열거형 정의하기

코드를 작성할 때, 열거형이 구조체보다 유용하고 적절하게 사용되는 상황에 대해서 살펴볼 것입니다. IP 주소를 다뤄야 하는 경우를 생각해 봅시다. 현재 IP 주소에는 두 개의 주요한 표준이 있습니다: 버전 4와 버전 6입니다. 프로그램에서 다룰 IP 주소의 경우의 수는 이 두 가지가 전부입니다: 모든 가능한 값들을 *나열(enumerate)* 할 수 있으며, 이 경우를 열거라고 부를 수 있습니다.

IP 주소는 버전 4나 버전 6중 하나이며, 동시에 두 버전이 될 수는 없습니다. IP 주소의 속성을 보면 열거형 자료 구조가 적절합니다. 왜냐하면, 열거형의 값은 variants 중 하나만 될 수 있기 때문입니다. 버전 4나 버전 6은 근본적으로 IP 주소이기 때문에, 이 둘은 코드에서 모든 종류의 IP 주소에 적용되는 상황을 다룰 때 동일한 타입으로 처리되는 것이 좋습니다.

`IpAddrKind` 이라는 열거형을 정의하면서 포함할 수 있는 IP 주소인 `V4` 과 `V6` 를 나열함으로써 이 개념을 코드에 표현할 수 있습니다. 이것들은 열거형의 *variants* 라고 합니다:

```
enum IpAddrKind {
    V4,
    V6,
}
```

이제 `IpAddrKind` 은 우리의 코드 어디에서나 쓸 수 있는 데이터 타입이 되었습니다.

열거형 값

아래처럼 `IpAddrKind` 의 두 개의 variants 에 대한 인스턴스를 만들 수 있습니다:

```
let four = IpAddrKind::V4;
let six = IpAddrKind::V6;
```

열거형의 variants 는 열거형을 정의한 식별자에 의해 이름 공간이 생기며, 두 개의 콜론을 사용하여 둘을 구분할 수 있습니다. `IpAddrKind::V4` 와 `IpAddrKind::V6` 의 값은 동일한 타입이기 때문에, 이 방식이 유용합니다: `IpAddrKind`. 이제 `IpAddrKind` 타입을 인자로 받는 함수를 정의할 수 있습니다:

```
fn route(ip_type: IpAddrKind) { }
```

그리고, variant 중 하나를 사용해서 함수를 호출할 수 있습니다:

```
route(IpAddrKind::V4);
```

```
route(IpAddrKind::V6);
```

열거형을 사용하면 이점이 더 있습니다. IP 주소 타입에 대해 더 생각해 볼 때, 지금으로써는 실제 IP 주소 *데이터*를 저장할 방법이 없습니다. 단지 어떤 *종류* 인지만 알 뿐입니다. 5장에서 구조체에 대해 방금 공부했다고 한다면, 이 문제를 Listing 6-1에서 보이는 것처럼 풀려고 할 것입니다:

```
enum IpAddrKind {
    V4,
    V6,
}

struct IpAddr {
    kind: IpAddrKind,
    address: String,
}

let home = IpAddr {
    kind: IpAddrKind::V4,
    address: String::from("127.0.0.1"),
};

let loopback = IpAddr {
    kind: IpAddrKind::V6,
    address: String::from("::1"),
};
```

Listing 6-1: `struct` 를 사용해서 IP 주소의 데이터와 `IpAddrKind` variant 저장하기

여기서 두 개의 필드를 갖는 `IpAddr` 를 정의했습니다: `IpAddrKind` 타입(이전에 정의한 열거형)인 `kind` 필드와 `String` 타입인 `address` 필드입니다. 구조체에 대한 두 개의 인스턴스가 있습니다. 첫 번째 `home` 은 `kind` 의 값으로 `IpAddrKind::V4` 을 갖고 연관된 주소 데이터로 `127.0.0.1` 를 갖습니다. 두 번째 `loopback` 은 `IpAddrKind` 의 다른 variant 인 `V6` 을 값으로 갖고, 연관된 주소로 `::1` 를 갖습니다. `kind` 와 `address` 의 값을 함께 사용하기 위해 구조체를 사용했습니다. 그렇게 함으로써 variant 가 연관된 값을 갖게 되었습니다.

각 열거형 variant 에 데이터를 직접 넣는 방식을 사용해서 열거형을 구조체의 일부로 사용하는 방식보다 더 간결하게 동일한 개념을 표현할 수 있습니다. `IpAddr` 열거형의 새로운 정의에서는 두 개의 `V4` 와 `V6` variant 는 연관된 `String` 타입의 값을 갖게 됩니다.

```
enum IpAddr {
    V4(String),
    V6(String),
}

let home = IpAddr::V4(String::from("127.0.0.1"));

let loopback = IpAddr::V6(String::from("::1"));
```

열거형의 각 variant 에 직접 데이터를 붙임으로써, 구조체를 사용할 필요가 없어졌습니다.

구조체 보다 열거형을 사용할 때 다른 장점이 있습니다. 각 variant 는 다른 타입과 다른 양의 연관

구조체 또는 열거형을 사용할 때 다른 방법이 있습니다. 각 variant 는 다른 타입과 다른 양의 관련된 데이터를 가질 수 있습니다. 버전 4 타입의 IP 주소는 항상 0 ~ 255 사이의 숫자 4개로 된 구성요소를 갖게 될 것입니다. `V4` 주소에 4개의 `u8` 값을 저장하길 원하지만, V6 주소는 하나의 String 값으로 표현되길 원한다면, 구조체로는 이렇게 할 수 없습니다. 열거형은 이런 경우를 쉽게 처리합니다:

```
enum IpAddr {
    V4(u8, u8, u8, u8),
    V6(String),
}

let home = IpAddr::V4(127, 0, 0, 1);

let loopback = IpAddr::V6(String::from("::1"));
```

두 가지 다른 종류의 IP 주소를 저장하기 위해 코드상에서 열거형을 정의하는 몇 가지 방법을 살펴봤습니다. 그러나, 누구나 알듯이 IP 주소와 그 종류를 저장하는 것은 흔하기 때문에, 표준 라이브러리에 사용할 수 있는 정의가 있습니다!

표준 라이브러리에서 `IpAddr` 를 어떻게 정의하고 있는지 살펴봅시다.

위에서 정의하고 사용했던 것과 동일한 열거형과 variant 를 갖고 있지만, variant 에 포함된 주소 데이터는 두 가지 다른 구조체로 되어 있으며, 각 variant 마다 다르게 정의하고 있습니다:

```
struct Ipv4Addr {
    // details elided
}

struct Ipv6Addr {
    // details elided
}

enum IpAddr {
    V4(Ipv4Addr),
    V6(Ipv6Addr),
}
```

이 코드에서 보듯이 열거형 variant 에 어떤 종류의 데이터라도 넣을 수 있습니다: 예를 들면 문자열, 숫자 타입, 혹은 구조체. 다른 열거형 조차도 포함할 수 있습니다! 또한 표준 라이브러리 타입들은 어떤 경우에는 해결책으로 생각한 것보다 훨씬 더 복잡하지 않습니다.

현재 스코프에 표준 라이브러리를 가져오지 않았기 때문에, 표준 라이브러리에 `IpAddr` 정의가 있더라도, 동일한 이름의 타입을 만들고 사용할 수 있습니다. 타입을 가져오는 것에 대해서는 7장에서 더 살펴볼 것입니다.

Listing 6-2 에 있는 열거형의 다른 예제를 살펴봅시다: 이 예제에서는 각 variants 에 다양한 유형의 타입들이 포함되어 있습니다:

```
enum Message {
    Quit,
```

```
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
```

Listing 6-2: `Message` 열거형은 각 variants 가 다른 타입과 다른 양의 값을 저장함.

이 열거형에는 다른 데이터 타입을 갖는 네 개의 variants 가 있습니다:

- `Quit` 은 연관된 데이터가 전혀 없습니다.
- `Move` 은 익명 구조체를 포함합니다.
- `Write` 은 하나의 `String` 을 포함합니다.
- `ChangeColor` 는 세 개의 `i32` 을 포함합니다.

Listing 6-2 에서 처럼 variants 로 열거형을 정의하는 것은 다른 종류의 구조체들을 정의하는 것과 비슷합니다. 열거형과 다른 점은 `struct` 키워드를 사용하지 않는다는 것과 모든 variants 가 `Message` 타입으로 그룹화된다는 것입니다. 아래 구조체들은 이전 열거형의 variants 가 갖는 것과 동일한 데이터를 포함할 수 있습니다:

```
struct QuitMessage; // 유닛 구조체
struct MoveMessage {
    x: i32,
    y: i32,
}
struct WriteMessage(String); // 튜플 구조체
struct ChangeColorMessage(i32, i32, i32); // 튜플 구조체
```

각기 다른 타입을 갖는 여러 개의 구조체를 사용한다면, 이 메시지 중 어떤 한 가지를 인자로 받는 함수를 정의하기 힘들 것입니다. Listing 6-2 에 정의한 `Message` 열거형은 하나의 타입으로 이것이 가능합니다.

열거형과 구조체는 한 가지 더 유사한 점이 있습니다: 구조체에 `impl` 을 사용해서 메소드를 정의한 것처럼, 열거형에도 정의할 수 있습니다. 여기 `Message` 열거형에 에 정의한 `call` 이라는 메소드가 있습니다:

```
impl Message {
    fn call(&self) {
        // 메소드 내용은 여기 정의할 수 있습니다.
    }
}

let m = Message::Write(String::from("hello"));
m.call();
```

열거형의 값을 가져오기 위해 메소드 안에서 `self` 를 사용할 것입니다. 이 예제에서 생성한 변수 `m` 은 `Message::Write(String::from("hello"))` 값을 갖게 되고, 이 값은 `m.call()`이 실행될

때, `call` 메소드 안에서 `self`가 될 것입니다.

표준 라이브러리에 있는 매우 흔하게 사용하고 유용한 열거형을 살펴봅시다: `Option`.

`Option` 열거형과 Null 값 보다 좋은 점들.

이전 절에서, `IpAddr` 열거형을 사용하여 작성한 프로그램에서는 러스트 타입 시스템을 사용하여 데이터뿐만 아니라 더 많은 정보를 담을 수 있는 방법을 살펴보았습니다.

이번 절에서는 표준 라이브러리에서 열거형으로 정의된 또 다른 타입인 `Option`에 대한 사용 예를 살펴볼 것입니다. `Option` 타입은 많이 사용되는데, 값이 있거나 없을 수도 있는 아주 흔한 상황을 나타내기 때문입니다. 이 개념을 타입 시스템의 관점으로 표현하자면, 컴파일러가 발생할 수 있는 모든 경우를 처리했는지 체크할 수 있습니다. 이렇게 함으로써 버그를 방지할 수 있고, 이것은 다른 프로그래밍 언어에서 매우 흔합니다.

프로그래밍 언어 디자인은 가끔 어떤 특성들이 포함되었는지의 관점에서 생각되기도 하지만, 포함되지 않은 특성들도 역시 중요합니다. 러스트는 다른 언어들에서 흔하게 볼 수 있는 null 특성이 없습니다. *Null* 은 값이 없다는 것을 표현하는 하나의 값입니다. null 을 허용하는 언어에서는, 변수는 항상 두 상태중 하나가 될 수 있습니다: null 혹은 null 이 아님.

null 을 고안한 Tony Hoare 의 "Null 참조 : 10 억 달러의 실수"에서 다음과 같이 말합니다:

> 나는 그것을 나의 10억 달러의 실수라고 생각한다. 그 당시 객체지향 언어에서 처음 참조를 위한 포괄적인 타입 시스템을 디자인하고 있었다. 내 목표는 컴파일러에 의해 자동으로 수행되는 체크를 통해 모든 참조의 사용은 절대적으로 안전하다는 것을 확인하는 것이었다. 그러나 null 참조를 넣고 싶은 유혹을 참을 수 없었다. 간단한 이유는 구현이 쉽다는 것이었다. 이것은 수없이 많은 오류와 취약점들, 시스템 종료를 유발했고, 지난 40년간 10억 달러의 고통과 손실을 초래했을 수도 있다.

---

null 값으로 발생하는 문제는, null 값을 null 이 아닌 값처럼 사용하려고 할 때 여러 종류의 오류가 발생할 수 있다는 것입니다. null이나 null이 아닌 속성은 어디에나 있을 수 있고, 너무나도 쉽게 이런 종류의 오류를 만들어 냅니다.

그러나, null 이 표현하려고 하는 것은 아직까지도 유용합니다: null 은 현재 어떤 이유로 유효하지 않고, 존재하지 않는 하나의 값입니다.

문제는 실제 개념에 있기보다, 특정 구현에 있습니다. 이와 같이 러스트에는 null 이 없지만, 값의 존재 혹은 부재의 개념을 표현할 수 있는 열거형이 있습니다. 이 열거형은 `Option<T>` 이며, 다음과 같이 표준 라이브러리에 정의되어 있습니다:

```
enum Option<T> {
    Some(T),
```

```
    Some(T),
    None,
}
```

`Option<T>` 열거형은 매우 유용하며 기본적으로 포함되어 있기 때문에, 명시적으로 가져오지 않아도 사용할 수 있습니다. 또한 variants 도 마찬가지입니다: `Option::` 를 앞에 붙이지 않고, `Some` 과 `None` 을 바로 사용할 수 있습니다. `Option<T>` 는 여전히 일반적인 열거형이고, `Some(T)` 과 `None` 도 여전히 `Option<T>` 의 variants 입니다.

`<T>` 는 러스트의 문법이며 아직 다루지 않았습니다. 제너릭 타입 파라미터이며, 제너릭에 대해서는 10 장에서 더 자세히 다룰 것입니다. 지금은 단지 `<T>` 가 `Option` 열거형의 `Some` variant 가 어떤 타입의 데이터라도 가질 수 있다는 것을 의미한다는 것을 알고 있으면 됩니다. 여기 숫자 타입과 문자열 타입을 갖는 `Option` 값에 대한 예들이 있습니다:

```
let some_number = Some(5);
let some_string = Some("a string");

let absent_number: Option<i32> = None;
```

`Some` 이 아닌 `None` 을 사용한다면, `Option<T>` 이 어떤 타입을 가질지 러스트에게 알려줄 필요가 있습니다. 컴파일러는 `None` 만 보고는 `Some` variant 가 어떤 타입인지 추론할 수 없습니다.

`Some` 값을 얻게 되면, 값이 있다는 것과 `Some` 이 갖고 있는 값에 대해 알 수 있습니다. `None` 값을 사용하면, 어떤 면에서는 null 과 같은 의미를 갖게 됩니다: 유효한 값을 갖지 않습니다. 그렇다면 왜 `Option<T>` 가 null 을 갖는 것보다 나을까요?

간단하게 말하면, `Option<T>` 와 `T` (`T` 는 어떤 타입이던 될 수 있음)는 다른 타입이며, 컴파일러는 `Option<T>` 값을 명확하게 유효한 값처럼 사용하지 못하도록 합니다. 예를 들면, 아래 코드는 `Option<i8>` 에 `i8` 을 더하려고 하기 때문에 컴파일되지 않습니다:

```
let x: i8 = 5;
let y: Option<i8> = Some(5);

let sum = x + y;
```

이 코드를 실행하면, 아래와 같은 에러 메시지가 출력됩니다:

```
error[E0277]: the trait bound `i8: std::ops::Add<std::option::Option<i8>>` is
not satisfied
 -->
  |
7 | let sum = x + y;
  |           ^^^^^
  |
```

주목하세요! 실제로, 이 에러 메시지는 러스트가 `Option<i8>` 와 `i8` 를 어떻게 더해야 하는지 모른다는 것을 의미하는데, 둘은 다른 타입이기 때문입니다. 러스트에서 `i8` 과 같은 타입의 값을 가

질 때, 컴파일러는 항상 유효한 값을 갖고 있다는 것을 보장할 것입니다. 값을 사용하기 전에 null 인지 확인할 필요도 없이 자신 있게 사용할 수 있습니다. 단지 `Option<i8>` 을 사용할 경우엔 (혹은 어떤 타입 이건 간에) 값이 있을지 없을지에 대해 걱정할 필요가 있으며, 컴파일러는 값을 사용하기 전에 이런 케이스가 처리되었는지 확인해 줄 것입니다.

다르게 얘기하자면, `T` 에 대한 연산을 수행하기 전에 `Option<T>` 를 `T` 로 변환해야 합니다. 일반적으로, 이런 방식은 null 과 관련된 가장 흔한 이슈 중 하나를 발견하는데 도움을 줍니다: 실제로 null 일 때, null 이 아니라고 가정하는 경우입니다.

null 이 아닌 값을 갖는다는 가정을 놓치는 경우에 대해 걱정할 필요가 없게 되면, 코드에 더 확신을 갖게 됩니다. null 일 수 있는 값을 사용하기 위해서, 명시적으로 값의 타입을 `Option<T>` 로 만들어 줘야 합니다. 그다음엔 값을 사용할 때 명시적으로 null 인 경우를 처리해야 합니다. 값의 타입이 `Option<T>` 가 아닌 모든 곳은 값이 null 아 아니라고 안전하게 *가정할 수 있습니다*. 이것은 null을 너무 많이 사용하는 문제를 제한하고 러스트 코드의 안정성을 높이기 위한 러스트의 의도된 디자인 결정사항입니다.

그럼 `Option<T>` 타입인 값을 사용할 때, `Some` variant 에서 `T` 값을 어떻게 가져와서 사용할 수 있을까요? `Option<T>` 열거형에서 다양한 상황에서 유용하게 사용할 수 있는 많은 메소드들이 있습니다; 문서에서 확인할 수 있습니다. `Option<T>` 의 메소드들에 익숙해지는 것은 러스트를 사용하는데 매우 유용할 것입니다.

일반적으로, `Option<T>` 값을 사용하기 위해서는 각 variant 를 처리할 코드가 필요할 것입니다. `Some(T)` 값일 경우만 실행되는 코드가 필요하고, 이 코드는 안에 있는 `T` 를 사용할 수 있습니다. 다른 코드에서는 `None` 값일 때 실행되는 코드가 필요가 하기도 하며, 이 코드에서는 사용할 수 있는 `T` 값이 없습니다. `match` 표현식은 제어 흐름을 위한 구분으로, 열거형과 함께 사용하면 이런 일들을 할 수 있습니다: 열거형이 갖는 variant 에 따라 다른 코드를 실행할 것이고, 그 코드는 매칭된 값에 있는 데이터를 사용할 수 있습니다.