는 C와 같은 언어에서는 프로그래머의 시간이 상당히 많이 소모될 수 있으며, 종종 악성 버그의 원인이 되기도 합니다. C++나 Rust와 같이 이를 지원하는 메커니즘을 제공하는 언어에서도 이러한 메커니즘은 소프트웨어 설계에 상당한 영향을 미칠 수 있으며, 종종 자체적으로 프로그래밍 오버헤드를 추가하기도 합니다. 저희는 이러한 프로그래머 오버헤드를 제거하는 것이 중요하다고 생각했고, 지난 몇 년간 가비지 컬렉션 기술이 발전하면서 충분히 저렴하고 충분히 낮은 지연 시간으로 구현할 수 있어 네트워크 시스템에 적합한 접근 방식이 될 수 있다는 확신을 갖게 되었습니다.

동시 프로그래밍의 어려움은 대부분 객체 수명 문제에 뿌리를 두고 있습니다. 객체가 스레드 간에 전달될 때 객체가 안전하게 해제되도록 보장하는 것이 번거로워집니다. 자동 가비지 컬렉션을 사용하면 동시 코드를 훨씬 쉽게 작성할 수 있습니다. 물론 동시 환경에서 가비지 컬렉션을 구현하는 것 자체가 어려운 일이지만, 모든 프로그램에서 가비지 컬렉션을 구현하는 것보다 한 번만 구현하는 것이 모두에게 도움이 됩니다.

마지막으로, 동시성을 제외하고 가비지 컬렉션을 사용하면 인터페이스 전반에서 메모리 관리 방법을 지정할 필요가 없으므로 인터페이스가 더 간단해집니다.

리소스 관리 문제에 새로운 아이디어를 제공하는 Rust와 같은 언어의 최근 작업이 잘못되었다는 뜻은 아닙니다. 저희는 이러한 작업을 장려하며 그 발전 방향을 지켜보는 것이 기대됩니다. 그러나 Go는 가비지 컬렉션을 통해 객체 수명을 다루고 가비지 컬렉션만 사용하는 보다 전통적인 접근 방식을 취합니다.

현재 구현은 마크 앤 스윕 수집기입니다. 머신이 멀티프로세서인 경우 수집기는 메인 프로그램과 병렬로 별도의 CPU 코어에서 실행됩니다. 최근 몇 년 동안 수집기에 대한 주요 작업을 통해 대규모 힙의 경우에도 일시 중지 시간이 밀리초 이하로 줄어들어 네트워크 서버에서 가비지 수집에 대한 주요 반대 사항 중 하나가 제거되었습니다. 알고리즘을 개선하고, 오버헤드와 지연 시간을 더욱 줄이고, 새로운 접근 방식을 모색하기 위한 작업이 계속되고 있습니다. 2018 ISMM 기조연설에서 Go 팀의 Rick Hudson이 지금까지의 진행 상황을 설명하고 향후 몇 가지 접근 방식을 제안합니다.

성능과 관련해서는 가비지 컬렉션 언어에서 일반적으로 사용되는 것보다 훨씬 더 많은 메모리 레이아웃과 할당을 프로그래머가 제어할 수 있다는 점을 명심하세요. 주의 깊은 프로그래머는 언어를 잘 사용하여 가비지 컬렉션 오버헤드를 크게 줄일 수 있습니다. Go의 프로파일링 도구 데모를 포함한 작업 예제는 Go 프로그램 프로파일링에 대한 기사를 참조하세요.