(https://github.com/rust-la새 사이트에 접속하세요! (https://www.rust-lang.org/)

자주 묻는 질문

이 페이지는 Rust 프로그래밍 언어에 대한 일반적인 질문에 답변하기 위해 존재합니다. 이 페이지는 언어에 대한 완전한 가이드가 아니며 언어를 가르치기 위한 도구도 아닙니다. 이 문서는 Rust 커뮤니티에서 자주 반복되는 질문에 대한 답변과 Rust의 일부 설계 결정에 대한 논리를 명확히 하기 위한 참고 자료입니다.

자주 묻는 질문이나 중요한 질문이 여기에 잘못 답변되어 있지 않다고 생각되면 언제든지 수정할 수 있도록 도와주세요(https://github.com/rust-lang/rust-www/blob/master/CONTRIBUTING.md).

목차

러스트 프로젝트새로운 사이트를 확인하세요! (https://www.rust-lang.org/)

이 프로젝트의 목표는 무엇인가요? (#이 프로젝트의 목표는 무엇인가요?)

안전하고 동시적이며 실용적인 시스템 언어를 설계하고 구현합니다.

이 수준의 추상화와 효율성을 갖춘 다른 언어가 만족스럽지 못하기 때문에 Rust가 존재합니다. 특히

1. 안전에 대한 관심이 너무 적습니다.
2. 동시성 지원이 원활하지 않습니다.
3. 실용적인 어포던스가 부족합니다.
4. 리소스를 제한적으로 제어할 수 있습니다.

Rust는 효율적인 코드와 편안한 추상화 수준을 모두 제공하면서 이 네 가지 사항을 모두 개선하는 대안으로 존재합니다.

이 프로젝트가 Mozilla에 의해 제어되나요? (#is-this-project-controlled-by-mozilla)

아니요. Rust는 2006년에 그레이든 호어의 파트타임 부업 프로젝트로 시작되어 3년 이상 유지되었습니다. 기본 테스트를 실행하고 핵심 개념을 시연할 수 있을 만큼 언어가 성숙해지자 2009년에 Mozilla가 참여했습니다. Rust는 여전히 Mozilla의 후원을 받고 있지만, 전 세계 여러 곳에서 모인 다양한 애호가 커뮤니티에 의해 개발되고 있습니다. Rust 팀(https://www.rust-lang.org/team.html)은 Mozilla 회원과 비회원 모두로 구성되어 있으며, 지금까지 1,900명 이상의 고유한 기여자(https://github.com/rust-lang/rust/)가 GitHub의 `rust`에 참여했습니다.

프로젝트 거버넌스(https://github.com/rust-lang/rfcs/blob/master/text/1068-rust- governance.md)와 관련하여 Rust는 프로젝트의 비전과 우선순위를 설정하고 글로벌 관점에서 프로젝트를 안내하는 핵심 팀에 의해 관리됩니다. 또한 핵심 언어, 컴파일러, Rust 라이브러리, Rust 도구, 공식 Rust 커뮤니티 운영 등 특정 관심 영역의 개발을 안내하고 촉진하는 하위 팀도 있습니다. 이러한 각 영역의 디자인은 RFC 프로세스(https://github.com/rust-lang/rfcs)를 통해 발전됩니다. RFC가 필요하지 않은 변경사항의 경우, `rustc` 리포지토리(https://github.com/rust-lang/rust)의 풀 리퀘스트를 통해 결정이 이루어집니다.

Rust의 비목표는 무엇인가요? (#무엇-어떤-비-목표)

1. 저희는 특별히 최첨단 기술을 사용하지 않습니다. 오래되고 확립된 기술이 더 좋습니다.
2. 표현력, 미니멀리즘, 우아함을 다른 목표보다 우선시하지 않습니다. 이러한 목표는 바람직하지만 종속적인 목표입니다.

3. C++ 또는 다른 언어의 모든 기능을 다루려는 의도는 없습니다. Rust는 대부분의 기능을 제공해야 합니다.
4. 트위터는 100% 정적이고, 100% 안전하며, 100% 효과적이거나 다른 어떤 의미에서도 지나치게 독단적일 의도가 없습니다. 장단점은 존재합니다.
5. 저희는 Rust가 "가능한 모든 플랫폼"에서 실행될 것을 요구하지 않습니다. 결국 널리 사용되는 하드웨어 및 소프트웨어 플랫폼에서 불필요한 성능 저하 없이 작동해야 합니다.

Mozilla는 어떤 프로젝트에서 Rust를 사용하나요? (#어떻게-모질라-사용-러스트)

주요 프로젝트는 Mozilla가 개발중인 실험적인 브라우저 엔진인 Servo(https://github.com/servo/servo)입니다.또한,이프로젝트는Firefox에 대한 자세한 내용(https://www.mozilla.org/show_bug.cgi?id=1135640)을 확인할 수 있습니다.

대규모 Rust 프로젝트에는 어떤 예가 있나요? (#대규모 러스트 프로젝트에는 어떤 예가 있나요?)

현재 가장 큰 두 개의 오픈 소스 Rust 프로젝트는 Servo(https://github.com/servo/servo)와 Rust 컴파일러(https://github.com/rust-lang/rust)입니다.

또 누가 Rust를 사용하나요? (#누가-다른-누구를-사용하고-러스트)

점점 늘어나는 조직! (friends.html)

Rust를 쉽게 사용하려면 어떻게 해야 하나요? (#하우-캔-아이-트라이-쉬운-러스트)

Rust를 사용해 보는 가장 쉬운 방법은 Rust 코드를 작성하고 실행할 수 있는 온라인 앱인 플레이펜(https://play.rust-lang.org/)을 이용하는 것입니다. 시스템에서 Rust를 사용해 보려면 설치(https://www.rust- lang.org/install.html)한 다음 책에 있는 추측 게임(https://doc.rust-lang.org/stable/book/second- edition/ch02-00-guessing-game-tutorial.html) 튜토리얼을 따라하세요.

녹 문제에 대한 도움을 받으려면 어떻게 해야 하나요? (#어떻게-녹-문제에-도움을-받을-수-있습니다)

여러 가지 방법이 있습니다. 할 수 있습니다:

- 공식 Rust 사용자 포럼인 users.rust-lang.org(https://users.rust-lang.org/)에 게시하세요.
- 공식 Rust IRC 채널(https://chat.mibbit.com/?server=irc.mozilla.org&channel=%23rust)(irc.mozilla.org의 #rust)에서 문의하세요.
- 스택 오버플로(https://stackoverflow.com/questions/tagged/rust)에 "rust" 태그와 함께 질문하
- 기• 비공식 러스트 서브 레딧인 /r/rust(https://www.reddit.com/r/rust)에 게시하기

시간이 지남에 따라 Rust가 이렇게 많이 변한 이유는 무엇인가요? (#why-has-rust-changed-so-much)

Rust는 안전하면서도 사용 가능한 시스템 프로그래밍 언어를 만들겠다는 목표에서 시작되었습니다. 이 목표를 달성하기 위해 많은 아이디어를 탐색했으며, 그 중 일부는 유지(수명, 특성)되고 일부는 폐기(타입스테이트 시스템, 그린 스레딩)되었습니다. 또한 1.0에 이르기까지 초기 설계를 업데이트하여 Rust의 기능을 최대한 활용하고 고품질의 일관된 크로스 플랫폼 API를 제공하기 위해 표준 라이브러리의 많은 부분이 다시 작성되었습니다. 이제 Rust가 1.0에 도달했으므로 언어가 "안정적"이라고 보장되며, 계속 발전할 수 있지만 현재 Rust에

서 작동하는 코드는 향후 릴리스에서도 계속 작동해야 합니다.

Rust 언어 버전 관리는 어떻게 이루어지나요? (#어떻게-러스트-언어-버전이-작동하나요)

Rust의 언어 버전 관리는 SemVer(http://semver.org/)를 따르며, 컴파일러 버그 수정, 안전 취약점 패치, 디스패치 또는 유형 추론을 변경하여 추가 어노테이션이 필요한 경우에만 마이너 버전에서 안정적인 API의 하위 호환성 변경이 허용됩니다. 마이너 버전 변경에 대한 자세한 지침은 언어(https://github.com/rust-lang/rfcs/blob/master/text/1122-language-semver.md) 및 표준 라이브러리(https://github.com/rust-lang/rfcs/blob/master/text/1105-api-evolution.md) 모두에 대한 승인된 RFC로 확인할 수 있습니다.

Rust는 안정, 베타, 야간 릴리스의 세 가지 "릴리스 채널"을 유지합니다. 안정 버전과 베타 버전은 6주마다 업데이트되며, 현재 야간 버전이 새로운 베타 버전이 되고, 현재 베타 버전이 새로운 안정 버전이 됩니다. 불안정 또는 기능 게이트 뒤에 숨겨진 것으로 표시된 언어 및 표준 라이브러리 기능은 다음과 같은 경우에만 사용할 수 있습니다.

야간 릴리스 채널에서 확인할 수 있습니다. 새로운 기능은 불안정한 상태로 출시되며, 핵심 팀과관련하위팀의승인을 받으면 "게이트 해제"됩니다. 안정적인 채널을 위해 강력한하위호환성을 보장하는 로빙이이루어집니다.

자세한 내용은 Rust 블로그 게시물 "결과물로서의 안정성"을 참조하세요. (http://blog.rust-lang.org/2014/10/30/Stability.html)

베타 또는 안정 채널에서 불안정한 기능을 사용할 수 있나요? (#불안정한 기능을 베타 또는 안정 채널에서 사용할 수 있나요?)

아니요, 불가능합니다. Rust는 베타 및 안정 채널에서 제공되는 기능의 안정성에 대한 강력한 보증을 제공하기 위해 열심히 노력하고 있습니다. 무언가가 불안정하다는 것은 아직 그런 보장을 제공할 수 없다는 뜻이며, 사람들이 그 기능에 의존하는 것을 원하지 않는다는 뜻입니다. 이를 통해 야간 공개 채널에서 변화를 시도할 수 있는 기회를 제공하는 동시에 안정성을 원하는 분들을 위한 강력한 보장을 유지할 수 있습니다.

베타 및 안정 채널은 6주마다 업데이트되며, 이따금씩 수정 사항이 베타 버전에 적용되기도 합니다. 야간 릴리스 채널을 이용하지 않고 기능이 제공되기를 기다리는 경우, 이슈 트래커에서 `B-불안정` (https://github.com/rust- lang/rust/issues?q=is%3A이슈+is%3A오픈+추적+레이블%3AB-불안정) 태그를 확인하여 해당 추적 이슈를 찾을 수 있습니다.

"기능 게이트"란 무엇인가요? (#기능-게이트란 무엇인가요?)

"기능 게이트"는 컴파일러, 언어 및 표준 라이브러리의 기능을 안정화하기 위해 Rust가 사용하는 메커니즘입니다. "게이트"된 기능은 야간 릴리스 채널에서만 액세스할 수 있으며, `#[feature]` 속성 또는 `-Z unstable-options` 명령줄 인수를 통해 명시적으로 활성화된 경우에만 액세스할 수 있습니다. 기능이 안정화되면 안정 릴리스 채널에서 사용할 수 있게 되며 명시적으로 사용 설정할 필요가 없습니다. 이 시점에서 해당 기능은 "게이트 해제"된 것으로 간주됩니다. 기능 게이트를 사용하면 개발자가 실험적인 기능을 안정 언어로 제공하기 전에 개발 중에 테스트할 수 있습니다.

왜 듀얼 MIT/ASL2 라이선스인가? (#왜-듀얼-MIT-ASL2-라이센스인가)

Apache 라이선스에는 특허 공격에 대한 중요한 보호 기능이 포함되어 있지만 GPL 버전 2와는 호환되지 않습니다. Rust를 GPL2와 함께 사용하는 데 문제가 발생하지 않도록 하기 위해 대체로 MIT 라이선스가 적용됩니다.

MPL이나 3중 라이선스가 아닌 BSD 스타일의 허가 라이선스가 필요한 이유는 무엇인가요? (#왜-퍼머시브-라이선스인가)

이는 부분적으로는 원 개발자(그레이든)의 선호도 때문이고, 부분적으로는 언어가 웹 브라우저와 같은 제품보다 더 많은 사용자와 다양한 임베딩 및 최종 사용처를 가진 경향이 있다는 사실 때문입니다. 저희는 가능한 한 많은 잠재적 기여자의 참여를 유도하고자 합니다.

성능

러스트는 얼마나 빠른가요? (#how-fast-is-rust)

빠릅니다! Rust는 이미 여러 벤치마크(예: 벤치마크 게임(https://benchmarksgame-team.pages.debian.net/benchmarksgame/faster/rust.html) 및 기타(https://github.com/kostya/benchmarks))에서 관용적 C 및 C++와 경쟁하고 있습니다.

C++와 마찬가지로 Rust는 제로 코스트 추상화(http://blog.rust-lang.org/2015/05/11/traits.html)를핵심원칙중 하나로 삼고 있습니다. 즉, Rust는 전통적인 의미의 런타임 시스템에서 발생하는 오버헤드가전혀없으며,트레이보스태치도 없습니다.

Rust가 LLVM을 기반으로 구축되고 LLVM의 관점에서 Clang을 닮기 위해 노력한다는 점을 감안할 때, 모든 LLVM 성능 개선은 Rust에도 도움이 됩니다. 장기적으로는 Rust 유형 시스템의 풍부한 정보로 인해 C/C++ 코드에서는 어렵거나 불가능한 최적화가 가능해질 것입니다.

녹 쓰레기는 수거되나요? (#is-rust-garbage-collected)

Rust의 핵심 혁신 중 하나는 가비지 컬렉션 *없이* 메모리 안전(세그폴트 없음)을 보장하는 것입니다.

GC를 피함으로써 Rust는 예측 가능한 리소스 정리, 메모리 관리 오버헤드 감소, 런타임 시스템 없음 등 다양한 이점을 제공할 수 있습니다. 이러한 모든 특성 덕분에 Rust는 간결하고 임의의 컨텍스트에 쉽게 임베드할 수 있으며, GC가 있는 언어와 Rust 코드를 훨씬 쉽게 통합할 수 있습니다 (http://calculist.org/blog/2015/12/23/neon-node-rust/).

Rust는 소유권 및 차용 시스템을 통해 GC의 필요성을 피하지만, 동일한 시스템이 일반적인 리소스 관리 (http://blog.skylight.io/rust- means-never-having-to-close-a-socket/) 및 동시성(http://blog.rust-lang.org/2015/04/10/Fearless- Concurrency.html)을 비롯한 여러 가지 다른 문제에도 도움이 됩니다.

단일 소유권으로 충분하지 않은 경우, Rust 프로그램은 GC 대신 표준 참조 카운팅 스마트 포인터 유형인 `Rc`(https://doc.rust-lang.org/std/rc/struct.Rc.html)와 스레드 안전에 대응하는 `Arc`(https://doc.rust-lang.org/std/sync/struct.Arc.html)에 의존합니다.

그러나 향후 확장 기능으로 가비지 수집 *옵션을* 검토하고 있습니다. 그 목표는 스파이더몽키 (https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey) 및 V8(https://developers.google.com/v8/?hl=en) JavaScript 엔진에서 제공하는 것과 같은 가비지 수집 런타임과 원활하게 통합하는 것입니다. 마지막으로, 일부 사람들은 컴파일러 지원 없이 순수한 Rust 가비지 수집기 (https://manishearth.github.io/blog/2015/09/01/designing-a-gc- in-rust/)를 구현하는 것을 검토했습니다.

내 프로그램이 느린 이유는 무엇인가요? (#왜-내-프로그램이-느린가요?)

최적화는 컴파일 속도를 늦추고 일반적으로 개발 중에 바람직하지 않으므로, 요청하지 않는 한 Rust 컴파일러는 최적화를 사용하여 컴파일하지 않습니다(https://users.rust-lang.org/t/why-does- cargo-build-not-optimise-by-default/4150/3).

카고로 컴파일하는 경우 `--release` 플래그를 사용한다. `rustc`로 직접 컴파일하는 경우, `-O` 플래그를 사용합니다. 이 중 어느 것이든 최적화를 켭니다.

Rust 컴파일이 느린 것 같습니다. 왜 그런가요? (#why-is-rustc-slow)

코드 번역 및 최적화. Rust는 효율적인 머신 코드로 컴파일되는 높은 수준의 추상화를 제공하며, 이러한 번역은 특히 최적화할 때 실행하는 데 시간이 걸립니다.

그러나 Rust의 컴파일 시간은 생각만큼 나쁘지 않으며 개선될 것이라고 믿을 만한 이유가 있습니다. 비슷한 규모의 프로젝트를 C++와 Rust로 비교했을 때 전체 프로젝트의 컴파일 시간은 일반적으로 비슷한 것으로 알려져 있습니다. Rust 컴파일이 느리다는 일반적인 인식은 상당 부분 C++와 Rust의 *컴파일 모델* 차이 때문입니다: C++의 컴파일 단위는 파일인 반면, Rust는 여러 파일로 구성된 크레이트입니다. 따라서 개발 중에 단일 C++ 파일을 수정하면 다음과 같은 문제가 발생할 수 있습니다.

를 사용하면 Rust보다 훨씬 적은 재컴파일이 필요합니다. 현재 컴파일러를 리팩터링하여 증분컴파일을 도입하기 위한 대대적인 노력이 진행 중이며, 이는 Rust에 C++ 모델의 컴파일 시간 이점을 제공할 것입니다 (https://github.com/rust-lang/rfcs/blob/master/text/1298-incremental-compilation.md).

컴파일 모델 외에도 Rust의 언어 설계 및 컴파일러 구현에는 컴파일 시간 성능에 영향을 미치는 몇 가지 다른 측면이 있습니다.

첫째, Rust는 적당히 복잡한 타입 시스템을 가지고 있으며 런타임에 Rust를 안전하게 만드는 제약 조건을 적용하는 데 무시할 수 없는 양의 컴파일 시간을 소비해야 합니다.

둘째, Rust 컴파일러는 오랜 기술 부채에 시달리고 있으며, 특히 품질이 좋지 않은 LLVM IR을 생성하여 LLVM이 "수정"하는 데 시간을 소비해야 합니다. Rust 컴파일러에 MIR(https://github.com/rust-lang/rfcs/blob/master/text/1211-mir.md)이라는 새로운 내부 표현을 추가하면 더 많은 최적화를 수행하고 생성된 LLVM IR의 품질을 개선할 수 있지만, 아직 이 작업은 이루어지지 않았습니다.

셋째, 코드 생성에 LLVM을 사용하는 것은 양날의 검과도 같습니다. Rust가 세계 최고 수준의 런타임 성능을 갖출 수 있게 해주지만, LLVM은 특히 품질이 낮은 입력으로 작업할 때 컴파일 타임 성능에 초점을 맞추지 않는 큰 프레임워크입니다.

마지막으로, Rust에서 선호하는 제네릭을 단일화하는 전략(예: C++)은 빠른 코드를 생성하지만, 다른 번역 전략보다 훨씬 더 많은 코드를 생성해야 합니다. Rust 프로그래머는 특성 객체를 사용하여 동적 디스패치를 대신 사용함으로써 이러한 코드 부피를 줄일 수 있습니다.

Rust의 해시맵이 느린 이유는 무엇인가요? (#왜-러스트-해시맵이-느린가)

기본적으로 Rust의 해시맵(https://doc.rust-lang.org/stable/std/collections/struct.HashMap.html)은 다양한 워크로드에서 합리적인 성능을 제공하면서 해시 테이블 충돌 공격 (http://programmingisterrible.com/post/40620375793/hash-table-denial-of-service- 공격-재방문)을 방지하도록 설계된 SipHash(https://131002.net/siphash/) 해싱 알고리즘을 사용합니다 (https://www.reddit.com/r/rust/comments/3hw9zf/rust_hasher_comparisons/cub4oh6).

SipHash는 많은 경우 경쟁력 있는 성능을 보여주지만(http://cglab.ca/%7Eabeinges/blah/hash-rs/), 다른 해싱 알고리즘보다 현저히 느린 경우는 정수와 같은 짧은 키를 사용하는 경우입니다. 이 때문에 Rust 프로그래머들은 종종 `HashMap`(https://doc.rust- lang.org/stable/std/collections/struct.HashMap.html)에서 느린 성능을 관찰합니다. 이러한 경우 FNV 해셔(https://crates.io/crates/fnv)가 자주 권장되지만, SipHash와 동일한 충돌 저항 속성을 가지고 있지 않다는 점에 유의하세요.

통합 벤치마킹 인프라가 없는 이유는 무엇인가요? (#왜-통합 벤치마킹이 없나요?)

있지만 야간 릴리스 채널에서만 사용할 수 있습니다. 궁극적으로는 통합 벤치마크를 위한 플러그인 시스템을 구축할 계획이지만, 당분간은 현재 시스템이 불안정하다고 판단됩니다 (https://github.com/rust-lang/rust/issues/29553).

Rust는 테일 콜 최적화를 수행하나요? (#does-rust-do-tail-call-optimization)

일반적으로는 그렇지 않습니다. 테일 콜 최적화는 제한된 상황에서 수행될 수 있지만 (http://llvm.org/docs/CodeGenerator.html#sibling-call-optimization), 보장되지는 않습니다 (https://mail.mozilla.org/pipermail/rust-dev/2013-April/003557.html). 이 기능은 항상 요구되어 왔기 때문에 Rust에는 키워드 (`become`)가 예약되어 있지만, 기술적으로 가능한지 여부와 구현 여부는 아직 명확하지 않습니다. 특정 컨텍스트에서 꼬리 호출 제거를 허용하는 확장(https://github.com/rust- lang/rfcs/pull/81)이 제안되었지만 현재 연기된 상태입니다.

Rust에 런타임이 있나요? (#does-rust-have-a-runtime)

Java와 같은 언어에서 사용하는 일반적인 의미는 아니지만 Rust 표준 라이브러리의 일부는 "런타임"으로간주할수 있으며, 재[illegible]입니다.소량의 초기화 코드가있습니다 (https://github.com/rust-lang/rust/blob/33916307780495fe311fe9c080b330d266f35bfb/src/libstd/rt.rs#L43)가 사용자의 주 함수보다 먼저 실행됩니다. Rust 표준 라이브러리는 유사한 런타임 초기화를 수행하는 C 표준 라이브러리에 추가로 링크되어 있습니다(http://www.embecosm.com/appnotes/ean9/html/ch05s02.html). 표준 라이브러리 없이도 Rust 코드를 컴파일할 수 있으며, 이 경우 런타임은 C와 거의 동일합니다.

구문

왜 중괄호를 사용하나요? Rust의 구문은 왜 하스켈이나 파이썬과 같을 수 없나요? (#왜-중괄호)

블록을 나타내기 위해 중괄호를 사용하는 것은 다양한 프로그래밍 언어에서 흔히 볼 수 있는 디자인 선택이며, Rust의 일관성은 이미 이 스타일에 익숙한 사람들에게 유용합니다.

또한 중괄호를 사용하면 프로그래머는 더 유연한 구문을 사용할 수 있고 컴파일러는 더 간단한 구문 분석기를 사용할 수 있습니다.

조건에 따라 괄호를 생략할 수 있는데 왜 한 줄 블록 주위에 중괄호를 넣어야 하나요? C 스타일이 허용되지 않는 이유는 무엇인가요? (#왜-왜-괄호-주변-블록)

C에서는 `if` -문 조건문에 괄호를 필수로 사용해야 하지만 중괄호는 선택 사항으로 남겨두는 반면, Rust에서는 `if` -표현식에 대해 그 반대의 선택을 합니다. 이렇게 하면 조건문을 본문과 명확하게 분리할 수 있고, Apple의 goto fail(https://gotofail.com/) 버그와 같이 리팩토링 중에 놓치기 쉬운 오류를 유발할 수 있는 선택적 중괄호의 위험을 피할 수 있습니다.

사전에는 왜 리터럴 구문이 없나요? (#왜-사전에는-리터럴-구문이-없는가)

Rust의 전반적인 디자인 선호도는 *언어의* 크기를 제한하는 동시에 강력한 *라이브러리를* 활성화하는 것입니다. Rust는 배열과 문자열 리터럴에 대한 초기화 구문을 제공하지만, 이는 언어에 내장된 유일한 컬렉션 유형입니다. 유비쿼터스 `Vec`(https://doc.rust- lang.org/stable/std/vec/struct.Vec.html) 컬렉션 유형을 비롯한 다른 라이브러리 정의 유형은 초기화를 위해 `vec!` (https://doc.rust-lang.org/stable/std/macro.vec!.html) 매크로와 같은 매크로를 사용합니다.

컬렉션을 초기화하기 위해 Rust의 매크로 기능을 사용하는 이러한 설계 선택은 향후 다른 컬렉션에도 일반적으로 확장될 예정이며, `HashMap`(https://doc.rust-

lang.org/stable/std/collections/struct.HashMap.html) 및 `Vec`(https://doc.rust-lang.org/stable/std/vec/struct.Vec.html) 뿐만 아니라 `BTreeMap`(https://doc.rust-lang.org/stable/std/collections/struct.BTreeMap.html) 등의 다른 컬렉션 유형도 간단하게 초기화할 수 있게 될 것으로 보입니다. 한편, 컬렉션을 초기화하는 데 더 편리한 구문을 원한다면 자체 매크로(https://stackoverflow.com/questions/27582739/how-do-i-create-a-hashmap-literal)를 생성하여 제공할 수 있습니다.

암시적 반환은 언제 사용해야 하나요? (#언제-암시적-리턴을-사용해야 하나요?)

Rust는 매우 표현 지향적인 언어이며, '암시적 반환'은 이러한 설계의 일부입니다. `if` s, `match` es, normal 블록과 같은 구문은 모두 Rust에서 표현식입니다. 예를 들어, 다음 코드는 `i64`(https://doc.rust-lang.org/stable/std/primitive.i64.html)가 홀수인지 확인하여 결과를 값으로 반환합니다:

```
fn is_odd(x: i64) -> boToryl
    if x % 2 != 0 { true } else { false }
}
```

o{새 사이트로 이동! (https://www.rust-lang.org/)

이렇게 더 단순화할 수도 있습니다:

```
fn is_odd(x: i64) -> bool {
    x % 2 != 0
}
```

각 예제에서 함수의 마지막 줄은 해당 함수의 반환값입니다. 함수가 세미콜론으로 끝나는 경우 반환 유형은 반환 값이 없음을 나타내는 `()`가 된다는 점에 유의해야 합니다. 암시적 반환은 세미콜론을 생략해야 작동합니다.

명시적 반환은 함수 본문이 끝나기 전에 반환하기 때문에 암시적 반환이 불가능한 경우에만 사용됩니다. 위의 각 함수는 반환 키워드와 세미콜론으로 작성할 수도 있지만, 그렇게 하면 불필요하게 장황해지고 Rust 코드의 규칙과 일치하지 않습니다.

함수 서명이 유추되지 않는 이유는 무엇인가요? (#why-부모-함수-서명-추론됨)

Rust에서 선언은 명시적 유형으로 제공되는 반면, 실제 코드는 유형이 추론되는 경향이 있습니다. 이러한 설계에는 몇 가지 이유가 있습니다:

- 필수 선언 서명은 모듈과 크레이트 수준 모두에서 인터페이스 안정성을 강화하는 데 도움이 됩니다.
- 시그니처는 프로그래머의 코드 이해도를 높여주며, 함수의 인수 유형을 추측하기 위해 전체 크레이트에서 추론 알고리즘을 실행할 필요 없이 항상 명시적이고 가까운 곳에 있는 시그니처를 사용할 수 있습니다.
- 추론은 한 번에 하나의 함수만 살펴보기만 하면 되므로 기계적으로 추론 알고리즘이 단순화됩니다.

왜 매치가 철저해야 하나요? (#왜-매치가-철저해야 하는가)

리팩터링 및 명확성을 지원합니다.

첫째, 모든 가능성이 일치 에 의해 커버되는 경우 나중에 열거형에 변형을 추가하면 런타임에 오류가 아닌 컴파일 실패가 발생합니다. 이러한 유형의 컴파일러 지원은 Rust에서 두려움 없는 리팩터링을 가능하게 합니다.

둘째, 전수 검사는 기본 대소문자의 의미를 명시적으로 만듭니다. 일반적으로 전수 검사가 아닌 유일한 안전한 방법은 일치하는 것이 없을 경우 스레드를 패닉 상태로 만드는 것입니다. Rust의 초기 버전에서는 대소문자를 모두 일치시킬 필요가 없었으며 이는 버그의 큰 원인으로 밝혀졌습니다.

와일드카드를 사용하면 지정되지 않은 모든 대소문자를 무시할 수 있습니다:

```
match val.do_something() {
    Cat(a) => { /* ... */ }
           _=> { /* ... */ }
}
```

숫자

새 사이트를 사용해 보세요! (https://www.rust-lang.org/)

부동 소수점 연산에는 f32와 `f64` 중 어떤 것을 선호해야 하나요? (#어떤 유형의 부동 소수점을 사용해야 하나요?)

어떤 것을 사용할지는 프로그램의 목적에 따라 달라집니다.

부동 소수점 숫자의 정밀도를 최대한 높이고 싶다면 `f64`(https://doc.rust-lang.org/stable/std/primitive.f64.html)를 사용하는 것이 좋습니다. 값의 크기를 작게 유지하거나 최대한 효율적으로 사용하는 데 더 관심이 있고 값당 비트 수가 적을 때 발생할 수 있는 부정확성에 대해 걱정하지 않는다면 `f32`(https://doc.rust-lang.org/stable/std/primitive.f32.html)가 더 좋습니다. 64비트 하드웨어에서도 일반적으로 `f32`(https://doc.rust-lang.org/stable/std/primitive.f32.html)의 연산이 더 빠릅니다. 일반적인 예로, 그래픽 프로그래밍은 고성능이 필요하고 화면의 픽셀을 표현하는 데 32비트 부동 소수점만으로도 충분하기 때문에 일반적으로 `f32`(https://doc.rust- lang.org/stable/std/primitive.f32.html)를 사용합니다.

확실하지 않은 경우 `f64`(https://doc.rust-lang.org/stable/std/primitive.f64.html)를 선택하면 정확도를 높일 수 있습니다.

왜 부호를 비교하거나 해시맵 또는 `BTreeMap` 키로 사용할 수 없나요? (#왜-왜-나는-부호를-비교할-수-없나요?)

부동 소수점은 `==` , `!=` , `<` , `<=` , `>` , `>=` 연산자와 `partial_cmp()` 함수로 비교할 수 있습니다. ==와 !=는 `PartialEq`(https://doc.rust- lang.org/stable/std/cmp/trait.PartialEq.html) 특성의 일부이며, `<` , `<=` , `>` , `>=` 및 `partial_cmp()`는 `PartialOrd`(https://doc.rust-lang.org/stable/std/cmp/trait.PartialOrd.html) 특성의 일부입니다.

부동 소수점에는 전체 순서가 없으므로 부동 소수점은 `Ord`(https://doc.rust-lang.org/stable/std/cmp/trait.Ord.html) 특성의 일부인 `cmp()` 함수와 비교할 수 없습니다. 또한, floats에 대한 전체 동등 관계도 없으므로 `Eq`(https://doc.rust- lang.org/stable/std/cmp/trait.Eq.html) 특성도 구현하지 않습니다.

부동 소수점 값 `NaN`(https://en.wikipedia.org/wiki/NaN)은 다른 부동 소수점 값이나 그 자체보다 작거나 크거나 같지 않기 때문에 부동 소수점에는 총 서열이나 동일성이 없습니다.

부동 소수점은 `Eq`(https://doc.rust-lang.org/stable/std/cmp/trait.Eq.html) 또는 `Ord`(https://doc.rust-lang.org/stable/std/cmp/trait.Ord.html)를 구현하지 않으므로, `BTreeMap`(https://doc.rust-lang.org/stable/std/collections/struct.BTreeMap.html) 또는 `HashMap`(https://doc.rust-lang.org/stable/std/collections/struct.HashMap.html)과 같이 특성 경계에 이러한 특성이 필요한 유형에는 사용할 수 없습니다. 이러한 유형은 키가 전체 순서 또는 전체 동일성 관계를 제공한다고 가정하고 그렇지

않으면 오작동하기 때문에 이 점이 중요합니다.

특정 경우에 유용할 수 있는 `Ord`(https://doc.rust-lang.org/stable/std/cmp/trait.Ord.html) 및 `Eq`(https://doc.rust- lang.org/stable/std/cmp/trait.Eq.html) 구현을 제공하기 위해 `f32`(https://doc.rust- lang.org/stable/std/cmp/trait.Eq.html)와 `f64`(https://doc.rust-lang.org/stable/std/primitive.f64.html)를 감싸는 상자(https://crates.io/crates/ordered-float)가 있습니다.

숫자 유형 간 변환은 어떻게 하나요? (#how-can-i-convert-between-numeric-types)

원시 타입에 대해 간단한 형 변환을 수행하는 `as` 키워드와 여러 타입 변환에 대해 구현된(그리고 자체 타입에 대해 구현할 수 있는) `Into`(https://doc.rust-lang.org/stable/std/convert/trait.Into.html) 및 `From`(https://doc.rust- lang.org/stable/std/convert/trait.From.html) 형의 두 가지 방법이 있습니다. `Into`(https://doc.rust- lang.org/stable/std/convert/trait.Into.html) 및 `From`(https://doc.rust-

lang.org/stable/std/convert/trait.From.html) 특성은 변환이 무손실인 경우에만 구현되므로, 예를 들어 `f64::try_from(0i32)` 는 구현되지 않습니다. 반면에 `0i32 as f64` 는 구현되지 않습니다. 반면에 두 기본 유형 사이를 변환할 때와마찬가지로 필요에 따라 값을 잘라냅니다.

Rust에 증감 연산자가 없는 이유는 무엇인가요? (#why-does-rust-have-increase-and- decrease-operators)

사전 증가와 사후 증가(및 이에 상응하는 감소)는 편리하지만 상당히 복잡하기도 합니다. 평가 순서에 대한 지식이 필요하고 C와 C++에서 종종 미묘한 버그와 정의되지 않은 동작으로 이어집니다. `x = x + 1` 또는 `x += 1` 은 약간 더 길지만 모호하지 않습니다.

문자열

문자열 또는 `Vec<T>`를 슬라이스(`&str` 및 `&[T]`)로 변환하려면 어떻게 해야 하나요? (#how-to-convert-string-or-vec-to-슬라이스)

일반적으로 슬라이스가 예상되는 곳이면 어디든 `String` 또는 `Vec<T>`에 대한 참조를 전달할 수 있습니다. Deref 강제(https://doc.rust-lang.org/stable/book/second-edition/ch15-02-deref.html)를 사용하면 `&` 또는 `& mut` 와 함께 참조로 전달될 때 `String` s(https://doc.rust-lang.org/stable/std/string/struct.String.html) 및 `Vec` s(https://doc.rust-lang.org/stable/std/vec/struct.Vec.html)가 각 슬라이스로 자동 강제됩니다.

`str` 및 `&[T]`에 구현된 메서드는 `String` 및 `Vec<T>`에서 직접 액세스할 수 있습니다. 예를 들어 `trim()`은 `&str`의 메서드이고 일부_스트링이 `String` 인 경우에도 일부_스트링 `.trim()` 이 작동합니다 .

일반 코드와 같은 경우에는 수동으로 변환해야 하는 경우도 있습니다. 다음과 같이 슬라이싱 연산자를 사용하여 수동으로 변환할 수 있습니다: `&my_vec[..]` .

`str`에서 문자열로 또는 그 반대로 변환하려면 어떻게 해야 하나요? (#how-to-convert-between-str-and-string)

`to_string()` (https://doc.rust-lang.org/stable/std/string/trait.ToString.html#tymethod.to_string) 메서드는 `&str`(https://doc.rust-lang.org/stable/std/primitive.str.html)을 문자열(https://doc.rust-lang.org/stable/std/string/struct.String.html)로 변환하고, 문자열 s(https://doc.rust-lang.org/stable/std/string/struct.String.html)는 참조를 빌릴 때 자동으로 `&str`(https://doc.rust-lang.org/stable/std/primitive.str.html)로 변환합니다. 다음 예시에서는 두 가지를 모두 보여줍니다:

```
fn main() {
    let s = "Jane Doe".to_string();
    say_hello(&s);
}

fn say_hello(name: &str) {
    println!("안녕하세요 {}!", name);
}
```

두 가지 문자열 유형의 차이점은 무엇인가요? (#what-are-the-differences-between-str- and-string)

문자열(https://doc.rust-lang.org/stable/std/string/struct.String.html)은힙에할당된UTF-8 바이트의 소유 버퍼입니다. MutablTerySoturtitnhgesn(ehwttpsist:e/!/(dtprsu.s/t/-wlawnwg.orugst/-slaanbgleo/rsgt/d)/string/struct.String.html)을 수정하여 필요에 따라 용량을 늘릴 수 있습니다. `&str`(https://doc.rust- lang.org/stable/std/primitive.str.html)은 고정 용량 '보기'가 `String`(https://doc.rust- lang.org/stable/std/string/struct.String.html)의 다른 곳에 할당되며, 일반적으로 `String` s(https://doc.rust-lang.org/stable/std/string/struct.String.html)에서 역참조된 슬라이스의 경우 힙에, 문자열 리터럴의 경우 정적 메모리에 할당됩니다.

`&str`(https://doc.rust-lang.org/stable/std/primitive.str.html)은 Rust 언어에서 구현된 원시 유형이며, `String`(https://doc.rust-lang.org/stable/std/string/struct.String.html)은 표준 라이브러리에서 구현됩니다.

문자열에서 O(1) 문자 액세스를 수행하려면 어떻게 해야 하나요? (#how-do-i-do-o1-character-access-in-a-string)

불가능합니다. 적어도 "문자"가 의미하는 바를 확실히 이해하고 문자열을 전처리하여 원하는 문자의 인덱스를 찾지 않는 한은 불가능합니다.

Rust 문자열은 UTF-8로 인코딩됩니다. UTF-8의 단일 시각적 문자는 ASCII로 인코딩된 문자열에서와 같이 반드시 1바이트가 아닙니다. 각 바이트를 "코드 단위"라고 합니다(UTF-16에서 코드 단위는 2바이트, UTF-32에서는 4바이트). "코드 포인트"는 하나 이상의 코드 단위로 구성되며, 문자에 가장 근접한 "그래프 클러스터"로 결합됩니다.

따라서 UTF-8 문자열의 바이트에 대해 인덱싱할 수는 있지만, 상수 시간에 코드 포인트 또는 그래프 클러스터에 액세스할 수는 없습니다. 그러나 원하는 코드 포인트 또는 그래프 클러스터가 시작되는 바이트가 어느 바이트에서 시작되는지 알고 있다면 상수 시간에 액세스할 *수 있습니다*. `str::find()`(https://doc.rust-lang.org/stable/std/primitive.str.html#method.find) 및 정규식 일치 등의 함수는 바이트 인덱스를 반환하여 이러한 종류의 액세스를 용이하게 합니다.

문자열이 기본적으로 UTF-8인 이유는 무엇인가요? (#why-are-strings-utf-8)

`str`(https://doc.rust-lang.org/stable/std/primitive.str.html) 유형이 UTF-8인 이유는 이 인코딩에서 특히 엔디안에 구애받지 않는 네트워크 전송에서 더 많은 텍스트를 관찰할 수 있으며, I/O의 기본 처리 방식이 각 방향의 코드 포인트를 다시 코딩할 필요가 없는 것이 최선이라고 생각하기 때문입니다.

이는 문자열 내에서 특정 유니코드 코드 포인트를 찾는 것이 O(n) 연산이라는 것을 의미하지만, 시작 바이트 인덱스가 이미 알려진 경우 예상대로 O(1)로 액세스할 수 있습니다. 한편으로 이것은 분명히 바람직하지 않지만, 다른 한편으로 이 문제에는 장단점이 많기 때문에 몇 가지 중요한 전제 조건을 지적하고자 합니다:

`str`(https://doc.rust-lang.org/stable/std/primitive.str.html)에서 ASCII 범위 코드 포인트를 스캔하는 것은 여전히 한 번에 한 바이트씩 안전하게 수행할 수 있습니다. `.as_bytes()`(https://doc.rust-

lang.org/stable/std/primitive.str.html#method.as_bytes)를 사용하는 경우, `u8`(https://doc.rust-lang.org/stable/std/primitive.u8.html)을 가져오는 데 `O(1)`의 비용만 들며 캐스팅하여 ASCII 범위 문자(https://doc.rust-lang.org/stable/std/primitive.char.html)와 비교할 수 있는 값을 생성할 수 있습니다. 따라서 (예를 들어) `'\n'`에서줄 바꿈을 하는 경우 바이트 기반 처리가 여전히 작동합니다. UTF-8은 이런 식으로 잘 설계되었습니다.

텍스트에 대한 대부분의 '문자 지향' 연산은 'ASCII 범위 코드포인트만'과 같이 매우 제한된 언어 가정 하에서만 작동합니다. ASCII 범위를 벗어나면 언어 단위(글리프, 단어, 단락) 경계를 결정하기 위해 복잡한 (상수 시간이 아닌) 알고리즘을 사용해야 하는 경우가 많습니다. "정직한" 언어 인식, 유니코드 승인 알고리즘을 사용하는 것이 좋습니다.

`char`(https://doc.rust-lang.org/stable/std/primitive.char.html) 타입은 UTF-32입니다. 한 번에 한 코드 포인트씩 알고리즘을 수행해야 하는 경우, `wstr = [char]` 유형을 작성하고 한 번에 `str`(https://doc.rust-lang.org/stable/std/primitive.str.html)을 언패킹한 다음 `wstr` 로 작업하는 것이 간단합니다. In

즉, 언어가 "기본적으로 UTF32로 디코딩"되지 않는다는 사실 때문에 디코딩(또는재인코딩)을 중단해서는 안 됩니다.

일반적으로 UTF-16 또는 UTF-32보다 UTF-8이 선호되는 이유에 대한 자세한 설명은 UTF-8 에브리웨어 선언문(http://utf8everywhere.org/)을 참조하세요.

어떤 문자열 유형을 사용해야 하나요? (#어떤 문자열 유형을 사용해야 하나요?)

Rust에는 각각 다른 용도로 사용되는 네 쌍의 문자열 유형이 있습니다(http://www.suspectsemantics.com/blog/2016/03/27/string-types-in-rust/). 각 쌍에는 "소유" 문자열 유형과 "슬라이스" 문자열 유형이 있습니다. 조직은 다음과 같습니다:

	"Slice" type	"Owned" type
UTF-8	`str`	문자열
OS 호환	`OsStr`	`OsString`
C 호환	`CStr`	`CString`
시스템 경로	경로	`PathBuf`

Rust의 다양한 문자열 유형은 서로 다른 용도로 사용됩니다. String과 `str`은 UTF-8로 인코딩된 범용 문자열입니다. OsString과 OsStr은 현재 플랫폼에 따라 인코딩되며 운영 체제와 상호 작용할 때 사용됩니다. `CString` 및 `CStr`은 C에서 문자열에 해당하는 Rust로 FFI 코드에서 사용되며, `PathBuf` 및 `Path`는 경로 조작에 특화된 메서드를 제공하는 `OsString` 및 `OsStr`을 둘러싼 편의 래퍼입니다.

스트링과 문자열을 모두 허용하는 함수를 작성하려면 어떻게 해야 하나요? (#왜-여러-종류의-스트링이-있나요?)

기능의 필요에 따라 몇 가지 옵션이 있습니다:

- 함수에 소유 문자열이 필요하지만 모든 유형의 문자열을 허용하려는 경우, `Into<String>` 바운드.
- 함수에 문자열 슬라이스가 필요하지만 모든 유형의 문자열을 허용하려는 경우 `AsRef<str>` 바인딩을 사용합니다. • 함수가 문자열 유형을 신경 쓰지 않고 두 가지 가능성을 균일하게 처리하려는 경우, 입력 유형으로 Cow<str>을 사용합니다.

Using `In<String>`으로

이 예제에서 함수는 소유 문자열과 문자열 조각을 모두 허용하며, 아무것도 하지 않거나 함수 본문 내에서 입력을 소유 문자열로 변환합니다. 변환은 명시적으로 수행해야 하며 그렇지 않으면 수행되지 않습니다.

```
fn accepts_both<S: Into<String>>(s: S) {
    let s = s.into();    // 이것은 s를 '문자열'로 변환합니다.
    // ... 나머지 함수
}
```

Using `AsRef<str>`

이 예제에서 함수는 소유 문자열과 문자열 조각을 모두 허용하며, 아무 작업도 수행하지 않거나 입력을 문자열 조각으로 변환합니다. 이 작업은 다음과 같이 참조로 입력을 받아 자동으로 수행할 수 있습니다:

```
fn accepts_both<S: AsRef<str> + Iterator<Item=&String>>(s: S) {
    // ... 함수 본문
}
```

Using `Cow<str>`

이 예제에서 함수는 일반 유형이 아니라 필요에 따라 소유된 문자열 또는 문자열 조각을 포함하는 컨테이너인 `Cow<str>` 을 받습니다.

```
fn accepts_cow(s: Cow<str>) {
    // ... 함수 본문
}
```

컬렉션

벡터나 링크된 리스트와 같은 데이터 구조를 Rust에서 효율적으로 구현할 수 있나요? (#can-i-implement-linked-lists-in-rust)

이러한 데이터 구조를 구현하는 이유가 다른 프로그램에 사용하기 위한 것이라면 표준 라이브러리에서 이러한 데이터 구조의 효율적인 구현이 제공되므로 구현할 필요가 없습니다.

그러나 단순히 배우기 위한 목적(http://cglab.ca/~abeinges/blah/too-many-lists/book/)이라면 안전하지 않은 코드를 사용해야 할 가능성이 높습니다. 이러한 데이터 구조는 안전한 Rust로 완전히 구현할 수 있지만, 안전하지 않은 코드를 사용할 때보다 성능이 저하될 가능성이 높습니다. 그 간단한 이유는 벡터 및 링크된 리스트와 같은 데이터 구조가 안전한 Rust에서는 허용되지 않는 포인터 및 메모리 연산에 의존하기 때문입니다.

예를 들어 이중 링크된 목록은 각 노드에 대해 두 개의 변경 가능한 참조가 있어야 하지만 이는 Rust의 변경 가능한 참조 앨리어싱 규칙을 위반합니다. `Weak<T>`(https://doc.rust-lang.org/stable/std/rc/struct.Weak.html)를 사용하여 이 문제를 해결할 수 있지만 성능이 원하는 것보다 떨어질 수 있습니다. 안전하지 않은 코드를 사용하면 변경 가능한 참조 앨리어싱 규칙 제한을 우회할 수 있지만 코드가 메모리 안전 위반을 일으키지 않는지 수동으로 확인해야 합니다.

컬렉션을 이동/소모하지 않고 반복하려면 어떻게 해야 하나요? (#어떻게-모음을-소비하지-않고-모음에-대해-반복할-수-있나요?)

가장 쉬운 방법은 컬렉션의 `IntoIterator`(https://doc.rust-lang.org/stable/std/iter/trait.IntoIterator.html) 구현을 사용하는 것입니다. 다음은

`&Vec`(https://doc.rust-lang.org/stable/std/vec/struct.Vec.html)에 대한 예제입니다:

```
let v = vec![1,2,3,4,5];
for item in &v {
    print!("{} ", item);
}
println!("\nLength: {}", v.len());
```

루프에 대한 녹은 `IntoIterator`(https://doc.rust-lang.org/stable/std/iter/trait.IntoIterator.html)에 정의된 `into_iter()`를 호출합니다 (https://doc.rust-lang.org/stable/std/iter/trait.IntoIterator.html)ting over. `IntoIterator`(https://doc.rust-lang.org/stable/std/iter/trait.IntoIterator.html) 특성을 구현하는 모든것은 `for` 루프를 사용하여 반복될 수 있습니다. `IntoIterator`(https://doc.rust-lang.org/stable/std/iter/trait.IntoIterator.html)는 `&Vec`(https://doc.rust-lang.org/stable/std/vec/struct.Vec.html) 및 `&mut Vec`(https://doc.rust-lang.org/stable/std/vec/struct.Vec.html)에 대해 구현되어 있어 `into_iter()`의 이터레이터가 컬렉션의 내용을 이동/소비하지 않고 빌려오도록 합니다. 다른 표준 컬렉션도 마찬가지입니다.

이동/소비하는 반복자가 필요한 경우 반복에 `&` 또는 `&mut` 없이 `for` 루프를 작성합니다.

차용 이터레이터에 직접 액세스해야 하는 경우 일반적으로 `iter()` 메서드를 호출하여 이터레이터를 가져올 수 있습니다.

배열 선언에 배열 크기를 입력해야 하는 이유는 무엇인가요? (#왜 배열 선언에 배열 크기를 입력해야 하나요?)

반드시 그럴 필요는 없습니다. 배열을 직접 선언하는 경우 요소 수에 따라 크기가 유추됩니다. 하지만 고정 크기 배열을 취하는 함수를 선언하는 경우 컴파일러는 해당 배열의 크기를 알아야 합니다.

한 가지 주의할 점은 현재 Rust는 크기가 다른 배열에 대한 제네릭을 제공하지 않는다는 것입니다. 가변적인 수의 값으로 구성된 연속적인 컨테이너를 받으려면 소유권이 필요한지 여부에 따라 `Vec`(https://doc.rust-lang.org/stable/std/vec/struct.Vec.html) 또는 슬라이스를 사용하세요.

소유권

주기가 포함된 그래프 또는 기타 데이터 구조를 구현하려면 어떻게 해야 하나요? (#how-can-i-implement-a-data- structure-that-contains-cycles)

최소 네 가지 옵션이 있습니다(너무 많은 링크된 목록 (http://cglab.ca/~abeinges/blah/too-many-lists/book/)에 자세히 설명되어 있습니다):

- `Rc`(https://doc.rust-lang.org/stable/std/rc/struct.Rc.html) 및 약 (https://doc.rust-lang.org/stable/std/rc/struct.Weak.html)을 사용하여 노드의 공유 소유권을 허용하는 방식으로 구현할 수 있지만, 이 방식은 메모리 관리 비용을 지불해야 합니다.

- 원시 포인터를 사용하는 안전하지 않은 코드를 사용하여 구현할 수 있습니다. 이 방법은 더 효율적이지만 Rust의 안전 보장을 우회합니다.
- 벡터와 인덱스를 해당 벡터에 사용합니다. 이 접근 방식에 대한 몇 가지 (http://smallcultfollowing.com/babysteps/blog/2015/04/06/modeling-graphs-in-rust-using-vector- indices/) 예제 및 설명(https://featherweightmusings.blogspot.com/2015/04/graphs-in-rust.html)이 있습니다.
- `UnsafeCell`에서 차용한 참조 사용(https://doc.rust-lang.org/stable/std/cell/struct.UnsafeCell.html). 이 접근 방식에 대한 설명과 코드 (https://github.com/nrc/r4cppp/blob/master/graphs/README.md#node-and-unsafecell)가 있습니다.

자체 필드 중 하나에 대한 참조가 포함된 구조체를 어떻게 정의할 수 있나요? (#how-can-i-define-a-structure- that-contains-a-reference-to-its-own-field- 중 하나에 대한 참조를 포함하는 구조체를 정의하려면 어떻게 해야 하나요?)

가능하지만 그렇게 하는 것은 쓸모가 없습니다. 구조체 자체가 영구적으로 빌려지므로 이동할 수 없습니다. 다음은 몇 가지 코드입니다. (https://www.rust-lang.org/)

```
std::cell::Cell을 사용합니다;

#[파생(디버그)] 구조체
Unmovable<'a> {
    X: U32,
    y: 셀<옵션<&'a u32>>,
}
fn main() {
    let test = 움직일 수 없음 { x: 42, y: Cell::new(None)
    }; test.y.set(Some(&test.x));

    println!("{:?}", test);
}
```

가치 전달, 소비, 이동, 소유권 이전의 차이점은 무엇인가요? (#what-is-the-difference-between-consuming-and-moving)

이는 같은 의미의 다른 용어입니다. 모든 경우에 값이 다른 소유자에게 이동되어 더 이상 사용할 수 없는 원래 소유자의 소유에서 벗어났음을 의미합니다. 유형이 복사 특성을 구현하는 경우 원래 소유자의 값은 무효화되지 않고 계속 사용할 수 있습니다.

어떤 유형의 값은 함수에 전달한 후 사용할 수 있지만 다른 유형의 값은 재사용하면 오류가 발생하는 이유는 무엇인가요? (#왜 일부 유형의 값을 재사용하면 다른 유형이 소비되는가)

타입이 `Copy`(https://doc.rust-lang.org/stable/std/ptr/fn.copy.html) 특성을 구현하면 함수에 전달될 때 복사됩니다. Rust의 모든 숫자 타입은 `Copy`(https://doc.rust- lang.org/stable/std/ptr/fn.copy.html)를 구현하지만, 구조체 타입은 기본적으로 `Copy`(https://doc.rust- lang.org/stable/std/ptr/fn.copy.html)를 구현하지 않으므로 대신 이동됩니다. 즉, 반환을 통해 함수 밖으로 다시 이동하지 않는 한 구조체를 더 이상 다른 곳에서 사용할 수 없습니다.

'이동된 값 사용' 오류는 어떻게 처리하나요? (#이동된 값 사용 오류를 처리하는 방법)

이 오류는 사용하려는 값이 새 소유자로 이동되었음을 의미합니다. 가장 먼저 확인해야 할 것은 문제의 이동이 필요한 것인지 여부입니다. 함수로 이동한 경우 이동 대신 참조를 사용하도록 함수를 다시 작성하는 것이 가능할 수 있습니다. 그렇지 않으면 이동하려는 유형이 `Clone`(https://doc.rust-lang.org/stable/std/clone/trait.Clone.html#tymethod.clone)을 구현하는 경우 이동하기 전에 해당 유형에 대해

`clone()`을 호출하면 복사본이 이동되고 원본은 계속 사용할 수 있습니다. 단, 값 복제는 비용이 많이 들고 추가 할당을 유발할 수 있으므로 일반적으로 값 복제는 최후의 수단으로 사용해야 한다는 점에 유의하세요.

이동된 값이 사용자 정의 유형인 경우 이동이 아닌 암시적 복사를 위해 `Copy`(https://doc.rust-lang.org/stable/std/ptr/fn.copy.html) 또는 명시적 복사를 위해 `Clone`(https://doc.rust-lang.org/stable/std/clone/trait.Clone.html#tymethod.clone)을 구현하는 것을 고려하세요.

`Copy`(https://doc.rust- lang.org/stable/std/ptr/fn.copy.html)는 `#[derive(Copy, Clone)]`로 구현되는 것이 가장 일반적입니다.

복사 (https://doc.rust-lang.org/stable/std/ptr/fn.copy.html) 에는 `#[derive(Clone)]`를 사용하여 `Clone`(https://doc.rust- lang.org/stable/std/clone/traTirtyColountethetmnle#wtysmiteeth(chdtplo)wavn rdusCtl-loanneg.(hgt)s://doc.rust- lang.org/stable/std/clone/trait.Clone.html#tymethod.clone) 이 필요합니다 .

이 중 어느 것도 가능하지 않은 경우, 소유권을 획득한 함수를 수정하여 함수가 종료될 때 값의 소유권을 반환하도록 할 수 있습니다.

메서드 선언에서 `self`, `&self` 또는 `&mut self`를 사용하는 규칙은 무엇인가요? (#what-are-the-rules-for-different-self-types-in-methods)

- 함수가 값을 소비해야 하는 경우 자체 사용
- 함수에 값에 대한 읽기 전용 참조만 필요한 경우 `&self`를 사용합니다.
- 함수가 값을 소비하지 않고 값을 변경해야 하는 경우 `&mut self`를 사용합니다.

차용 검사기를 어떻게 이해해야 하나요? (#how-can-i-understand-the-borrow-checker)

빌리기 검사기는 Rust 코드를 평가할 때 Rust 책(https://doc.rust-lang.org/stable/book/second-edition/ch15-02-deref.html)의 빌리기 섹션에서 찾을 수 있는 몇 가지 규칙만 적용합니다. 이러한 규칙은 다음과 같습니다:

> 첫째, 모든 차입은 소유자보다 더 큰 범위에서 지속되지 않아야 합니다. 둘째, 이 두 가지 유형의 차입 중 하나 또는 다른 유형을 보유할 수 있지만 동시에 두 가지 유형을 모두 보유할 수는 없습니다:
>
> - 리소스에 대한 하나 이상의 참조(&T)• 정확히
>
> 하나의 변경 가능한 참조(&mut T)

규칙 자체는 간단하지만, 특히 수명 및 소유권에 대한 추론에 익숙하지 않은 사람들에게는 규칙을 일관되게 따르는 것이 쉽지 않습니다.

대여 검사기를 이해하기 위한 첫 번째 단계는 검사기에서 발생하는 오류를 읽는 것입니다. 차용 검사기가 식별한 문제를 해결하는 데 양질의 지원을 제공하기 위해 많은 노력을 기울였습니다.
차용 검사기 문제가 발생하면 첫 번째 단계는 보고된 오류를 천천히 주의 깊게 읽고 설명된 오류를 이해한 후에만 코드에 접근하는 것입니다.

두 번째 단계는 `Cell`(https://doc.rust-lang.org/stable/std/cell/struct.Cell.html), `RefCell`(https://doc.rust-lang.org/stable/std/cell/struct.RefCell.html), `Cow`(https://doc.rust-lang.org/stable/std/borrow/enum.Cow.html) 등 Rust 표준 라이브러리에서 제공하는 소유권 및 변경 가능성 관련 컨테이너 유형에 익숙해지는 것입니다. 이들은 특정 소유권 및 가변성 상황을 표현하는 데 유용하고 필요한 도구이며, 성능 비용이 최소화되도록 작성되었습니다.

차용 검사기를 이해하는 데 있어 가장 중요한 부분은 연습입니다. Rust의 강력한 정적 분석 보장은 엄격하며 이

전에 많은 프로그래머가 작업했던 것과는 상당히 다릅니다. 모든 것에 완전히 익숙해지려면 시간이 좀 걸릴 것입니다.

차용 검사기를 사용하는 데 어려움을 겪고 있거나 인내심이 부족하다면 언제든지 Rust 커뮤니티 (community.html)에 도움을 요청하세요.

`Rc`는 언제 유용하나요? (#when-is-rc-useful)

이는 Rust의 비원자 참조 카운트 포인터 유형인 `Rc`(https://doc.rust- lang.org/stable/std/rc/struct.Rc.html)에 대한 공식 문서에서 다루고 있습니다. 간단히 말해, `Rc`(https://doc.rust-lang.org/stable/std/rc/struct.Rc.html)와 스레드 안전 사촌인 `Arc`(https://doc.rust-lang.org/stable/std/sync/struct.Arc.html)는 공유 소유권을 표현하고 아무도 액세스 권한이 없을 때 시스템에서 관련 메모리를 자동으로 할당 해제하도록 하는 데 유용합니다.

함수에서 클로저를 반환하려면 어떻게 해야 하나요? (#함수에서 클로저를 반환하는 방법)

새 사이트를 사용해 보세요! (https://www.rust-lang.org/)

함수에서 클로저를 반환하려면 "이동 클로저"여야 합니다. 즉, 클로저가 `move` 키워드로 선언되어야 합니다. Rust 책(https://doc.rust-lang.org/book/closures.html#move- closures)에 설명된 대로, 이렇게 하면 부모 스택 프레임과 무관하게 클로저에 캡처된 변수의 자체 복사본이 제공됩니다. 그렇지 않으면 클로저를 반환하면 더 이상 유효하지 않은 변수에 액세스할 수 있기 때문에 안전하지 않습니다. 다시 말해, 잠재적으로 유효하지 않은 메모리를 읽을 수 있게 됩니다. 또한 클로저는 `Box`(https://doc.rust-lang.org/stable/std/boxed/struct.Box.html)로 감싸서 힙에 할당되어야 합니다. 이에 대한 자세한 내용은 책(https://doc.rust-lang.org/book/closures.html#returning- closures)을 참조하세요.

디레프 강제는 무엇이며 어떻게 작동하나요? (#무엇이-디레프-강요인가요?)

디레프 강제(https://doc.rust-lang.org/book/deref-coercions.html)는 포인터(예: `&Rc<T>` 또는 `&Box<T>`)에 대한 참조를 해당 내용에 대한 참조(예: `&T`)로 자동 변환하는 편리한 강제입니다. 디레프 강제는 Rust를 보다 인체공학적으로 사용하기 위해 존재하며, 디레프(https://doc.rust-lang.org/stable/std/ops/trait.Deref.html) 특성을 통해 구현됩니다.

디레프 구현은 호출 유형에 대한 불변 참조를 취하고 대상 유형에 대한 참조(동일한 수명의 참조)를 반환하는 디레프 메서드 호출에 의해 구현 유형이 대상으로 변환될 수 있음을 나타냅니다. 접두사 연산자 `*`는 `deref` 메서드의 축약어입니다.

이를 '강압'이라고 부르는 이유는 다음과 같은 규칙 때문이며, 이는 Rust 책에서 인용한 것입니다 (https://doc.rust- lang.org/stable/book/second-edition/ch15-02-deref.html):

> `U` 유형이 있고 `Deref<Target=T>` 를 구현하는 경우, `&U` 값은 자동적으로 `&T` .

예를 들어, &Rc<String>이 있는 경우 이 규칙을 통해 `&String`으로 강제로 변환한 다음 같은 방식으로 `&str`로 강제로 변환합니다. 따라서 함수가 `&str` 매개변수를 받는 경우, &Rc<String>을 직접 전달할 수 있으며, 모든 강제는 `Deref` 특성을 통해 자동으로 처리됩니다.

가장 일반적인 종류의 디레프 강압은 다음과 같습니다:

- &Rc<T>에서 `&T`
- &Box<T>에서 `&T`
- &Arc<T>에서 `&T`
- `&Vec<T>` to `&[T]`
- &String to `&str`

라이프타임

왜 평생인가요? (#why-lifetimes)

라이프타임은 메모리 안전성에 대한 Rust의 해답입니다. 이를 통해 Rust는 가비지 컬렉션의 성능 비용 없이 메모리 안전성을 보장할 수 있습니다. 이는 다양한 학술 연구를 기반으로 하며, Rust 책(https://doc.rust-lang.org/stable/book/first-edition/bibliography.html#type- 시스템)에서 확인할 수 있습니다.

평생 구문이 왜 이런 식으로 되어 있나요? (#왜-평생-구문이-이런-방식인가)

'a 구문은 ML 프로그래밍 언어 제품군에서 유래한 것으로, 'a는 일반 유형매개변수를나타내는 데 사용됩니다. Rust의경우,이구문이정확히ehthtpins:g//thwawt ww.arussut-nlamngb.org가눈에띄고,특성 및 참조와 함께 유형 선언에 잘어울립니다. 대체 구문에 대한 논의가 있었지만 명확하게 더 나은 것으로 입증된 대체 구문은 없습니다.

함수에서 빌린 것을 내가 만든 것에 반환하려면 어떻게 해야 하나요? (#how-do-I-return-a-borrow-to-something-created-from-a-function)

빌린 항목이 함수보다 오래 지속되도록 해야 합니다. 출력 수명을 입력 수명에 다음과 같이 바인딩하면 됩니다:

```
타입 풀 = 타입 아레나<것>;

// (아래의 수명은 명시적으로
// 설명 목적; 생략할 수 있습니다.
// 이후 FAQ 항목에 설명된 삭제 규칙) fn
create_borrowed<'a>(pool: &'a Pool,
                        x: i32,
                        y: i32) -> &'a Thing {
    pool.alloc(Thing { x: x, y: y })
}
```

대안은 String과 같은 소유 유형을 반환하여 참조를 완전히 제거하는 것입니다. (https://doc.rust-lang.org/stable/std/string/struct.String.html):

```
fn happy_birthday(name: &str, age: i64) -> String {
    format!("안녕하세요 {}! 당신은 {}살입니다!", name, age)
}
```

이 방법은 더 간단하지만 불필요한 할당을 초래하는 경우가 많습니다.

왜 어떤 레퍼런스에는 &'a T 와같은 수명이 있고 어떤 레퍼런스에는 &T 와 같은 수명이 없는가요? (#언제-라이프타임이-명확해야-하는가)

실제로 *모든* 참조 유형에는 수명이 있지만 대부분의 경우 명시적으로 작성할 필요는 없습니다. 규칙은 다음과 같습니다:

1. 함수 본문 내에서 수명을 명시적으로 작성할 필요 없이 항상 올바른 값을 유추할 수 있어야 합니다.
2. 함수 *서명* 내에서(예를 들어 인자 유형 또는 반환 유형) 수명을 명시적으로 작성해야 할 *수도* 있습니다. 수명은 다음 세 가지 규칙으로 구성된 "수명 생략"(https://doc.rust-lang.org/book/lifetimes.html#lifetime-elision)이라는 간단한 기본값 체계를 사용합니다:
    - 함수의 인자에서 제거된 각 수명은 별개의 수명 매개변수가 됩니다.

- 엘리드 여부와 관계없이 입력 수명이 정확히 하나만 있는 경우 해당 수명은 해당 함수의 반환 값에서 엘리드된 모든 수 명 에 할당됩니다.
- 입력 수명이 여러 개 있지만 그 중 하나가 &self 또는 &mut self인 경우, 제거된 모든 출력 수명에 자체의 수명이 할당됩니다.

3. 마지막으로 구조체 또는 열거형 정의에서 모든 수명은 명시적으로 선언되어야 합니다.

새 사이트를 사용해 보세요! (https://www.rust-lang.org/)

이러한 규칙으로 인해 컴파일 에러가 발생하면 Rust 컴파일러는 오류가 발생했으며, 추론 프로세스의 어느 단계에서 오류가 발생했는지에 따라 잠재적인 해결 방법을 제안합니다.

Rust는 어떻게 "널 포인터 없음"과 "댕글링 포인터 없음"을 보장할 수 있을까요? (#how-can-rust-guarantee-null-포인터 없음)

Foo 또는 `&mut Foo` 타입의 값을 구성하는 유일한 방법은 참조가 가리키는 `Foo` 타입의 기존 값을 지정하는 것입니다. 참조는 지정된 코드 영역(참조의 수명)에 대한 원래 값을 "차용"하며, 차용되는 값은 차용 기간 동안 이동하거나 삭제할 수 없습니다.

널이 없는 값의 부재를 어떻게 표현하나요?(#how-do-i-express-the-absense-of-a-value- without-null)

옵션(https://doc.rust-lang.org/stable/std/option/enum.Option.html) 유형은 일부 `(T)` 또는 없음일 수 있습니다. 일부 `(T)` 는 `T` 유형의 값이 포함되어 있음을 나타내며, 없음은 값이 없음을 나타냅니다.

제네릭

"단일화"란 무엇인가요? (#왓-이즈-모노모피제이션)

단형화는 해당 함수(또는 구조체)에 대한 호출의 매개변수 유형(또는 구조체의 용도)에 따라 일반 함수(또는 구조체)의 각 용도를 특정 인스턴스로 특화합니다.

단일화 과정에서 함수가 인스턴스화되는 각 고유한 유형 집합에 대해 제네릭 함수의 새 복사본이 변환됩니다. 이는 C++에서 사용하는 것과 동일한 전략입니다. 이 전략은 모든 호출 사이트에 특화되고 정적으로 파견되는 빠른 코드를 생성하지만, 다양한 유형으로 인스턴스화된 함수는 여러 함수 인스턴스로 인해 다른 번역 전략으로 생성할 때보다 더 큰 바이너리가 생성되는 '코드 부풀림'을 유발할 수 있다는 단점이 있습니다.

유형 매개변수 대신 특성 객체(https://doc.rust-lang.org/book/trait-objects.html)를 받아들이는 함수는 단일화 과정을 거치지 않습니다. 대신 특성 객체에 대한 메서드가 런타임에 동적으로 파견됩니다.

함수와 변수를 캡처하지 않는 클로저의 차이점은 무엇인가요? (#what-the- difference-between-a-function-and-a-closure-that-does-not-capture)

함수와 클로저는 작동 방식은 동일하지만 구현 방식이 다르기 때문에 런타임 표현이 다릅니다.

함수는 언어에 내장된 프리미티브인 반면, 클로저는 본질적으로 세 가지 특성 중 하나에 대한 구문 설탕입니다:

후 `Fn`(https://doc.rust-lang.org/stable/std/ops/trait.Fn.html), `FnMut`(https://doc.rust-lang.org/stable/std/ops/trait.FnMut.html) 및 `FnOnce`(https://doc.rust-lang.org/stable/std/ops/trait.FnOnce.html). 클로저를 만들면 Rust 컴파일러는 이 세 가지 중 적절한 특성을 구현하고 캡처된 환경 변수를 멤버로 포함하는 구조체를 자동으로 생성하고 이 구조체를 함수로 호출할 수 있도록 만듭니다. 베어 함수는 환경을 캡 처 할 수 없습니다.

이러한 특성의 가장 큰 차이점은 `self` 매개 변수를 취하는 방식입니다. `Fn` (https://doc.rust-lang.org/stable/std/ops/trait.Fn.html)은 `&self` 를 취하고, `FnMut` (https://doc.rust-lang.org/stable/std/ops/trait.FnMut.html)는 `&mut self` 를취하고, `FnOnce` (https://doc.rust- lang.org/stable/std/ops/trait.FnOnce.html)는 `self` 를 취합니다.

클로저가 환경 변수를 캡처하지 않더라도 런타임에는 다른 클로저와 동일하게 두 개의 포인터로 표현됩니다.

상위 유형이란 무엇이며, 왜 내가 이런 유형을 원하고, 왜 Rust에는 이런 유형이 없나요? (#높은 종류란 무엇인가요?)

상위 형은 매개변수가 채워지지 않은 형입니다. `Vec`(https://doc.rust-lang.org/stable/std/vec/struct.Vec.html), `Result` (https://doc.rust-lang.org/stable/std/result/enum.Result.html), `HashMap`(https://doc.rust-lang.org/stable/std/collections/struct.HashMap.html) 같은 타입 생성자는 모두 상위 타입의 예이며, 각 타입은 실제로 `Vec<u32>` 같은 특정 타입을 나타내기 위해 몇 가지 추가 타입 매개 변수를 필요로 합니다. 상위 유형에 대한 지원은 이러한 "불완전한" 유형을 함수의 제네릭을 포함하여 "완전한" 유형을 사용할 수 있는 모든 곳에서 사용할 수 있음을 의미합니다.

`i32`(https://doc.rust-lang.org/stable/std/primitive.i32.html), `bool`(https://doc.rust-lang.org/stable/std/primitive.bool.html), `char`(https://doc.rust- lang.org/stable/std/primitive.char.html)와 같은 모든 완전한 유형은 종류 `*`입니다(이 표기법은 유형 이론 분야에서 유래한 것입니다).

`Vec<T>`(https://doc.rust-lang.org/stable/std/vec/struct.Vec.html)와 같이 매개변수가 하나인 타입은 동종입니다.

`* -> *`,즉, `Vec<T>`(https://doc.rust-lang.org/stable/std/vec/struct.Vec.html)는 `i32`(https://doc.rust-lang.org/stable/std/primitive.i32.html)와 같은 완전한 타입을 받아 완전한 타입 `Vec<i32>` 를 반환합니다. `HashMap<K, V, S>` (https://doc.rust- lang.org/stable/std/collections/struct.HashMap.html)처럼 세 개의 매개 변수가 있는 형은 `* -> * -> *` 종류이며, 세 개의 완전한 형(예: `i32` (https://doc.rust-lang.org/stable/std/primitive.i32.html), `String`(https://doc.rust-lang.org/stable/std/string/struct.String.html), `RandomState`(https://doc.rust-lang.org/stable/std/collections/hash_map/struct.RandomState.html))를 받아 새로운 완전한 형 `HashMap<i32, String, RandomState>` 를 생성합니다.

이러한 예제 외에도 형 생성자는 *평생*인자를 취할 수 있으며, 이 를 `Lt` 로 표시하겠습니다. 예를 들어, 슬라이스 `::Iter`는 `Iter<'a, u32>`처럼 인스턴스화되어야 하므로 `Lt -> * -> *` 종류를 갖습니다.

상위 유형에 대한 지원이 부족하기 때문에 특정 종류의 일반 코드를 작성하기가 어렵습니다. 특히 이터레이터와 같은 개념을 추상화할 때 문제가 되는데, 이터레이터는 적어도 수명에 걸쳐 매개변수화되는 경우가 많기 때문입니다. 이는 결국 Rust의 컬렉션을 추상화하는 특성을 만들지 못하게 만들었습니다.

또 다른 일반적인 예로는 단일 유형이 아닌 유형 생성자인 함수 또는 모나드와 같은 개념이 있습니다.

현재 Rust는 다른 개선 사항에 비해 우선순위가 낮았기 때문에 상위 유형에 대한 지원은 제공하지 않습니다. 디자인 전반에 걸친 주요 변경 사항이기 때문에 신중하게 접근하고자 합니다. 하지만 현재 지원되지 않는 본질적인 이유는 없습니다.

일반 유형에서 `<T=Foo>`와 같은 명명된 유형 매개변수는 무엇을 의미하나요? (#무엇-이름이 지정된 유형 매개변수는-일반 유형에서-어떤-의미를-가집니다)

이를 연관 유형(https://doc.rust-lang.org/stable/book/second-edition/ch19-03- advanced-traits.html)이라고 하며, `where` 절로 표현할 수 없는 특성 바운드를 표현할 수 있습니다. 예를 들어, 일반 바운드 `X: Bar<T=Foo>`는 "`X`는 특성 `Bar`를 구현해야 하며, `Bar`의 구현에서 `X`는 `Bar`의 연관 유형인 `T`에 대해 `Foo`를 선택해야 합니다"를 의미합니다. 이러한 제약 조건을 `where` 절을 통해 표현할 수 없는 예로는 `Box<Bar<T=Foo>>` 와 같은 특성 객체가 있습니다.

제네릭은 종종 유형 계열을 포함하며, 한 유형이 계열의다른모든 유형을 결정하기 때문에 연관된 유형이 존재합니다. 예를들어, 그래프자체와노드 및 에지에 대한 연관 유형이 있습니다. 각 그래프 유형은 연관된 유형을 고유하게 결정합니다. 연관 유형을 사용하면 이러한 유형 계열로 훨씬 간결하게 작업할 수 있으며, 많은 경우에 더 나은 유형 추론을 제공할 수 있습니다.

운영자에게 과부하가 걸릴 수 있나요? 어떤 오퍼레이터에 어떻게 과부하가 걸리나요? (#어떻게-오버로드-운영자)

관련 특성을 사용하여 다양한 연산자에 대한 사용자 정의 구현을 제공할 수 있습니다: 의 경우 (https://doc.rust-lang.org/stable/std/ops/trait.Add.html), `*`의 경우 `Mul`(https://doc.rust-lang.org/stable/std/ops/trait.Mul.html) 등을 추가하세요. 다음과 같이 보입니다:

```
사용 std::ops::추가; 구조


체 Foo;


임포트 Foo에 대한 추가 {
    유형 출력 = Foo;
    fn add(self, rhs: Foo) -> Self::Output {
        println!("추가 중!");
        self
    }
}
```

다음 연산자는 과부하가 걸릴 수 있습니다:

Operation	Trait
`+`	추가 (https://doc.rust-lang.org/stable/std/ops/trait.Add.html)
`+=`	애드어사인 (https://doc.rust-lang.org/stable/std/ops/trait.AddAssign.html)
바이너리 -	서브 (https://doc.rust-lang.org/stable/std/ops/trait.Sub.html)
`-=`	서브어시스트 (https://doc.rust-lang.org/stable/std/ops/trait.SubAssign.html)
`*`	물 (https://doc.rust-lang.org/stable/std/ops/trait.Mul.html)
`*=`	멀어사인(https://doc.rust-lang.org/stable/std/ops/trait.MulAssign.html)
`/`	`Div` (https://doc.rust-lang.org/stable/std/ops/trait.Div.html)
`/=`	`DivAssign` (https://doc.rust-lang.org/stable/std/ops/trait.DivAssign.html)
유니타리 -	부정 (https://doc.rust-lang.org/stable/std/ops/trait.Neg.html)
`%`	`Rem` (https://doc.rust-lang.org/stable/std/ops/trait.Rem.html)
`%=`	`RemAssign` (https://doc.rust-lang.org/stable/std/ops/trait.RemAssign.html)

`&`	비트앤 (https://doc.rust-lang.org/stable/std/ops/trait.BitAnd.html)
`|`	비트오르 (https://doc.rust-lang.org/stable/std/ops/trait.BitOr.html)
`| =`	비트오어어사인 (https://doc.rust-lang.org/stable/std/ops/trait.BitOrAssign.html)
`^`	비트소르(https://doc.rust-lang.org/stable/std/ops/trait.BitXor.html)
`^=`	비트엑스오어사인(https://doc.rust-lang.org/stable/std/ops/trait.BitXorAssign.html)
`!`	`Not` (https://doc.rust-lang.org/stable/std/ops/trait.Not.html)
`<<`	`Shl` (https://doc.rust-lang.org/stable/std/ops/trait.Shl.html)
`<<=`	`ShlAssign` (https://doc.rust-lang.org/stable/std/ops/trait.ShlAssign.html)
`>>`	`Shr` (https://doc.rust-lang.org/stable/std/ops/trait.Shr.html)

Operation	Trait
>>=	ShrAssign (https://doc.rust-lang.org/stable/std/ops/trait.ShrAssign.html)
*	데레프(https://doc.rust-lang.org/stable/std/ops/trait.Deref.html)
mut *	데레프멋 (https://doc.rust-lang.org/stable/std/ops/trait.DerefMut.html)
[]	색인 (https://doc.rust-lang.org/stable/std/ops/trait.Index.html)
mut []	IndexMut (https://doc.rust-lang.org/stable/std/ops/trait.IndexMut.html)

Eq/PartialEq와 Ord/PartialOrd를 분할하는 이유는 무엇인가요? (#why-the-split-between-eq-partialeq- and-ord-partialord)

Rust에는 값이 부분적으로만 정렬되거나 부분적으로만 동등한 일부 유형이 있습니다. 부분 정렬이란 주어진 유형의 값 중 서로보다 작지도 크지도 않은 값이 있을 수 있음을 의미합니다. 부분 동일성은 주어진 유형의 값 중 자신과 같지 않은 값이 있을 수 있음을 의미합니다.

부동소수점 유형(f32(https://doc.rust-lang.org/stable/std/primitive.f32.html) 및 f64(https://doc.rust-lang.org/stable/std/primitive.f64.html))이 각각 좋은 예입니다. 모든 부동소수점 유형은 NaN("숫자가 아님"을 의미) 값을 가질 수 있습니다. NaN은 그 자체와 같지 않으며(NaN == NaN은 거짓), 다른 부동 소수점 값보다 작거나 크지 않습니다. 따라서 f32(https://doc.rust- lang.org/stable/std/primitive.f32.html)와 f64(https://doc.rust-lang.org/stable/std/primitive.f64.html)는 모두 PartialOrd(https://doc.rust-lang.org/stable/std/cmp/trait.PartialOrd.html)와 PartialEq(https://doc.rust-lang.org/stable/std/cmp/trait.PartialEq.html)를 구현하지만 Ord(https://doc.rust-lang.org/stable/std/cmp/trait.Ord.html)는 구현하지 않으며 Eq(https://doc.rust-lang.org/stable/std/cmp/trait.Eq.html)는 구현하지 않습니다.

앞선 부등식에 대한 질문(#why-cant-i-compare-floats)에서 설명한 것처럼, 일부 컬렉션은 올바른 결과를 제공하기 위해 전체 순서/평등에 의존하기 때문에 이러한 구분이 중요합니다.

입력 / 출력

파일을 문자열로 읽으려면 어떻게 하나요? (#how-do-i-read-a-file-into-a-string)

std::io의 읽기(https://doc.rust-lang.org/stable/std/io/trait.Read.html) 특성에 정의된 read_to_string()(https://doc.rust-lang.org/stable/std/io/trait.Read.html#method.read_to_string) 메서드 사용(https://doc.rust-lang.org/stable/std/io/index.html)을 참조하세요.

```
std::io::Read를 사용합니다;
std::fs::File을 사용합니다;
fn read_file(path: &str) -> Result<String, std::io::Error> {
    let mut s = String::new();
    let _ = File::open(path)?.read_to_string(&mut s); // `s`는 다음의 내용을 포함합니다.
    확인
}

fn main() {
    match read_file("foo.txt") {
        Ok(_) => println!("파일 내용을 찾았습니다!"),
        Err(err) => println!("오류로 파일 내용 가져오기에 실패했습니다: {}", err)
    };
}
```


파일 입력을 효율적으로 읽으려면 어떻게 해야 하나요? (#어떻게-파일 입력을-효율적으로-읽는가)

파일(https://doc.rust-lang.org/stable/std/fs/struct.File.html) 유형은 읽기(h t t p s : / / d o c . r u s t - l a n g . o r g / s t a b l e / s t d / i o / t r a i t . R e a d . h t m l) 특성을 구현하며, 여기에는 `read()` (https://doc.rust- lang.org/stable/std/io/trait.Read.html#tymethod.read), `read_to_end()` (https://doc.rust- lang.org/stable/std/io/trait.Read.html#method.read_to_end), `bytes()` (https://doc.rust- lang.org/stable/std/io/trait.Read.html#method.bytes), `chars()` (https://doc.rust-lang.org/stable/std/io/trait.Read.html#method.chars), `take()` (https://doc.rust-lang.org/stable/std/io/trait.Read.html#method.take)를 참조하세요. 이러한 각 함수는 주어진 파일에서 일정량의 입력을 읽습니다. `read()` (https://doc.rust-lang.org/stable/std/io/trait.Read.html#tymethod.read)는 한 번의 호출로 기본 시스템이 제공하는 만큼의 입력을 읽습니다. `read_to_end()` (https://doc.rust-lang.org/stable/std/io/trait.Read.html#method.read_to_end)는 전체 버퍼를 벡터로 읽어 필요한 만큼의 공간을 할당합니다. `bytes()` (https://doc.rust- lang.org/stable/std/io/trait.Read.html#method.bytes) 및 `chars()` (https://doc.rust- lang.org/stable/std/io/trait.Read.html#method.chars)는 각각 파일의 바이트와 문자를 반복할 수 있게 해 줍니다. 마지막으로 `take()` (https://doc.rust-lang.org/stable/std/io/trait.Read.html#method.take)를 사용하면 파일에서 임의의 바이트 수까지 읽을 수 있습니다. 이 함수를 종합하면 필요한 모든 데이터를 효율적으로 읽을 수 있습니다.

버퍼링된 읽기의 경우, 읽기 시 시스템 호출 횟수를 줄이는 데 도움이 되는 `BufReader`(https://doc.rust-lang.org/stable/std/io/struct.BufReader.html) 구조체를 사용합니다.

Rust에서 비동기 입출력을 하려면 어떻게 해야 하나요? (#how-do-i-do-asynchronous-input-output-in-rust)

Rust에서 비동기 입출력을 제공하는 라이브러리는 mio(https://github.com/carllerche/mio), tokio(https://github.com/tokio-rs/tokio-core), mioco(https://github.com/dpc/mioco), coio-rs(https://github.com/zonyitoo/coio-rs), rotor(https://github.com/tailhook/rotor) 등 여러 가지가 있습니다.

Rust에서 명령줄 인수를 얻으려면 어떻게 해야 하나요? (#how-do-i-get-command-line-arguments)

가장 쉬운 방법은 입력 인자에 대한 이터레이터를 제공하는 `Args`(https://doc.rust-lang.org/stable/std/env/struct.Args.html)를 사용하는 것입니다.

더 강력한 기능을 원하신다면, crates.io(https://crates.io/keywords/argument)에 다양한 옵션이 있습니다! (https://www.rust-lang.org/)

오류 처리

Rust에 예외가 없는 이유는 무엇인가요? (#왜-녹슬지-않는-예외가-있는가)

예외는 제어 흐름에 대한 이해를 복잡하게 만들고, 타입 시스템 외부에서 유효성/무효성을 표현하며, 멀티스레드 코드(Rust의 주요 초점)와 제대로 상호 운용되지 않습니다.

Rust는 오류 처리에 대한 유형 기반 접근 방식을 선호하며, 이는 책(https://doc.rust-lang.org/stable/book/second-edition/ch09-00-error-handling.html)에서 자세히 다루고 있습니다. 이는 Rust의 제어 흐름, 동시성 및 기타 모든 것에 더 잘 맞습니다.

도대체 왜 모든 곳에서 `unwrap()`을 사용하나요? (#whats-the-deal-with-unwrap)

`unwrap()`은 `Option`(https://doc.rust- lang.org/stable/std/option/enum.Option.html) 또는 `Result`(https://doc.rust- lang.org/stable/std/결과/enum.Result.html) 내부의 값을 추출하고 값이 없으면 패닉을 발생시키는 함수입니다.

`unwrap()`은 잘못된 사용자 입력과 같이 발생할 것으로 예상되는 오류를 처리하는 기본 방법이 되어서는 안 됩니다. 프로덕션 코드에서는 값이 비어 있지 않다는 어설션처럼 취급해야 하며, 이를 위반하면 프로그램이 충돌하게 됩니다.

아직 오류를 처리하고 싶지 않은 빠른 프로토타입이나 오류 처리로 인해 요점이 흐려질 수 있는 블로그 게시물에도 유용합니다.

`try!` 매크로를 사용하는 예제 코드를 실행하려고 할 때 오류가 발생하는 이유는 무엇인가요? (#why-do-i-get-error- with-try)

함수의 반환 유형에 문제가 있는 것일 수 있습니다. `try!` (https://doc.rust-lang.org/stable/std/macro.try!.html) 매크로는 `Result`(https://doc.rust-lang.org/stable/std/result/enum.Result.html)에서 값을 추출하거나 `Result`(https://doc.rust-lang.org/stable/std/result/enum.Result.html)가 가지고 있는 오류를 가지고 일찍 반환합니다. 즉, `try`(https://doc.rust- lang.org/stable/std/macro.try!.html)는 `Err` -구체화된 타입이 `From::from(err)`을 구현하는 `Result`(https://doc.rust- lang.org/stable/std/result/enum.Result.html) 자체를 반환하는 함수에서만 작동합니다. 특히, 이것은 `try!` (https://doc.rust- lang.org/stable/std/macro.try!.html) 매크로는 메인 함수

내부에서 작동할 수 없습니다.

모든 곳에 결과를 묻는 것보다 오류 처리를 더 쉽게 할 수 있는 방법이 있나요? (#error-handling-without- result)

다른 사람의 코드에서 `Result` s(https://doc.rust- lang.org/stable/std/result/enum.Result.html)를 처리하지 않는 방법을 찾고 있다면 항상 `unwrap()` (https://doc.rust-lang.org/stable/core/option/enum.Option.html#method.unwrap)이 있지만 원하는 것이 아닐 수도 있습니다. 결과(https://doc.rust-lang.org/stable/std/result/enum.Result.html)는 일부 계산이 성공적으로 완료될 수도 있고 완료되지 않을 수도 있음을 나타내는 지표입니다. 이러한 실패를 명시적으로 처리하도록 요구하는 것은 Rust가 견고성을 장려하는 방법 중 하나입니다. Rust는 `try!` 매크로(https://doc.rust-lang.org/stable/std/macro.try!.html)와 같은 도구를 제공하여 실패를 인체공학적으로 처리할 수 있도록 합니다.

오류를 처리하고 싶지 않다면 `unwrap()`(https://doc.rust- lang.org/std/option/enum.Option.html#method.unwrap) 또는 `expect`(https://doc.rust-lang.org/std/option/enum.Option.html#method.expect)를 사용하면실패 시 코드가 패닉 상태에 빠지고 일반적으로 프로세스가 종료됩니다.

동시성

안전하지 않은 블록 없이 스레드 전체에서 정적 값을 사용할 수 있나요? (#can-i-use-static-values-across-threads- without-un-safe-block)

동기화된 경우 변이는 안전합니다. 정적 `Mutex`(https://doc.rust-lang.org/stable/std/sync/struct.Mutex.html)를 돌연변이(lazy-static(https://crates.io/crates/lazy_static/) 상자를 통해 지연 초기화)해도 안전하지 않은 블록이 필요하지 않으며, 정적 `AtomicUsize`(https://doc.rust-lang.org/stable/std/sync/atomic/struct.AtomicUsize.html)(lazy_static 없이 초기화 가능)를 돌연변이해도 마찬가지입니다.

보다 일반적으로 유형이 동기화(https://doc.rust-lang.org/stable/std/marker/trait.Sync.html)를 구현하고 드롭(https://doc.rust-lang.org/stable/std/ops/trait.Drop.html)을 구현하지 않는 경우 정적(https://doc.rust-lang.org/book/const-and-static.html#static)에서 사용할 수 있습니다.

매크로

식별자를 생성하는 매크로를 작성할 수 있나요? (#can-i-write-a-macro-to-generate-identifiers)

현재는 없습니다. 러스트 매크로는 "위생 매크로"(https://en.wikipedia.org/wiki/Hygienic_macro)로, 다른 식별자와 예기치 않은 충돌을 일으킬 수 있는 식별자의 캡처나 생성을 의도적으로 피합니다. 이 매크로의 기능은 일반적으로 C 전처리기와 관련된 매크로 스타일과는 크게 다릅니다. 매크로 호출은 항목, 메서드 선언, 문, 표현식, 패턴 등 매크로가 명시적으로 지원되는 위치에서만 나타날 수 있습니다. 여기서 "메서드 선언"은 메서드를 넣을 수 있는 빈 공간을 의미합니다. 부분적인 메서드 선언을 완성하는 데는 사용할 수 없습니다. 같은 논리로 부분 변수 선언을 완성하는 데에도 사용할 수 없습니다.

디버깅 및 툴링

Rust 프로그램을 디버깅하려면 어떻게 하나요? (#how-do-i-debug-rust-programs)

Rust 프로그램은 C 및 C++와 동일하게 gdb(https://sourceware.org/gdb/current/onlinedocs/gdb/) 또는 lldb(http://lldb.llvm.org/tutorial.html)를 사용하여 디버깅할 수 있습니다. 실제로 모든 Rust 설치에는 플랫폼 지원 여부에 따라 rust-gdb와 rust-lldb 중 하나 또는 둘 다 제공됩니다. 이 래퍼는 Rust 예쁜 인쇄가 활성화된 gdb 및 lldb에 대한 래퍼입니다.

표준 라이브러리 코드에서 패닉이 발생했다고 `rustc`가 말했습니다. 코드에서 실수를 찾으려면 어떻게 해야 하나요? (#어떻게 패닉을 찾나요)

이 오류는 일반적으로 클라이언트 코드에서 `unwrap()` ing(https://doc.rust-lang.org/stable/core/option/enum.Option.html#method.unwrap)의 `None` 또는 `Err`로 인해 발생합니다. 환경 변수 `RUST_BACKTRACE=1`을 설정하여 백트레이스를 활성화하면 자세한 정보를 얻는 데 도움이 됩니다.

디버그 모드(카고 빌드의 기본값)에서 컴파일하는 것도 도움이 됩니다. 제공된 디버거와 같은 디버거를 사용하면

`rust-gdb` 또는 rust-lldb는 a[illegible] (https://www.rust-lang.org/)

어떤 IDE를 사용해야 하나요? (#what-ide-should-i-use)

Rust를 사용하는 개발 환경에는 여러 가지 옵션이 있으며, 모든 옵션은 공식 IDE 지원 페이지 (https://forge.rust-lang.org/ides.html)에 자세히 설명되어 있습니다.

`GOFMT`는 훌륭합니다. 러스트프엠티는 어디에 있나요? (#어디서-러스트에프엠티)

rustfmt는 바로 여기(https://github.com/rust-lang-nursery/rustfmt)에서 확인할 수 있으며, Rust 코드를 최대한 쉽고 예측 가능하게 읽을 수 있도록 활발히 개발되고 있습니다.

낮은 수준

바이트 단위로 메모피하려면 어떻게 하나요? (#how-do-i-memcpy-bytes)

기존 슬라이스를 안전하게 복제하려면 `clone_from_슬라이스`(https://doc.rust-lang.org/stable/std/primitive.slice.html#method.clone_from_슬라이스)를 사용할 수 있습니다.

겹칠 가능성이 있는 바이트를 복사하려면 복사(https://doc.rust-lang.org/stable/std/ptr/fn.copy.html)를 사용합니다. 겹치지 않는 바이트를 복사하려면 복사_비중복 (https://doc.rust-lang.org/stable/std/ptr/fn.copy_nonoverlapping.html)을 사용하세요. 이 두 함수는 모두 언어의 안전 보장을 무력화시키는 데 사용될 수 있으므로 안전하지 않습니다. 사용할 때 주의하세요.

표준 라이브러리 없이도 Rust가 정상적으로 작동할 수 있나요? (#does-rust-work-without-the-standard-library)

물론입니다. Rust 프로그램은 `#![no_std]` 속성을 사용하여 표준 라이브러리를 로드하지 않도록 설정할 수 있습니다. 이 속성을 설정하면 플랫폼에 구애받지 않는 프리미티브만 있는 Rust 코어 라이브러리를 계속 사용할 수 있습니다. 따라서 IO, 동시성, 힙 할당 등은 포함되지 않습니다.

Rust로 운영체제를 작성할 수 있나요? (#can-i-write-an-operating-system-in-rust)

예! 실제로 이를 위한 여러 프로젝트가 진행 중입니다(http://wiki.osdev.org/Rust).

파일이나 기타 바이트 스트림에서 빅 엔디안 또는 리틀 엔디안 형식의 `i32` 또는 `f64`와 같은 숫자 유형

을 읽거나 쓰려면 어떻게 해야 하나요? (#how-can-i-write-endian-independent-values)

이를 위한 유틸리티를 제공하는 바이트오더 크레이트(http://burntsushi.net/rustdoc/byteorder/)를 확인해 보세요.

Rust는 특정 데이터 레이아웃을 보장하나요? (#does-rust-guarantee-data-layout)

기본값은 아닙니다. 일반적인 경우 열거형과 구조체 레이아웃은 정의되지 않습니다. 이를 통해 컴파일러는 잠재적으로 중첩된 변형을 식별하고 압축하기 위해 패딩을 재사용하는 등의 최적화를 수행할 수 있습니다.

열거형, 패딩을 제거하기 위해 필드 재정렬 등 데이터를 전달하지 않는 열거형("C형")은 정의된 표현을 사용할 수 있습니다. 이러한 열거형은 데이터가 없는 이름 목록이라는 점에서 쉽게 구분할 수 있습니다:

```
열거형 CLike {
    A,
    B = 32,
    C = 34,
    D
}
```

새 사이트를 사용해 보세요! (https://www.rust-lang.org/)

이러한 열거형에 `#[repr(C)]` 속성을 적용하여 동등한 C 코드에서와 동일한 표현을 제공할 수 있습니다. 이를 통해 대부분의 사용 사례에서 C 열거형도 사용되는 FFI 코드에서 Rust 열거형을 사용할 수 있습니다. 이 속성을 구조체에도 적용하여 C 구조체와 동일한 레이아웃을 얻을 수도 있습니다.

크로스 플랫폼

Rust에서 플랫폼별 동작을 표현하는 관용적인 방법은 무엇인가요? (#how-do-i-express-platform-specific-behavior)

플랫폼별 동작은 `target_os` , `target_family` , `target_endian` 등과 같은 조건부 컴파일 속성(https://doc.rust- lang.org/reference/attributes.html#conditional-compilation)을 사용하여 표현할 수 있습니다.

Rust를 Android/iOS 프로그래밍에 사용할 수 있나요? (#can-rust-be-used-for-android-ios-programs)

가능합니다! 이미 Android(https://github.com/tomaka/android- rs-glue)와 iOS(https://www.bignerdranch.com/blog/building-an-ios-app-in-rust-part-1/) 모두에서 Rust를 사용한 예가 있습니다. 설정하는 데 약간의 작업이 필요하지만 Rust는 두 플랫폼 모두에서 잘 작동합니다.

웹 브라우저에서 Rust 프로그램을 실행할 수 있나요? (#can-i-run-my-rust-program-in-a-web-browser)

아마도요. Rust는 asm.js(http://asmjs.org/)와 WebAssembly(http://webassembly.org/) 모두에 대해 실험적으로 지원(https://davidmcneil.gitbooks.io/the-rusty-web/)하고 있습니다.

Rust에서 교차 컴파일하려면 어떻게 하나요? (#how-do-i-cross-compile-rust)

Rust에서는 교차 컴파일이 가능하지만 설정하려면 약간의 작업(https://github.com/japaric/rust-cross/blob/master/README.md)이 필요합니다. 모든 Rust 컴파일러는 크로스 컴파일러이지만 라이브러리는 대상 플랫폼에 맞게 크로스 컴파일해야 합니다.

Rust는 지원되는 각 플랫폼에 대한 표준 라이브러리(https://static.rust-lang.org/dist/index.html)의 사본을 배

포하고 있으며, 배포 페이지에서 찾을 수 있는 각 빌드 디렉터리의 `rust-std-*` 파일에 포함되어 있지만 아직 자동화된 설치 방법은 없습니다.

모듈 및 상자

모듈과 크레이트의 관계는 무엇인가요? (#모듈과 상자 사이의 관계는 무엇인가요?)

- 크레이트는 컴파일 단위로, Rust 컴파일러가 작동할 수 있는 가장 작은 양의 코드입니다.

  를 켜세요. 새 사이트를 사용해 보세요! (https://www.rust-lang.org/)

- 모듈은 상자 안에 있는 (중첩된) 코드 구성 단위입니다. • 상자에는 이름이

  지정되지 않은 최상위 모듈이 암시적으로 포함되어 있습니다.

- 재귀 정의는 모듈에 포함될 수 있지만 상자는 포함되지 않습니다.

왜 Rust 컴파일러가 제가 사용 중인 라이브러리를 찾을 수 없나요? (#왜-러스트 컴파일러가-사용 중인 라이브러리를-찾을-수-없나요?)

여러 가지 답이 있을 수 있지만, 일반적인 실수는 사용 선언이 상자 루트에 상대적이라는 것을 깨닫지 못하는 것입니다. 프로젝트의 루트 파일에 정의된 경우 사용할 경로를 사용하도록 선언을 다시 작성해 보고 문제가 해결되는지 확인하세요.

또한 사용 경로를 각각 현재 모듈 또는 상위 모듈에 상대적인 것으로 명확히 하는 `self` 및 `super`도 있습니다.

라이브러리 사용에 대한 자세한 내용은 Rust 책의 "상자 및 모듈" 장(https://doc.rust-lang.org/stable/book/second-edition/ch07-00-modules.html)을 참조하세요.

모듈 파일을 그냥 사용하지 않고 상자의 최상위 레벨에 `mod`로 선언해야 하는 이유는 무엇인가요? (#왜-나는-모듈을-모드로-선언해야 하나요?)

Rust에서 모듈을 선언하는 방법에는 인라인 또는 다른 파일에 선언하는 두 가지 방법이 있습니다. 다음은 각각에 대한 예시입니다:

```
// In main.rs
mod hello {
    pub fn f() {
        println!("안녕하세요!");
    }
}

fn main() {
    hello::f();
}
```

```
// In main.rs
mod hello;

fn main() {
    hello::f();
}

// In hello.rs
pub fn f() {
    println!("안녕하세요!");
}
```

첫 번째 예제에서는 모듈이 사용되는 파일과 동일한 파일에 정의되어 있습니다. 두 번째 예제에서는 메인 파일의 모듈 선언이 컴파일러에 `hello.rs` 또는 `hello/mod.rs` 중 하나를 찾아서 해당 파일을 로드하도록 지시합니다.

mod와 use의 차이점에 유의하세요. mod는 모듈이 존재한다고 선언하는 반면, use는 다른곳에서선언된모듈을 참조합니다.

프록시를 사용하도록 Cargo를 구성하려면 어떻게 해야 하나요? (#how-do-i-configure-cargo-to-use-a-proxy)

Cargo 설정 문서(http://doc.crates.io/config.html)에 설명된 대로, 설정 파일의 [http] 아래에 "proxy" 변수를 설정하여 Cargo가 프록시를 사용하도록 설정할 수 있습니다.

이미 크레이트를 사용하고 있는데도 컴파일러가 메서드 구현을 찾을 수 없는 이유는 무엇인가요? (#왜- 컴파일러가 메서드 구현을 찾을 수 없나요?)

특성에 정의된 메서드의 경우 특성 선언을 명시적으로 임포트해야 합니다. 즉, 구조체가 해당 특성을 구현하는 모듈을 임포트하는 것만으로는 충분하지 않으며 특성 자체도 임포트해야 합니다.

컴파일러가 사용 선언을 유추할 수 없는 이유는 무엇인가요? (#왜 컴파일러가 사용 선언문을 유추할 수 없나요?)

아마도 그럴 수도 있지만 원하지 않을 수도 있습니다. 많은 경우 컴파일러가 단순히 주어진 식별자가 정의된 위치를 찾아 임포트할 올바른 모듈을 결정할 수 있지만, 일반적으로는 그렇지 않을 수도 있습니다. 경쟁 옵션 중 하나를 선택하기 위한 rustc의 결정 규칙은 경우에 따라 놀라움과 혼란을 야기할 수 있으며, Rust는 이름에 대한 출처를 명시하는 것을 선호합니다.

예를 들어, 컴파일러는 식별자 정의가 경쟁하는 경우 가장 먼저 가져온 모듈의 정의가 선택된다고 말할 수 있습니다. 따라서 모듈 foo와 모듈 bar가 모두 식별자 baz 를 정의하지만 foo가 가장 먼저 등록된 모듈인 경우 컴파일러는 사용 foo::baz; 를 삽입합니다.

```
모드 푸; 모드
바;

// 컴파일러가 삽입할 foo::baz //를 사용합니다.

fn main() {
  baz();
}
```

이런 일이 발생할 것을 알고 있다면 키 입력 횟수를 줄일 수 있지만, 실제로는 바즈() 를 바::바즈() 로만들고했는데예상치 못한 오류 메시지가 표시될 가능성이 크게 증가하고, 함수 호출의 의미가 모듈 선언에 의존하게 되어 코드 가독성이 떨어집니다. 이는 우리가 기꺼이 감수할 수 있는 절충안이 아닙니다.

하지만 앞으로는 IDE가 선언을 관리하는 데 도움을 줄 수 있으므로 이름을 가져오는 데는 기계의 도움을 받

지만 그 이름의 출처에 대한 명시적인 선언이라는 두 가지 장점을 모두 누릴 수 있습니다.

동적 Rust 라이브러리 로딩은 어떻게 하나요? (#how-do-i-do-dynamic-rust-library-loading)

동적 링크를 위한 크로스 플랫폼 시스템을 제공하는 라이브러로딩 (https://crates.io/crates/libloading)을 사용하여 Rust에서 동적 라이브러리를 가져오세요.

crates.io에 네임스페이스가 없는 이유는 무엇인가요? (#why-doesnt-crates-io-have-namespaces)

의 공식 설명(https://internals.rust-lang.org/t/crates-io-package-policies/1041)을 인용합니다.

https://crates.io (https://crates.io)사이트! (https://www.rust-lang.org/)

> crates.io를 사용한 첫 달에 많은 분들이 네임스페이스 패키지(https://github.com/rust-lang/crates.io/issues/58) 도입 가능성에 대해 문의해 주셨습니다.
>
> 네임스페이스 패키지를 사용하면 여러 작성자가 하나의 일반적인 이름을 사용할 수 있지만, Rust 코드에서 패키지가 참조되는 방식과 패키지에 대한 사람 간 커뮤니케이션이 복잡해집니다. 언뜻 보기에는 여러 작성자가 `http`와같은 이름을 주장할 수 있지만, 이는 단순히 사람들이 해당 패키지를 `wycats`의 `http` 또는 `reem`의 `http`로 참조해야 한다는 것을 의미하며, `wycats-http` 또는 `reem-http`와 같은 패키지 이름에 비해 이점이 거의 없습니다.
>
> 네임스페이싱이 없는 패키지 에코시스템을 살펴본 결과, 사람들은 "tenderlove's libxml2"와 같은 이름 대신 `nokogiri`와 같은 더 창의적인 이름을 사용하는 경향이 있다는 것을 발견했습니다. 이러한 창의적인 이름은 부분적으로는 계층 구조가 없기 때문에 짧고 기억하기 쉬운 경향이 있습니다. 패키지에 대해 간결하고 명확하게 전달하기가 더 쉬워집니다. 흥미로운 브랜드를 만들 수 있습니다. 그리고 우리는 단일 네임스페이스 내에서 커뮤니티가 번창하고 있는 NPM 및 RubyGems와 같은 10,000개 이상의 패키지 에코시스템이 성공하는 것을 보았습니다.

요컨대, 저희는 피스톤이 단순히 피스톤이라는 이름 대신 (다른 사용자들이 와이캣츠/게임 엔진과 같은 이름을 선택할 수 있도록) bvssvni/게임 엔진과 같은 이름을 선택했다면 카고 생태계가 더 나아질 것이라고 생각합니다.

네임스페이스는 여러 가지 면에서 엄밀히 말해 더 복잡하고, 향후 필요할 경우 호환 가능하게 추가할 수 있기 때문에 단일 공유 네임스페이스를 고수할 것입니다.

라이브러리

HTTP 요청은 어떻게 하나요? (#how-can-i-make-an-http-request)

표준 라이브러리에는 HTTP 구현이 포함되어 있지 않으므로 외부 크레이트를 사용해야 합니다. reqwest(http://docs.rs/reqwest)가 가장 간단합니다. hyper(https://github.com/hyperium/hyper)를 기반으로 하고 Rust로 작성되었지만 다른 여러 가지도 있습니다(https://crates.io/keywords/http). curl(https://docs.rs/curl) 크레이트는 널리 사용되며 curl 라이브러리에 대한 바인딩을 제공합니다.

Rust에서 GUI 애플리케이션을 작성하려면 어떻게 해야 하나요? (#how-can-i-write-a-gui-application)

Rust에서 GUI 애플리케이션을 작성하는 방법에는 여러 가지가 있습니다. 이 GUI 프레임워크 목록 (https://github.com/kud1ing/awesome-rust#gui)을 확인하세요.

JSON/XML을 구문 분석하려면 어떻게 해야 하나요? (#how-can-i-parse-json-xml)

Serde(https://github.com/serde-rs/serde)는 다양한 형식으로 Rust 데이터를 직렬화 및 역직렬화하는 데 권장되는 라이브러리입니다.

표준 2D+ 벡터 및 도형 상자가 있나요? (#is-there-a-standard-2d-vector-crate)

아직 없습니다! 작성하고 싶으신가요?

Rust에서 OpenGL 앱을 작성하려면 어떻게 해야 하나요? (#how-do-i-write-an-opengl-app)

새 사이트를 사용해 보세요! (https://www.rust-lang.org/)

Glium(https://github.com/tomaka/glium)은 Rust에서 OpenGL 프로그래밍을 위한 주요 라이브러리입니다. GLFW(https://github.com/bjz/glfw-rs)도 좋은 옵션입니다.

Rust로 비디오 게임을 작성할 수 있나요? (#can-i-write-a-video-game-in-rust)

할 수 있습니다! Rust의 주요 게임 프로그래밍 라이브러리는 Piston(http://www.piston.rs/)이며, Rust의 게임 프로그래밍을 위한 서브 레딧(https://www.reddit.com/r/rust_gamedev/)과 IRC 채널(Mozilla IRC(https://wiki.mozilla.org/IRC)의 `#rust-gamedev`)도 있습니다.

디자인 패턴

Rust는 객체 지향인가요? (#is-rust-객체 지향)

멀티 패러다임입니다. OO 언어로 할 수 있는 많은 작업을 Rust에서도 할 수 있지만 모든 작업을 할 수 있는 것은 아니며, 항상 익숙한 추상화를 사용하는 것은 아닙니다.

객체 지향 개념을 Rust에 매핑하려면 어떻게 해야 하나요? (#how-do-i-map-객체 지향 개념을 러스트에 매핑하는 방법)

상황에 따라 다릅니다. 다중 상속 (https://www.reddit.com/r/rust/comments/2sryuw/ideaquestion_about_multiple_inheritence/)*과* 같은 객체 지향 개념을 Rust로 번역하는 방법이 있지만, Rust는 객체 지향이 아니므로 번역 결과는 OO 언어의 모습과 크게 다를 수 있습니다.

선택적 매개변수가 있는 구조체의 구성은 어떻게 처리하나요? (#선택적 매개변수가 있는 구조체를 어떻게 구성하나요?)

가장 쉬운 방법은 구조체의 인스턴스를 생성하는 데 사용하는 함수(일반적으로 `new()`)에 `Option` (https://doc.rust-lang.org/stable/std/option/enum.Option.html) 타입을 사용하는 것입니다. 또 다른 방법은 빌더 패턴(https://doc.rust-lang.org/stable/book/first-edition/method-syntax.html#builder- 패턴)을 사용하는 것인데, 빌더 패턴에서는 멤버 변수를 인스턴스화하는 특정 함수만 빌드된 타입을 구성하기 전에 호출해야 합니다.

Rust에서 전역 변수는 어떻게 하나요? (#how-do-i-do-global-variables)

Rust의 전역은 컴파일 타임에 계산된 전역 상수에 대해 `const` 선언을 사용하여 수행할 수 있으며, 변경 가능한 전역에는 정적을 사용할 수 있습니다. 정적 뮤트 변수를 수정하려면 안전하지 않은 를 사용해야 하는데, 이는 안전한 Rust에서는 발생하지 않도록 보장된 것 중 하나인 데이터 경합을 허용하기 때문입니다. 정적 값과 정적 값의 한 가지 중요한 차이점은 정적 값에 대한 참조는 가능하지만 지정된 메모리 위치가 없는 정적 값에 대한 참조는 불가능하다는 점입니다. const와 정적 에 대한 자세한 내용은 Rust 책(https://doc.rust-lang.org/book/const-and-static.html)을 참조하십시오.

절차적으로 정의된 컴파일 시간 상수를 설정하려면 어떻게 해야 하나요? (#how-can-i-set-compile-time-상수를-절차적으로-정의하는-방법)

Rust는 현재 컴파일 시간 상수를 제한적으로 지원합니다. 정적 선언과 유사하지만 변경 불가능하고 메모리에 지정된 위치가 없는 `const` 선언을 사용하여 프리미티브를 정의할 수 있을 뿐만 아니라 `const` 함수 및 고유 메서드를 정의할 수도 있습니다.

이러한 메커니즘을 통해 정의할 수 없는 절차 상수를 정의하려면 `lazy-static`(https://github.com/rust-langT-rnyuoruset rtyh/elanzeyw-stsaitteic! .(rhs)ttcprsa:t/e/,wwwhwic.hs srcgo/)mpile-time평가를사용하여처음 사용할 때 상수를 자동으로 평가할 수있습니다.

메인 실행 전에 초기화 코드를 실행할 수 있나요? (#can-i-run-code-before-main)

Rust에는 "메인 이전 생명체"라는 개념이 없습니다. 가장 근접한 것은 정적 변수를 처음 사용할 때 느리게 초기화하여 "비포 메인"을 시뮬레이션하는 `lazy-static`(https://github.com/Kimundi/lazy-static.rs) 상자를 통해 확인할 수 있습니다.

Rust는 전역에 상수 표현식이 아닌 값을 허용하나요? (#does-rust-allow-non-常수 표현식-값-for-globals)

전역에는 상수 표현식이 아닌 생성자를 가질 수 없으며 소멸자도 전혀 가질 수 없습니다. 정적 생성자는 정적 초기화 순서를 이식적으로 보장하기 어렵기 때문에 바람직하지 않습니다. 메인 이전의 생명은 종종 결함으로 간주되므로 Rust는 이를 허용하지 않습니다.

'정적 초기화 순서 실패'에 대한 C++ FQA(http://yosefk.com/c++fqa/ctors.html#fqa-10.12)와 이 기능이 있는 C#의 문제점에 대한 Eric Lippert의 블로그(https://ericlippert.com/2013/02/06/static-constructors-part-one/)를 참조하세요.

지연 정적(https://crates.io/crates/lazy_static/) 상자를 사용하여 상수 표현식이 아닌 전역의 근사치를 구할 수 있습니다.

기타 언어

C의 구조체 `X { static int X; };` 와 같은 것을 Rust에서 구현하려면 어떻게 해야 하나요? (#how-can-i-use-static- fields)

Rust에는 위의 코드 스니펫에 표시된 것처럼 정적 필드가 없습니다. 대신 정적 필드를 선언할 수 있습니다. 변수를 해당 모듈에 비공개로 유지합니다.

C 스타일 열거형을 정수로 변환하거나 그 반대로 변환하려면 어떻게 해야 하나요? (#how-can-i-convert-a-c-style-enum-to-an-integer)

C 스타일 열거형을 정수로 변환하는 것은 `e`를 `i64`(여기서 `e`는 일부 열거형입니다)와 같은 `as` 표현식을 사용하여 수행할 수 있습니다.

다른 방향으로 변환하는 것은 다른 숫자 값을 열거형의 다른 잠재적 값에 매핑하는 일치 문을 사용하여 수행할 수 있습니다.

Rust 프로그램이 C 프로그램보다 바이너리 크기가 더 큰 이유는 무엇인가요? (#왜-러스트-프로그램은-C-프로그램보다-바이너리-크기가-더-큰가)

Rust 프로그램이 기본적으로 기능적으로 동등한 C 프로그램보다 더 큰 바이너리 크기를 갖는 데에는 몇 가지 요인이 있습니다. 일반적으로 Rust는 작은 프로그램의 크기가 아닌 실제 프로그램의 성능에 맞게 최적화하는 것을 선호합니다.

Monomorphization

Rust는 제네릭을 단형화하므로 제네릭 함수 또는 유형이사용되는각구체적인유형에대해 새로운 버전의 제네릭 함수가 생성됩니다. !T(hhitstpisss/i/mwilwawr 는 C++에서작동합니다.예를들어,다음 프로그램에서:

```
fn foo<T>(t: T) {
    // ... 무언가를 하세요
}

fn main() {
    foo(10);        // i32
    foo("hello"); // &str
}
```

최종 바이너리에는 두 가지 버전의 `foo`가 존재하는데, 하나는 `i32` 입력에 특화되고 다른 하나는 `&str` 입력에 특화됩니다. 이렇게 하면 일반 함수의 정적 디스패치를 효율적으로 수행할 수 있지만, 바이너리가 커지는 대가를 치르게 됩니다.

Debug symbols

Rust 프로그램은 릴리스 모드에서 컴파일할 때에도 일부 디버그 심볼을 유지한 채로 컴파일합니다. 이러한 디버그 심볼은 패닉에 대한 백트레이스 제공에 사용되며 스트립 또는 다른 디버그 심볼 제거 도구로 제거할 수 있습니다. Cargo를 사용하여 릴리스 모드에서 컴파일하는 것은 최적화 레벨 3을 rustc로 설정하는 것과 동일하다는 점도 알아두면 유용합니다. 최근 대체 최적화 수준(`s` 또는 `z`라고 함)이 등장했으며 (https://github.com/rust-lang/rust/pull/32386), 컴파일러가 성능보다는 크기에 맞게 최적화하도록 지시합니다.

Jemalloc

Rust는 컴파일된 Rust 바이너리에 약간의 크기를 추가하는 기본 얼로케이터로 jemalloc을 사용합니다. 시스템에서 제공하는 여러 일반적인 얼로케이터와 비교했을 때 일관되고 우수한 성능 특성을 가진 얼로케이터이기 때문에 Jemalloc이 선택되었습니다. 사용자 정의 얼로케이터(https://github.com/rust-lang/rust/issues/32838)를 더 쉽게 사용할 수 있도록 하는 작업이 진행 중이지만 아직 완료되지 않았습니다.

Link-time optimization

Rust는 기본적으로 링크 시간 최적화를 수행하지 않지만, 그렇게 하도록 지시할 수 있습니다. 이렇게 하면 Rust 컴파일러가 잠재적으로 수행할 수 있는 최적화의 양이 증가하고 바이너리 크기에 약간의 영향을 미칠 수 있습니다. 이 효과는 앞서 언급한 크기 최적화 모드와 함께 사용하면 더 커질 수 있습니다.

Standard library

Rust 표준 라이브러리에는 일부 프로그램에서는 바람직하지 않을 수 있는 libbacktrace와 libunwind가 포함되어 있습니다. 따라서 `#![no_std]`를 사용하면 바이너리가 더 작아질 수 있지만 일반적으로 작성 중인 Rust 코드의 종류가 크게 변경될 수 있습니다. 표준 라이브러리 없이 Rust를 사용하는 것이 기능적으로 동등한 C

코드에 더 가까운 경우가 많다는 점에 유의하세요.

예를 들어, 다음 C 프로그램은 이름을 읽고 해당 이름을 가진 사람에게 "안녕하세요"라고 말합니다.

```
#include <stdio.h>

int main(void) {
    printf("이름이 무엇입니까?"\n");
    char input[100] = {0};
    scanf("%s", input); printf("안녕하세요 %s!\n", input); return 0;
}
```

새 사이트를 사용해 보세요! (https://www.rust-lang.org/)

이 내용을 Rust에서 다시 작성하면 다음과 같은 결과가 나올 수 있습니다:

```
std::io를 사용합니다;

fn main() {
    println!("이름이 뭐예요?"); let mut input = String::new();
    io::stdin().read_line(&mut 입력).unwrap();
    println!("Hello {}!", input);
}
```

이 프로그램을 컴파일하여 C 프로그램과 비교하면 더 큰 바이너리를 가지며 더 많은 메모리를 사용합니다. 하지만 이 프로그램은 위의 C 코드와 정확히 동일하지는 않습니다. 이에 상응하는 Rust 코드는 다음과 같이 보일 것입니다:

```
#![feature(lang_items)]
#![feature(libc)]
#![feature(no_std)]
#![feature(start)]
#![NO STD]

EXTERNAL CRATE LIBC;

외부 "C" {
    fn printf(fmt: *const u8, ...) -> i32;
    fn scanf(fmt: *const u8, ...) -> i32;
}

#[시작]
fn start(_argc: isize, _argv: *const *const u8) -> isize {
    unsafe {
        printf(b"이름이 무엇입니까?\n\0".as_ptr());
        let mut input = [0u8; 100];
        scanf(b"%s\0".as_ptr(), &mut input);
        printf(b"안녕하세요 %s입니다!\n\0".as_ptr(),
        &input); 0
    }
}

#[lang="eh_personality"] extern fn eh_personality() {}
#[lang="panic_fmt"] fn panic_fmt() -> ! { loop {} }
#[lang="stack exhausted"] extern fn stack exhausted() {}
```

새 사이트를 사용해 보세요! (https://www.rust-lang.org/)

프로그래머의 복잡성이 증가하고 일반적으로 Rust에서 제공하는 정적 보증이 부족하다는 단점이 있지만 메모리 사용량에서 C와 거의 일치합니다(여기서는 안전하지 않은 를 사용하여 피함).

왜 Rust에는 C처럼 안정적인 ABI가 없으며, 왜 외부로 주석을 달아야 하나요? (#why- no-stable-abi)

ABI를 약속하는 것은 향후 잠재적으로 유리한 언어 변경을 제한할 수 있는 큰 결정입니다. Rust가 2015년 5월에야 1.0에 도달했다는 점을 감안하면, 안정적인 ABI와 같은 큰 약속을 하기에는 아직 이르다고 할 수 있습니다. 그렇다고 해서 앞으로 그런 일이 일어나지 않을 것이라는 의미는 아닙니다. (C++는 안정적인 ABI를 지정하지 않고도 수년 동안 버텨왔습니다.)

외부 키워드를 사용하면 Rust에서 다른 언어와의 상호 운용을 위해 잘 정의된 C ABI와 같은 특정 ABI를 사용할 수 있습니다.

Rust 코드로 C 코드를 호출할 수 있나요? (#can-rust-code-call-c-code)

예. Rust에서 C 코드를 호출하는 것은 C++에서 C 코드를 호출하는 것만큼 효율적이도록 설계되었습니다.

C 코드가 러스트 코드를 호출할 수 있나요? (#can-c-code-call-rust-code)

예. Rust 코드는 외부 선언을 통해 노출되어야 C-ABI와 호환됩니다. 이러한 함수는 함수 포인터로 C 코드에 전달하거나 `#[no_mangle]` 속성을 지정하여 심볼 맹글링을 비활성화하면 C 코드에서 직접 호출할 수 있습니다.

저는 이미 완벽한 C++를 작성합니다. Rust는 저에게 무엇을 제공하나요? (#why-rust-vs-cxx)

새 사이트를 사용해 보세요! (https://www.rust-lang.org/)

최신 C++에는 안전하고 올바른 코드를 작성하고 오류를 줄일 수 있는 많은 기능이 포함되어 있지만 완벽하지는 않으며 여전히 안전하지 않은 코드가 쉽게 도입될 수 있습니다. 이는 C++ 핵심 개발자들이 극복하기 위해 노력하고 있는 부분이지만, C++는 현재 구현하려고 하는 많은 아이디어보다 오래된 역사로 인해 제한이 있습니다.

Rust는 처음부터 안전한 시스템 프로그래밍 언어로 설계되었기 때문에 C++에서 안전성을 확보하는 것을 복잡하게 만드는 과거의 설계 결정에 제한을 받지 않습니다. C++에서 안전은 신중한 개인적 규율을 통해 달성할 수 있으며, 잘못하기 쉽습니다. Rust에서는 안전이 기본입니다. 안전 버그가 있는지 코드를 다시 확인하느라 시간을 낭비하지 않고도 자신보다 덜 완벽한 사람이 포함된 팀에서 작업할 수 있습니다.

Rust에서 C++ 템플릿 전문화에 해당하는 작업을 수행하려면 어떻게 해야 하나요? (#how-to-get-cxx-style-template- specialization)

Rust에는 현재 템플릿 전문화와 정확히 일치하는 기능이 없지만 작업 중이며(https://github.com/rust-lang/rfcs/pull/1210) 곧 추가될 예정입니다. 그러나 관련 유형을 통해 유사한 효과를 얻을 수 있습니다 (https://doc.rust-lang.org/stable/book/second-edition/ch19-04- advanced-types.html).

Rust의 소유권 시스템은 C++의 이동 시맨틱과 어떤 관련이 있나요? (#how-does-ownership-relate-to-cxx-move-semantics)

기본 개념은 비슷하지만 두 시스템은 실제로는 매우 다르게 작동합니다. 두 시스템 모두에서 값을 '이동'한다는 것은 기본 리소스에 대한 소유권을 이전하는 방법입니다. 예를 들어 문자열을 이동하면 문자열을 복사하는 것이 아니라 문자열의 버퍼를 전송합니다.

Rust에서는 소유권 이전이 기본 동작입니다. 예를 들어 문자열을 인수로 받는 함수를 작성하면 이 함수는 호출자가 제공한 문자열 값의 소유권을 가져옵니다:

```
fn process(s: String) { }

fn caller() {
    s = String::from("안녕하세요, 세상!");
    프로세스(들); // `s`의 소유권을 `process`로 이전합니다. 프로세스(들); // 오
    류! 소유권이 이미 이전되었습니다.
}
```

위의 스니펫에서 볼 수 있듯이 함수 호출자에서 첫 번째 프로세스 호출은 변수 s의 소유권을 이전합니다. 컴파일러는 소유권을 추적하므로 두 번째 프로세스 호출은 동일한 값의 소유권을 두 번 넘겨주는 것은 불법이므로 오류

가 발생합니다. 또한 Rust는 해당 값에 대한 미결 참조가 있는 경우 값을 이동할 수 없도록 합니다.

C++는 다른 접근 방식을 취합니다. C++에서 기본값은 값을 복사하는 것입니다(보다 구체적으로 복사 생성자를 호출하는 것). 그러나 호출자는 문자열과 같은 "rvalue 참조"를 사용하여 인수를 선언하여 해당 인수가 소유한 일부 리소스(이 경우 문자열의 내부 버퍼)에 대한 소유권을 가져갈 것임을 나타낼 수 있습니다. 그런 다음 호출자는 임시 표현식을 전달하거나 `std::move`를 사용하여 명시적으로 이동해야 합니다. 그러면 위의 함수 프로세스와 대략적으로 동일하게 될 것입니다:

```
void 프로세스(문자열&& sTry {유}t 새 사이트! (https://www.rust-lang.org/)

void caller() {
    문자열 s("안녕하세요, 세상!");
    process(std::move(s));
    프로세스(STD::이동);
}
```

C++ 컴파일러는 이동을 추적할 의무가 없습니다. 예를 들어, 위의 코드는 최소한 clang의 기본 설정을 사용하여 경고나 오류 없이 컴파일됩니다. 또한 C++에서는 문자열 `s` 자체(내부 버퍼가 아닌 경우)의 소유권이 호출자에게 있으므로 호출자가 반환할 때 `s`에 대한 소멸자가 실행됩니다(반대로 Rust에서는 이동된 값은 새 소유자에 의해서만 삭제됩니다).

Rust에서 C++로, 또는 C++에서 Rust로 상호 운용하려면 어떻게 해야 하나요? (#how-to-interoperate-with-cxx)

Rust와 C++는 모두 C용 외부 함수 인터페이스(https://doc.rust-lang.org/book/ffi.html)를 제공하며, 이를 사용하여 서로 통신할 수 있습니다. C 바인딩을 작성하는 것이 너무 지루하다면 언제든지 rust-bindgen(https://github.com/servo/rust-bindgen)을 사용하여 실행 가능한 C 바인딩을 자동으로 생성할 수 있습니다.

Rust에 C++ 스타일 생성자가 있나요? (#does-rust-have-cxx-style-constructors)

함수는 언어 복잡성을 추가하지 않고 생성자와 동일한 목적을 수행합니다. Rust에서 생성자와 동등한 함수의 일반적인 이름은 `new()`이지만, 이는 언어 규칙이라기보다는 관습에 불과합니다. 사실 `new()` 함수는 다른 함수와 똑같습니다. 그 예는 다음과 같습니다:

```
구조체 Foo {
    a: i32,
    B: F64,
    C: BOL,
}

impl Foo {
    fn new() -> Foo {
        Foo {
            a: 0,
            b: 0.0,
            c: false,
        }
    }
}
```

Rust에 복사 생성자가 있나요? (#does-rust-have-copy-constructors)

정확히는 아닙니다. `Copy`를 구현하는 타입은 추가 작업 없이 표준 C와 유사한 "얕은 복사"를 수행합니다 (C++의 간단한 복사 가능 타입과 유사). 사용자 정의 복사 동작이 필요한 복사 유형을 구현하는 것은 불가능합니다. 대신, Rust에서 "복사 생성자"는 복제 특성을 구현하고 명시적으로 복제 메서드를 호출하여 생성됩니다. 사용자 정의 복사 연산자를 명시적으로 만들면 근본적인 복잡성이 드러나므로 개발자가 잠재적으로 비용이 많이 드는 연산을 쉽게 식별할 수 있습니다.

Rust에 이동 생성자가 있나요? (#does-rust-have-move-constructors)

새 사이트를 사용해 보세요! (https://www.rust-lang.org/)

아니요. 모든 유형의 값은 `memcpy`를 통해 이동됩니다. 이렇게 하면 할당, 전달 및 반환에 언와인딩과 같은 부작용이 발생하지 않는 것으로 알려져 있으므로 일반 안전하지 않은 코드를 훨씬 간단하게 작성할 수 있습니다.

바둑과 러스트는 어떻게 비슷하고 어떻게 다른가요? (#비교-고-러스트)

Rust와 Go는 디자인 목표가 상당히 다릅니다. 다음 차이점은 나열하기에는 너무 많은 차이점이 있지만, 그 중 몇 가지 중요한 차이점입니다:

- Rust는 Go보다 수준이 낮습니다. 예를 들어, 가비지 콜렉터가 필요한 Go와는 달리 Rust는 가비지 콜렉터가 필요하지 않습니다. 일반적으로 Rust는 C 또는 C++와 비슷한 수준의 제어 기능을 제공합니다.
- Rust는 안전성과 효율성을 보장하는 동시에 높은 수준의 어포던스를 제공하는 데 중점을 두는 반면, Go는 작고 단순한 언어로 빠르게 컴파일되고 다양한 도구와 잘 작동하는 데 초점을 맞추고 있습니다.
- Rust는 제네릭을 강력하게 지원하지만 Go는 그렇지 않습니다.
- Rust는 하스켈의 타입 클래스에서 가져온 타입 시스템을 포함해 함수형 프로그래밍의 세계에서 많은 영향을 받았습니다. Go는 기본 일반 프로그래밍을 위한 인터페이스를 사용하는 더 단순한 타입 시스템을 가지고 있습니다.

Rust 특성은 하스켈 타입클래스와 어떻게 비교하나요? (#how-do-rust-traits-compare-to-haskell-typeclasses)

Rust 특성은 하스켈 타입 클래스와 유사하지만, 현재로서는 더 높은 종류의 타입을 표현할 수 없기 때문에 그다지 강력하지 않습니다. Rust의 연관된 형은 하스켈 형 계열과 동일합니다.

하스켈 타입클래스와 러스트 특성의 몇 가지 구체적인 차이점은 다음과 같습니다:

- Rust 특성에는 `Self`라는 암시적 첫 번째 매개변수가 있습니다. Rust의 특성 바는 다음에 해당합니다. 클래스는 하스켈의 `Bar` 자체에 해당하며, Rust의 `Bar<Foo>` 특성은 하스켈의 `Bar foo` 자체 클래스에 해당합니다.
- Rust에서 "슈퍼트래잇" 또는 "슈퍼클래스 제약 조건"은 다음과 같이 쓰이는 특성 `Sub: Super` 와 비교됩니다.

  하스켈의 슈퍼 셀프 `=>` 서브 셀프 클래스.
- Rust는 고아 인스턴스를 금지하므로 Rust의 일관성 규칙이 Haskell과 다릅니다.
- Rust의 임포트 해결은 두 일부가 겹치는지 여부를 결정하거나 잠재적 임포트 중에서 선택할 때 관련 위치 절과 특성 바운드를 고려합니다. 하스켈은 인스턴스 선언의 제약 조건만 고려하며 다른 곳에서 제공된 제약 조건은 무시합니다.

- Rust의 특성 하위 집합("객체 안전"(https://github.com/rust-lang/rfcs/blob/master/text/0255- object-safety.md) 특성)을 특성 객체를 통한 동적 디스패치에 사용할 수 있습니다. 하스켈에서는 GHC의 `ExistentialQuantification` 을 통해 동일한 기능을 사용할 수 있습니다.

문서

스택 오버플로우에 대한 Rust 답변이 왜 그렇게 많은가요? (#왜-왜-많은-러스트-답변이-스택오버플로우-왜-잘못되었는가)

Rust 언어는 수년 동안 사용되어 왔으며 2015년 5월에야 버전 1.0에 도달했습니다. 그 이전에는 언어가 크게 변경되어 이전 버전의 언어에서는 스택 오버플로우에 대한 답변이 많이 제공되었습니다.

시간이 지남에 따라 현재 버전에 대한 답변이 점점 더 많이 제공되어 오래된 답변의 비율이 높아짐에 따라 이 문제가 개선될 것입니다. (https://www.rust-lang.org/)

Rust 문서에서 문제를 어디에서 보고하나요? (#어디서-어디서-어디서-러스트 문서에 문제를 보고해야 하나요?)

Rust 컴파일러 이슈 트래커(https://github.com/rust-lang/rust/issues)의 Rust 문서에서 이슈를 보고할 수 있습니다. 먼저 기여 가이드라인(https://github.com/rust-lang/rust/blob/master/CONTRIBUTING.md#writing-documentation)을 읽어보시기 바랍니다.

프로젝트가 의존하는 라이브러리에 대한 rustdoc 문서를 보려면 어떻게 해야 하나요? (#how-do-i-view-rustdoc- documentation-for-a-library-my-project-depends-on)

카고 문서를 사용하여 자체 프로젝트에 대한 문서를 생성하면 활성 종속성 버전에 대한 문서도 생성됩니다. 이 문서는 프로젝트의 대상`/doc` 디렉터리에 저장됩니다. 사용 `cargo doc --open`을 사용하여 문서를 빌드한 후 열거나, 그냥 `target/doc/index.html`을 엽니다.
스스로.
