



GDG Golang Korea, FEB 25, 2021

Go generic

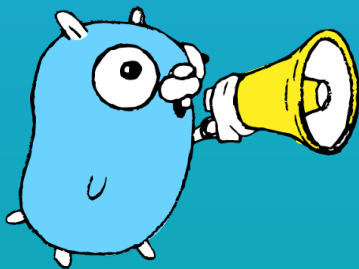
왜 없었고, 왜 생기고, 무엇이 달라질까?



Hyeongjun Kim

Naver Search

hjkim2246@gmail.com



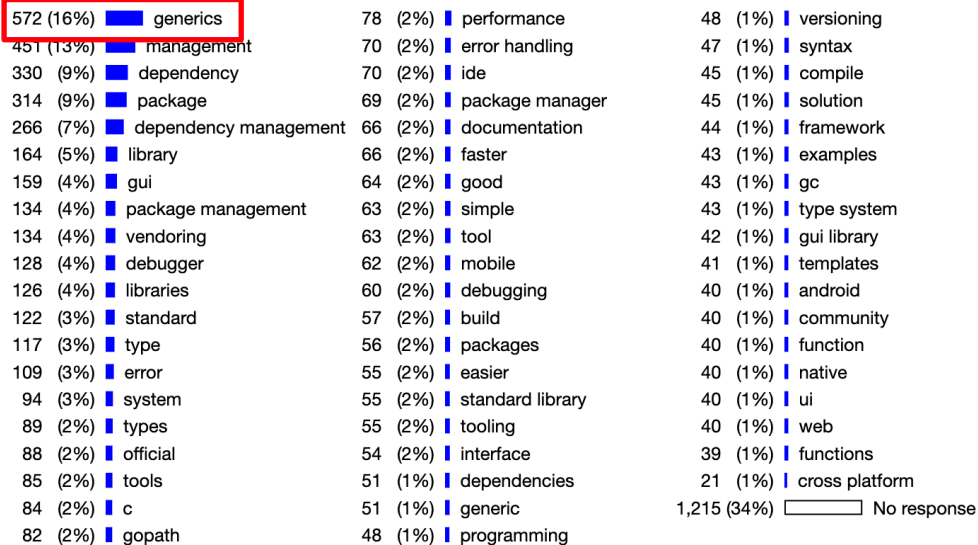
Intro	01
Generic Programming	02
Why there's no generic in go	03
How to replace generic in go	04
What could be different?	05
Reference	06

go generic은 수많은 gopher들의 오랜 소망이었습니다.



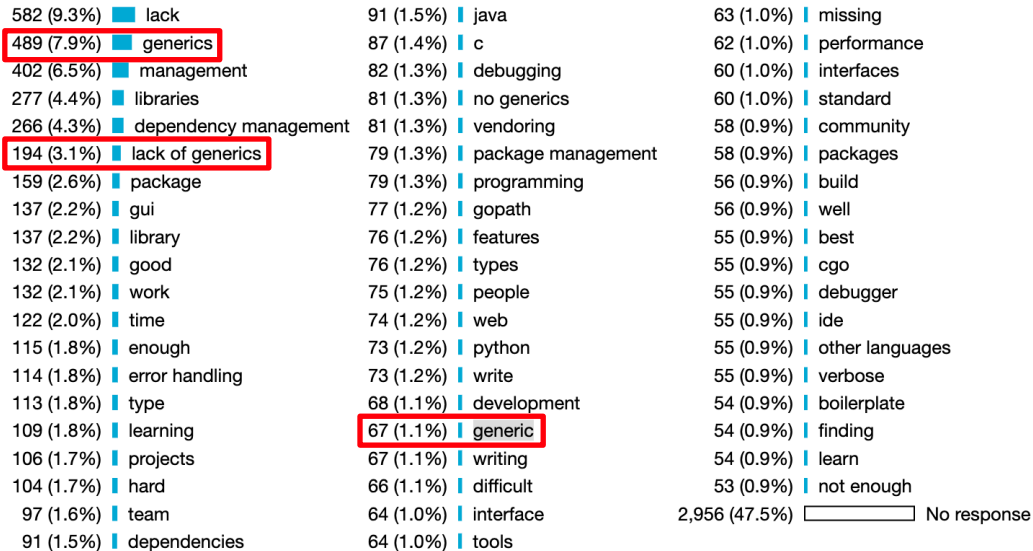
go generic은 수많은 gopher들의 오랜 소망이었습니다.

What changes would improve Go most?



go generic은 수많은 gopher들의 오랜 소망이었습니다.

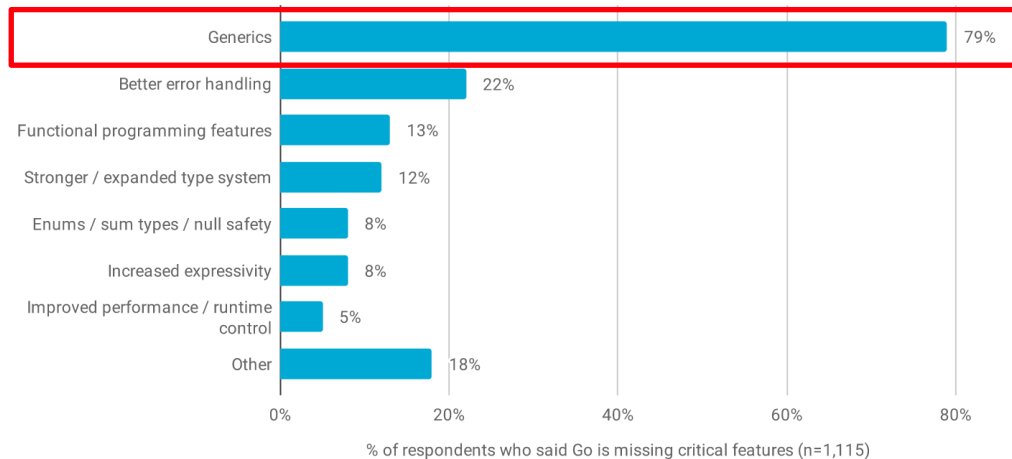
What is the biggest challenge you personally face using Go today?



go generic은 수많은 gopher들의 오랜 소망이었습니다.

Which critical language features do you need that are not available in Go?

(free-text response)



그리고 2021년 1월 13일에 정식 proposal이 올라왔고

spec: add generic programming using type parameters #43651



Open

ianlancetaylor opened this issue on 13 Jan · 441 comments



ianlancetaylor commented on 13 Jan · edited ▾

Contributor



We propose adding support for type parameters to Go. This will change the Go language to support a form of generic programming.

(이렇게 만들면 괜찮을거 같아!)

그리고 2021년 1월 13일에 정식 proposal이 올라왔고

이게 accept되면서 구현에 들어가 빠르면 1.18 beta 버전에 추가될 예정입니다!

spec: add generic programming using type parameters #43651

Open

ianlancetaylor opened this issue on 13 Jan · 441 comments



ianlancetaylor commented on 13 Jan · edited

Contributor



We propose adding support for type parameters to Go. This will change the Go language to support a form of generic programming.

(이렇게 만들면 괜찮을거 같아!)



rsc commented 5 days ago

Contributor



No change in consensus, so **accepted**. 🎉

This issue now tracks the work of implementing the proposal.

— rsc for the proposal review group



836



9



483



6



259



269



88

(콜!)

<https://github.com/golang/go/issues/43651>



오늘 이 자리에서는

오늘 이 자리에서는

Generic 프로그래밍이 무엇인지!

오늘 이 자리에서는

Generic 프로그래밍이 무엇인지!

왜 그토록 원했는지!

오늘 이 자리에서는

Generic 프로그래밍이 무엇인지!

왜 그토록 원했는지!

앞으로 뭐가 달라지는지! 다뤄보려 합니다.

Wikipedia

Generic programming is a **style of computer programming** in which algorithms are written in terms of **types** to-be-specified-later that are then **instantiated when needed for specific types provided as parameters**.

(타입을 파라미터로 보내서 인스턴스화 시점에 타입을 결정하는 프로그래밍 기법)

Wikipedia 한글

제네릭 프로그래밍은 데이터 형식에 의존하지 않고, 하나의 값이 여러 다른 데이터 타입들을 가질 수 있는 기술에 중점을 두어 재사용성을 높일 수 있는 프로그래밍 방식이다.

Wikipedia

Generic programming is a **style of computer programming** in which algorithms are written in terms of **types** to-be-specified-later that are then **instantiated when needed for specific types provided as parameters**.

(타입을 파라미터로 보내서 인스턴스화 시점에 타입을 결정하는 프로그래밍 기법)

Wikipedia 한글

제네릭 프로그래밍은 데이터 형식에 의존하지 않고, 하나의 값이 여러 다른 데이터 타입들을 가질 수 있는 기술에 중점을 두어 **재사용성을 높일 수 있는 프로그래밍 방식**이다.

타입을 파라미터로 보내서 인스턴스화 시점에 변수의 타입을 결정한다?



Java에서 generic을 사용하여 인스턴스화 시점에 변수의 타입을 결정하는 예제

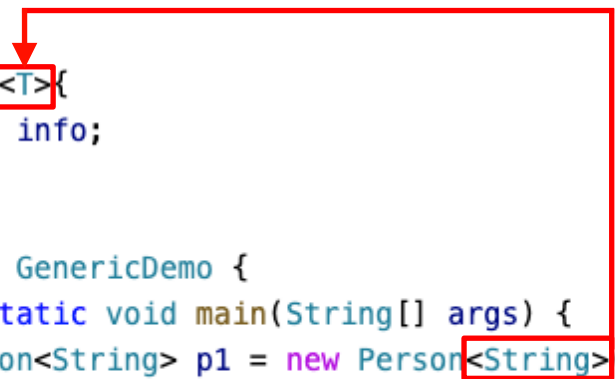
```
class Person<T>{  
    public T info;  
}  
  
public class GenericDemo {  
    public static void main(String[] args) {  
        Person<String> p1 = new Person<String>();  
        Person<StringBuilder> p2 = new Person<StringBuilder>();  
    }  
}
```

Java에서 generic을 사용하여 인스턴스화 시점에 변수의 타입을 결정하는 예제

```
class Person<T>{  
    public T info; 아직 타입이 결정되지 않은 변수  
}  
  
public class GenericDemo {  
    public static void main(String[] args) {  
        Person<String> p1 = new Person<String>();  
        Person<StringBuilder> p2 = new Person<StringBuilder>();  
    }  
}
```

Java에서 generic을 사용하여 인스턴스화 시점에 변수의 타입을 결정하는 예제

```
class Person<T>{  
    public T info;  
}  
  
public class GenericDemo {  
    public static void main(String[] args) {  
        Person<String> p1 = new Person<String>();  
        Person<StringBuilder> p2 = new Person<StringBuilder>();  
    }  
}
```



Java에서 generic을 사용하여 인스턴스화 시점에 변수의 타입을 결정하는 예제

```
class Person<T>{  
    public T info;  
}  
    타입 구체화  
  
public class GenericDemo {  
    public static void main(String[] args) {  
        Person<String> p1 = new Person<String>();  
        Person<StringBuilder> p2 = new Person<StringBuilder>();  
    }  
}
```

Java에서 generic을 사용하여 인스턴스화 시점에 변수의 타입을 결정하는 예제

```
class Person<T>{  
    public T info;  
}  
    타입 구체화  
  
public class GenericDemo {  
    public static void main(String[] args) {  
        Person<String> p1 = new Person<String>();  
        Person<StringBuilder> p2 = new Person<StringBuilder>();  
    }  
}
```

오호! 근데 generic을 쓰면 왜 코드의 재사용성이 좋아질까요?



golang에서 slice를 뒤집는 알고리즘을 작성한다고 가정해보겠습니다.

```
func ReverseInts(s []int) {  
    first := 0  
    last := len(s) - 1  
    for first < last {  
        s[first], s[last] = s[last], s[first]  
        first++  
        last--  
    }  
}
```

로직은 동일하지만, 다른 데이터 타입 슬라이스를 파라미터로 받기 위해서
함수를 새로 만들어야 하는 반복이 발생합니다.

```
func ReverseInts(s []int) {  
    first := 0  
    last := len(s) - 1  
    for first < last {  
        s[first], s[last] = s[last], s[first]  
        first++  
        last--  
    }  
}
```

```
func ReverseStr(s []string) {  
    first := 0  
    last := len(s) - 1  
    for first < last {  
        s[first], s[last] = s[last], s[first]  
        first++  
        last--  
    }  
}
```


generic을 사용하면 데이터 타입에 상관없이 적용할 수 있는 로직을 작성할 수 있어
코드의 재사용성이 좋아집니다.

type을 파라미터로 전달

```
func Reverse (type Elemet) (s []Element) {  
    first := 0  
    last := len(s) - 1  
    for first < last {  
        s[first], s[last] = s[last], s[first]  
        first++  
        last--  
    }  
}
```

generic을 사용하면 어떤 점들이 좋은가요?

generic을 사용하면 어떤 점들이 좋은가요?

1. 컴파일 시점에 타입 체크를 더 강력하게 할 수 있다.

generic을 통한 컴파일 타임에서 타입 체크 강화

```
1  abstract class Info{
2      public abstract int getLevel();
3  }
4  class EmployeeInfo extends Info{
5      public int rank;
6      EmployeeInfo(int rank){ this.rank = rank; }
7      public int getLevel(){
8          return this.rank;
9      }
10 }
11 class Person<T extends Info>{
12     public T info;
13     Person(T info){ this.info = info; }
14 }
15 public class GenericDemo {
16     public static void main(String[] args) {
17         Person p1 = new Person(new EmployeeInfo(1));
18         Person<String> p2 = new Person<String>("부장");
19     }
20 }
```

generic을 통한 컴파일 타임에서 타입 체크 강화

```
1  abstract class Info{
2      public abstract int getLevel();
3  }
4  class EmployeeInfo extends Info{
5      public int rank;
6      EmployeeInfo(int rank){ this.rank = rank; }
7      public int getLevel(){
8          return this.rank;
9      }
10 }
11 class Person<T extends Info>{           파라미터로 전달받을 타입에 제약 조건 부여
12     public T info;
13     Person(T info){ this.info = info; }
14 }
15 public class GenericDemo {
16     public static void main(String[] args) {
17         Person p1 = new Person(new EmployeeInfo(1));
18         Person<String> p2 = new Person<String>("부장");
19     }
20 }
```

<https://www.opentutorials.org/course/1223/6237>

generic을 통한 컴파일 타임에서 타입 체크 강화

```
1 abstract class Info{
2     public abstract int getLevel();
3 }
4 class EmployeeInfo extends Info{
5     public int rank;
6     EmployeeInfo(int rank){ this.rank = rank; }
7     public int getLevel(){
8         return this.rank;
9     }
10 }
11 class Person<T extends Info>{
12     public T info;
13     Person(T info){ this.info = info; }
14 }
15 public class GenericDemo {
16     public static void main(String[] args) {
17         Person p1 = new Person(new EmployeeInfo(1));
18         Person<String> p2 = new Person<String>("부장");
19     }
20 }
```

getLevel 함수를 구현하라는 제약

파라미터로 전달받을 타입에 제약 조건 부여

<https://www.opentutorials.org/course/1223/6237>

generic을 통한 컴파일 타임에서 타입 체크 강화

```
1  abstract class Info{
2      public abstract int getLevel();
3  }
4  class EmployeeInfo extends Info{
5      public int rank;
6      EmployeeInfo(int rank){ this.rank = rank; }
7      public int getLevel(){
8          return this.rank;
9      }
10 }
11 class Person<T extends Info>{
12     public T info;
13     Person(T info){ this.info = info; }
14 }
15 public class GenericDemo {
16     public static void main(String[] args) {
17         Person p1 = new Person(new EmployeeInfo(1));
18         Person<String> p2 = new Person<String>("부장");
19     }
20 }
```

정상 동작

generic을 통한 컴파일 타임에서 타입 체크 강화

```
1  abstract class Info{
2      public abstract int getLevel();
3  }
4  class EmployeeInfo extends Info{
5      public int rank;
6      EmployeeInfo(int rank){ this.rank = rank; }
7      public int getLevel(){
8          return this.rank;
9      }
10 }
11 class Person<T extends Info>{
12     public T info;
13     Person(T info){ this.info = info; }
14 }
15 public class GenericDemo {
16     public static void main(String[] args) {
17         Person p1 = new Person(new EmployeeInfo(1));
18         Person<String> p2 = new Person<String>("부장");
19     }
20 }
```

컴파일 에러 발생!

<https://www.opentutorials.org/course/1223/6237>

generic을 사용하면 어떤 점들이 좋은가요?

1. 컴파일 시점에 타입 체크를 더 강력하게 할 수 있다.
2. 타입 캐스팅을 제거할 수 있다.

generic을 통한 타입 캐스팅 제거

```
1 public class GenericDemo {
2     public static void main(String[] args) {
3         List list = new ArrayList(); 타입을 명시해주지 않으면 Object 클래스로 객체를 수신
4         list.add("hello");
5         String s = (String) list.get(0); 객체를 읽어올 때, 타입 캐스팅이 필요
6
7         List<String> list = new ArrayList<String>();
8         list.add("hello");
9         String s = list.get(0);
10    }
11 }
```

<https://docs.oracle.com/javase/tutorial/java/generics/why.html>

generic을 통한 타입 캐스팅 제거

```
1 public class GenericDemo {  
2     public static void main(String[] args) {  
3         List list = new ArrayList();  
4         list.add("hello");  
5         String s = (String) list.get(0);  
6  
7         List<String> list = new ArrayList<String>(); 타입을 미리 명시  
8         list.add("hello");  
9         String s = list.get(0); 타입 캐스팅 제거 가능  
10    }  
11 }
```

generic을 사용하면 어떤 점들이 좋은가요?

1. 컴파일 시점에 타입 체크를 더 강력하게 할 수 있다.
2. 타입 캐스팅을 제거할 수 있다.
3. 프로그래머가 generic algorithm을 짤 수 있도록 도와준다.

generic algorithm 예시

Comparable 인터페이스가 구현되어 있으면 Sort 적용이 가능!

```
public class SortedTree<E implements Comparable<E>> {  
    void insert(E comparableSortTreeElement) {  
        //...  
    }  
}
```

```
public static void main(String[] args) {  
    SortedTree<String> sortedTreeOfStrings = new SortedTree<String>();  
    sortedTreeOfStrings.insert("abcde");  
}
```

인스턴스화 시점에 타입 전달!

이렇게 좋은데 왜 고랭에는 없었죠?



Go FAQ: Why does Go not have generic types?

Go was intended as a language for writing server programs that would be easy to maintain over time. Polymorphic programming did not seem essential to the language's goals at the time, and so was left out for simplicity.

Generics are convenient but they come at a cost in complexity in the type system and run-time. We haven't yet found a design that gives value proportionate to the complexity, although we continue to think about it. Meanwhile, Go's built-in maps and slices, plus the ability to use the empty interface to construct containers (with explicit unboxing) mean in many cases it is possible to write code that does what generics would enable, if less smoothly.

Go FAQ: Why does Go not have generic types?

- golang은 시간이 지나도 유지보수 하기 쉽도록 설계하였다.
- 이 관점에서 다형성은 필수적인 요소로 보이지 않아 단순함을 위해 배제하였다.
- Generic이 편리한 것은 사실이지만, 이는 타입 체계와 런 타임의 복잡성을 수반한다.
- generic이 없는 대신, built-in maps, slice, interface{}을 이용하여 generic의 효과를 낼 수 있다.

Go FAQ: Why does Go not have generic types?

- golang은 시간이 지나도 유지보수 하기 쉽도록 설계하였다.
- 이 관점에서 다형성을 필수적인 요소로 보이지 않아 다수함을 위해 배제하였다.
복잡해서 뺐다! 근데 없어도 비슷하게 구현할 수 있다!
- Generic이 편리한 것은 사실이지만, 이는 타입 체계와 런 타임의 복잡성을 수반한다.
- generic이 없는 대신, built-in maps, slice, interface{}을 이용하여 generic의 효과를 낼 수 있다.

Why does Go not have generic types?

각 언어별 generic 구현 방식과 부작용 (by Russ Cox)

- C: 뺐다!

→ **slow programmers!**

- C++: Compile-time specialization, macro expansion

→ **slow compile time!**

- Java: 모든 것을 추상화 시켰다.

→ **slow execution time!**

Why does Go not have generic types?

각 언어별 generic 구현 방식과 부작용 (by Russ Cox)

- C: 뺐다!

→ slow programmers!

결국 선택의 문제!

- C++: Compile-time specialization, macro expansion

→ slow compile time!

제네릭의 딜레마!

- Java: 모든 것을 추상화 시켰다.

→ slow execution time!

<https://research.swtch.com/generic>

그동안 go 프로젝트에서 generic이 필요할 때 어떻게 했는가?

그동안 go 프로젝트에서 generic이 필요할 때 어떻게 했는가?

1. 진짜로 generic이 필요한지 요구사항을 검토한다.



그동안 go 프로젝트에서 generic이 필요할 때 어떻게 했는가?

1. 진짜로 generic이 필요한지 요구사항을 검토한다.
2. 코드 복붙으로 해결할 수 있는지 검토한다.

```
func ReverseInts(s []int) {  
    ...  
}  
  
func ReverseStr(s []string) {  
    ...  
}
```

같은 함수, 다른 파라미터!

그동안 go 프로젝트에서 generic이 필요할 때 어떻게 했는가?

1. 진짜로 generic이 필요한지 요구사항을 검토한다.
2. 코드 복붙으로 해결할 수 있는지 검토한다.
3. 인터페이스로 해결한다.

Go sort package 소스코드

```
type Interface interface {  
    Len() int  
    Less(i, j int) bool  
    Swap(i, j int)  
}
```

Sorting에 필요한 공통 요소들 인터페이스화

```
func Sort(data Interface) {  
    n := data.Len()  
    quickSort(data, 0, n, maxDepth(n))  
}
```

파라미터로 interface 전달

Go sort package 소스코드

```
type Interface interface {  
    Len() int  
    Less(i, j int) bool  
    Swap(i, j int)  
}
```

이상적인 방법!

Sorting에 필요한 공통 요소들 인터페이스화

그러나 추가적인 코딩이 필요하고, 추상화 구조가 복잡해질 수록 한계가 있다!

```
func Sort(data Interface) {  
    n := data.Len()  
    quickSort(data, 0, n, maxDepth(n))  
}
```

파라미터로 interface 전달

그동안 go 프로젝트에서 generic이 필요할 때 어떻게 했는가?

1. 진짜로 generic이 필요한지 요구사항을 검토한다.
2. 코드 복붙으로 해결할 수 있는지 검토한다.
3. 인터페이스로 해결한다.
4. type assertion으로 해결한다.

How to replace generic?

```
type Container []interface{}
```

컨테이너는 빈 인터페이스의 슬라이스

```
type (c *Container) Put(elem interface{}) {  
    *c = append(*c, elem)  
}
```

```
func (c *Container) Get() interface{} {  
    elem := (*c)[0]  
    *c = (*c)[1:]  
    return elem  
}
```

```
func assertExample() {  
    intContainer := &Container{}  
    intContainer.Put(7)  
    intContainer.Put(42)  
    elem, ok := intContainer.Get().(int)  
    if !ok {  
        fmt.Println("Unable to read an int from intContainer")  
    }  
    fmt.Printf("assertExample: %d (%T)\n", elem, elem)  
}
```

런 타임에 자료형 변환을 수행하여 타입 검사

How to replace generic?

```
type Container []interface{}
```

컨테이너는 빈 인터페이스의 슬라이스

```
type (c *Container) Put(elem interface{}) {  
    *c = append(*c, elem)  
}
```

```
func (c *Container) Get() interface{} {  
    elem := (*c)[0]  
    *c = (*c)[1:]  
    return elem  
}
```

컴파일 시점에서 타입 체크가 불가, 런 타임 에러의 위험 소지!

```
intContainer := &Container{}  
intContainer.Put(7)  
intContainer.Put(42)  
elem, ok := intContainer.Get().(int)  
if !ok {  
    fmt.Println("Unable to read an int from intContainer")  
}  
fmt.Printf("assertExample: %d (%T)\n", elem, elem)
```

런 타임에 자료형 변환을 수행하여 타입 검사

그동안 go 프로젝트에서 generic이 필요할 때 어떻게 했는가?

1. 진짜로 generic이 필요한지 요구사항을 검토한다.
2. 코드 복붙으로 해결할 수 있는지 검토한다.
3. 인터페이스로 해결한다.
4. type assertion으로 해결한다.
5. reflection으로 해결한다.

(런 타임에 인터페이스나 구조체 등의 타입 정보를 얻어내거나 결정하는 기능)

How to replace generic?

```
type Cabinet struct {  
    s reflect.Value  
}  
  
    런 타임에 결정되는 타입  
  
func NewCabinet(t reflect.Type) *Cabinet {  
    return &Cabinet{  
        s: reflect.MakeSlice(reflect.SliceOf(t), 0, 10)  
    }  
}  
  
func (c *Cabinet) Put(val interface{}) {  
    if reflect.ValueOf(val).Type() != c.s.Type().Elem() {  
        panic(fmt.Sprintf("Put: cannot put a %T into a slice of %s",  
            val, c.s.Type().Elem()))  
    }  
    c.s = reflect.Append(c.s, reflect.ValueOf(val))  
}  
  
func (c *Cabinet) Get(retref interface{}) {  
    retref = c.s.Index(0)  
    c.s = c.s.Slice(1, c.s.Len())  
}
```

```
func reflectExample() {  
    f := 3.14152  
    g := 0.0  
    c := NewCabinet(reflect.TypeOf(f)) 런 타임에 타입을 가져와 자료구조 생성  
    c.Put(f)  
  
    fmt.Println(c.s.Index(0))  
    c.Get(&g)  
    fmt.Printf("reflectExample: %f (%T)\n", g, g)  
}
```

How to replace generic?

```
type Cabinet struct {
    s reflect.Value
}

func NewCabinet(t reflect.Type) *Cabinet {
    return &Cabinet{
        s: reflect.MakeSlice(reflect.SliceOf(t), 0, 10),
    }
}

func (c *Cabinet) Put(val interface{}) {
    if reflect.ValueOf(val).Type() != c.s.Type().Elem() {
        panic(fmt.Sprintf("Put: cannot put a %T into a slice of %s",
            val, c.s.Type().Elem()))
    }
    c.s = reflect.Append(c.s, reflect.ValueOf(val))
}

func (c *Cabinet) Get(retref interface{}) {
    retref = c.s.Index(0)
    c.s = c.s.Slice(1, c.s.Len())
}
```

```
func reflectExample() {
    f := 3.14152
    g := 0.0
    c := NewCabinet(reflect.TypeOf(f))
    c.Put(f)

    fmt.Println(c.s.Index(0))
    c.Get(&g)
    fmt.Printf("reflectExample: %f (%T)\n", g, g)
}
```

런 타임에 타입 검사 실행

How to replace generic?

```
type Cabinet struct {  
    s reflect.Value  
}  
  
func NewCabinet(t reflect.Type) *Cabinet {  
    return &Cabinet{  
        s: reflect.MakeSlice(reflect.SliceOf(t), 0, 10),  
    }  
}
```

```
func reflectExample() {  
    f := 3.14152  
    g := 0.0  
    c := NewCabinet(reflect.TypeOf(f))  
    c.Put(f)  
  
    fmt.Println(c.s.Index(0))  
    c.Get(&g)  
    fmt.Printf("reflectExample: %f (%T)\n", g, g)  
}
```

```
func (c *Cabinet) Put(val interface{}) {  
    if reflect.ValueOf(val).Type() != c.s.Type().Elem() {  
        panic(fmt.Sprintf("Put: cannot put a %T into a slice of %s",  
            val, c.s.Type().Elem()))  
    }  
    c.s = reflect.Append(c.s, reflect.ValueOf(val))  
}
```

코드의 복잡성 증가,

런 타임에 잦은 타입 검사로 인한 성능 저하 유발!

```
func (c *Cabinet) Get(retref interface{}) {  
    retref = c.s.Index(0)  
    c.s = c.s.Slice(1, c.s.Len())  
}
```


그동안 go 프로젝트에서 generic이 필요할 때 어떻게 했는가?

1. 진짜로 generic이 필요한지 요구사항을 검토한다.
2. 코드 복붙으로 해결할 수 있는지 검토한다.
3. 인터페이스로 해결한다.
4. type assertion으로 해결한다.
5. reflection으로 해결한다.
6. Code generator로 해결한다.

How to replace generic?

```
package capsule
```

```
import "github.com/cheekybits/genny/generic"
```

 코드 제너레이터 패키지

```
type Item generic.Type
```

 코드 제너레이터가 인식하는 타입

```
type ItemCapsule struct {  
    s []Item  
}
```

```
func NewItemCapsule() *ItemCapsule {  
    return &ItemCapsule{s: []Item{}}  
}
```

```
func (c *ItemCapsule) Put(val Item) {  
    c.s = append(c.s, val)  
}
```

```
func (c *ItemCapsule) Get() Item {  
    r := c.s[0]  
    c.s = c.s[1:]  
    return r  
}
```

<https://appliedgo.net/generics/>

How to replace generic?

```
package capsule

import "github.com/cheekybits/genny/generic"

type Item generic.Type

type ItemCapsule struct {
    s []Item
}

func NewItemCapsule() *ItemCapsule {
    return &ItemCapsule{s: []Item{}}
}

func (c *ItemCapsule) Put(val Item) {
    c.s = append(c.s, val)
}

func (c *ItemCapsule) Get() Item {
    r := c.s[0]
    c.s = c.s[1:]
    return r
}
```



컴파일 전에
code generator 실행

```
package capsule

import "github.com/cheekybits/genny/generic"

type Uint32Capsule struct {
    s []uint32
}

func NewUint32Capsule() *Uint32Capsule {
    return &Uint32Capsule{s: []uint32{}}
}

func (c *Uint32Capsule) Put(val uint32) {
    c.s = append(c.s, val)
}

func (c *Uint32Capsule) Get() uint32 {
    r := c.s[0]
    c.s = c.s[1:]
    return r
}

func generateExample() {
    var u uint32 = 42
    c := NewUint32Capsule()
    c.Put(u)
    v := c.Get()
    fmt.Printf("generateExample: %d (%T)\n", v, v)
}
```

<https://appliedgo.net/generics/>

How to replace generic?

```
package capsule

import "github.com/cheekybits/genny/generic"

type Item generic.Type

type ItemCapsule struct {
    s []Item
}
```

```
func NewItemCapsule() *ItemCapsule {
    return &ItemCapsule{s: []Item{}}
```

```
func (c *ItemCapsule) Put(val Item) {
    c.s = append(c.s, val)
}
```

```
func (c *ItemCapsule) Get() Item {
    r := c.s[0]
    c.s = c.s[1:]
    return r
}
```

```
package capsule

import "github.com/cheekybits/genny/generic"

type Uint32Capsule struct {
    s []uint32
}

func NewUint32Capsule() *Uint32Capsule {
    return &Uint32Capsule{s: []uint32{}}
```

```
func (c *Uint32Capsule) Put(val uint32) {
    c.s = append(c.s, val)
}
```

```
func (c *Uint32Capsule) Get() uint32 {
    r := c.s[0]
    c.s = c.s[1:]
    return r
}
```

```
func generateExample() {
    var u uint32 = 42
    c := NewUint32Capsule()
    c.Put(u)
    v := c.Get()
    fmt.Printf("generateExample: %d (%T)\n", v, v)
}
```

컴파일 시간을 느리게 한다!

컴파일 전에
빌드 의존성이 늘어난다!
code generator 실행

<https://appliedgo.net/generics/>

어떤 방법으로도 깔끔하게 generic을 지원하지 못한다.

그랬기 때문에 수 많은 gopher들이 generic을 원했고, 추가하게 되었다!

개발자들은 아래와 같은 원칙을 지키며 go에서 generic을 지원하는 방안을 제안

- 언어에 새로운 개념 추가를 최소화 (ex. syntax, keyword)
- 사용자가 쉽게 쓸 수 있어야 한다. 복잡한 작업은 generic package 개발자들이 담당
- 빌드 타임은 짧게! 실행 속도는 빠르게!
- Go 언어의 명확성과 심플함을 유지

“Generic can bring a significant benefit to the language.

But they are only worth of doing if go still feels like go”

Ian Lance Tylor, go generic proposal writer, google

What could be different?

generic이 추가된 golang은 어떻게 달라질까요?



generic이 추가된 golang은 어떻게 달라질까요?

1. generic을 이용하여 함수들을 심플하게 구현할 수 있습니다.
2. generic을 이용하여 자료구조들을 심플하게 구현할 수 있습니다.

What could be different?

generic 타입 파라미터 수신 함수 호출 예시

```
func Reverse (type Element) (s []Element) {  
    first := 0  
    last := len(s) - 1  
    for first < last {  
        s[first], s[last] = s[last], s[first]  
        first++  
        last--  
    }  
}  
  
func ReverseAndPrint(s []int) {  
    fmt.Println(Reverse(int)(s))  
    fmt.Println(Reverse(s))  
}
```

<https://blog.golang.org/why-generics>

What could be different?

generic 타입 파라미터 수신 함수 호출 예시

type 파라미터를 수신

```
func Reverse (type Element) (s []Element) {  
    first := 0  
    last := len(s) - 1  
    for first < last {  
        s[first], s[last] = s[last], s[first]  
        first++  
        last--  
    }  
}  
  
func ReverseAndPrint(s []int) {  
    fmt.Println(Reverse(int)(s))  
    fmt.Println(Reverse(s))  
}
```

<https://blog.golang.org/why-generics>

What could be different?

generic 타입 파라미터 수신 함수 호출 예시

type 파라미터를 수신

```
func Reverse (type Element) (s []Element) {  
    first := 0  
    last := len(s) - 1  
    for first < last {  
        s[first], s[last] = s[last], s[first]  
        first++  
        last--  
    }  
}
```

```
func ReverseAndPrint(s []int) {  
    fmt.Println(Reverse(int)(s))  
    fmt.Println(Reverse(s))  
}
```

type 파라미터를 별도로 전달하여 함수 호출

<https://blog.golang.org/why-generics>

What could be different?

generic 타입 파라미터 수신 함수 호출 예시

type 파라미터를 수신

```
func Reverse (type Element) (s []Element) {  
    first := 0  
    last := len(s) - 1  
    for first < last {  
        s[first], s[last] = s[last], s[first]  
        first++  
        last--  
    }  
}
```

```
func ReverseAndPrint(s []int) {  
    fmt.Println(Reverse(int)(s))  
    fmt.Println(Reverse(s))  
}
```

그냥 호출해도 컴파일러가 인식해서 처리

What could be different?

contract를 사용한 generic 타입 자료형 제약

```
contract Sequence(T) {  
    T string, []byte  
}  
  
func IndexByte (type T Sequence) (s T, b byte) int {  
    for i:= 0; i< len(s); i++ {  
        if s[i] == b {  
            return i  
        }  
    }  
    return -1  
}
```

<https://blog.golang.org/why-generics>

What could be different?

contract를 사용한 generic 타입 자료형 제약

```
contract Sequence(T) {  
    T string, []byte  
}
```

Generic 타입 뒤에 제약 추가

```
func IndexByte (type T Sequence) (s T b byte) int {  
    for i:= 0; i< len(s); i++ {  
        if s[i] == b {  
            return i  
        }  
    }  
    return -1  
}
```

<https://blog.golang.org/why-generics>

What could be different?

contract를 사용한 generic 타입 자료형 제약

```
contract Sequence(T) {  
    T string, []byte  
}  
  
func IndexByte (type T Sequence) (s T, b byte) int {  
    for i:= 0; i< len(s); i++ {  
        if s[i] == b {  
            return i  
        }  
    }  
    return -1  
}
```

len()을 적용하기 위해선 아무 타입이나 들어오면 안됨! 제약이 필요!

<https://blog.golang.org/why-generics>

What could be different?

contract를 사용한 generic 타입 자료형 제약

```
contract Sequence(T) {  
    T string, []byte  
}
```

타입 T는 string 이거나 []byte여야 한다는 제약!

```
func IndexByte (type T Sequence) (s T, b byte) int {  
    for i:= 0; i< len(s); i++ {  
        if s[i] == b {  
            return i  
        }  
    }  
    return -1  
}
```

<https://blog.golang.org/why-generics>

What could be different?

contract를 사용한 generic 타입 메서드 제약

```
contract Stringer(T) {  
    T String() string  
}  
  
func ToStringers (type E Stringer) (s []E) []string {  
    r := make([]string, len(s))  
    for i, v := range s {  
        r[i] = v.String()  
    }  
    return r  
}
```

What could be different?

contract를 사용한 generic 타입 메서드 제약

```
contract Stringer(T) {  
    T String() string  
}  
  
func ToStringers (type E Stringer) (s []E) []string {  
    r := make([]string, len(s))  
    for i, v := range s {  
        r[i] = v.String() String() 함수가 필요!  
    }  
    return r  
}
```

What could be different?

contract를 사용한 generic 타입 메서드 제약

```
contract Stringer(T) {  
    T String() string  
}
```

String() 함수를 구현해야 한다는 제약!

```
func ToStringers (type E Stringer) (s []E) []string {  
    r := make([]string, len(s))  
    for i, v := range s {  
        r[i] = v.String()  
    }  
    return r  
}
```

String() 함수가 필요!

What could be different?

구조체 정의에 generic type 추가

struct 정의 시에 generic 타입 수신

```
type Graph (type Node, Edge G) struct { ... }
```

```
contract G(Node, Edge) {  
    Node Edges() []Edge  
    Edge Nodes() (from Node, to Node)  
}
```

```
func New (type Node, Edge G) (nodes []Node) *Graph(Node, Edge) {  
    ...  
}
```

```
func (g *Graph(Node, Edge)) ShortestPath(from, to Node) []Edge {  
    ...  
}
```

What could be different?

구조체 정의에 generic type 추가

struct 정의 시에 generic 타입 수신

```
type Graph (type Node, Edge G) struct { ... }
```

```
contract G(Node, Edge) {  
    Node Edges() []Edge  
    Edge Nodes() (from Node, to Node)  
}
```

Node, Edge 타입이 구현해야할 메서드 제약

```
func New (type Node, Edge G) (nodes []Node) *Graph(Node, Edge) {  
    ...  
}
```

```
func (g *Graph(Node, Edge)) ShortestPath(from, to Node) []Edge {  
    ...  
}
```

What could be different?

구조체 정의에 generic type 추가

struct 정의 시에 generic 타입 수신

```
type Graph (type Node, Edge G) struct { ... }
```

```
contract G(Node, Edge) {  
    Node Edges() []Edge  
    Edge Nodes() (from Node, to Node)  
}
```

Node, Edge 타입이 구현해야할 메서드 제약

```
func New (type Node, Edge G) (nodes []Node) *Graph(Node, Edge) {  
    ...  
}
```

생성된 구조체는 generic 타입과 함께 표현

```
func (g *Graph(Node, Edge)) ShortestPath(from, to Node) []Edge {  
    ...  
}
```

What could be different?

구조체 정의에 generic type 추가

struct 정의 시에 generic 타입 수신

```
type Graph (type Node, Edge G) struct { ... }
```

```
contract G(Node, Edge) {  
    Node Edges() []Edge  
    Edge Nodes() (from Node, to Node)  
}
```

Node, Edge 타입이 구현해야할 메서드 제약

```
func New (type Node, Edge G) (nodes []Node) *Graph(Node, Edge) {  
    ...  
}
```

생성된 구조체는 generic 타입과 함께 표현

```
func (g *Graph(Node, Edge)) ShortestPath(from, to Node) []Edge {  
    ... 구조체에 메서드를 추가할 때에도 generic 추가  
}
```


generic을 이용한 Min, Max 함수의 구현

```
contract Ordered(T) {  
    T int, int8, int16, int32, int64,  
      uint, uint8, uint16, uint32, uint64, uintptr,  
      float32, float64,  
      string  
}  
  
func Min (type T contracts.Ordered) (a, b T) T {  
    if a < b {  
        return a  
    }  
    return b  
}
```

What could be different?

generic을 이용한 Min, Max 함수의 구현

```
contract Ordered(T) {  
    T int, int8, int16, int32, int64,  
      uint, uint8, uint16, uint32, uint64, uintptr,  
      float32, float64,  
      string  
}  
  
func Min (type T contracts.Ordered) (a, b T) T {  
    if a < b {  
        return a  
    }  
    return b  
}
```

<https://blog.golang.org/why-generics>

What could be different?

generic을 이용한 Min, Max 함수의 구현

```
contract Ordered(T) {  
    T int, int8, int16, int32, int64,  
      uint, uint8, uint16, uint32, uint64, uintptr,  
      float32, float64,  
      string  
}
```

go의 기본 자료형들

```
func Min (type T contracts.Ordered) (a, b T) T {  
    if a < b {  
        return a  
    }  
    return b  
}
```

<https://blog.golang.org/why-generics>

What could be different?

generic을 이용한 자료구조(binary tree)의 구현

```
type Tree (type E) struct {  
    root    *node(E)  
    compare func(E, E) int  
}
```

```
type node (type E) struct {  
    val      E  
    left, right *node(E)  
}
```

```
func New (type E) (cmp func(E, E) int) *Tree(E) {  
    return &Tree(E){compare: cmp}  
}
```

What could be different?

generic을 이용한 자료구조(binary tree)의 구현

```
func (t *Tree(E)) find(v E) **node(E) {
    pn := &t.root
    for *pn != nil {
        switch cmp := t.compare(v, (*pn).val); {
        case cmp < 0:
            pn = &(*pn).left
        case cmp > 0:
            pn = &(*pn).right
        default:
            return pn
        }
    }
    return pn
}
```

```
func (t *Tree(E)) Contains(v E) bool {
    return *t.find(v) != nil
}

func (t *Tree(E)) Insert(v E) bool {
    pn := t.find(v)
    if *pn != nil {
        return false
    }
    *pn = &node(E){val: v}
    return true
}
```

<https://blog.golang.org/why-generics>

What could be different?

이 외에 generic을 통해 구현 가능한 것들

함수

- slice에서 min, max 값 찾기
- slice에서 평균, 표준 편차 구하기
- maps에서 union, intersection 구하기
- node와 edge로 구성된 graph에서 shortest path 찾기
- slice/map에 변화를 주어 새로운 slice/map 얻어내기
- Timeout 없이 channel에서 읽어오기
- 두 채널을 하나의 채널로 합치기
- 여러 함수를 병렬적으로 호출하고, 결과를 하나의 슬라이스로 리턴하기

자료구조

- Sets
- self-balancing trees
- Multimaps
- Concurrent hash maps

개인적인 go generic에 대한 생각

- 평소에 generic의 필요성을 느끼셨던 분들은 편하게 쓰시면 좋을 것 같습니다.
- 팀 내 공용 라이브러리 개발자들에게 유용할 것 같습니다.
- 오픈 소스 개발하시는 분들에게는 새로운 컨트리뷰션의 기회가 될 것 같습니다.

질문있으신가요?



Reference

- [1] Go survey results 2016, 2017, 2019, google, <https://blog.golang.org/survey2016-results>, <https://blog.golang.org/survey2017-results>, <https://blog.golang.org/survey2019-results>
- [2] Why Generics?, Ian Lance Taylor, 2019, google, <https://blog.golang.org/why-generics>
- [3] 제네릭, 생활코딩, <https://opentutorials.org/course/1223/6237>
- [4] Why Use Generics?, oracle, <https://docs.oracle.com/javase/tutorial/java/generics/why.html>
- [5] Go FAQ, google, <https://golang.org/doc/faq#generics>
- [6] The Generic Dilemma, Russ Cox, google, <https://research.swtch.com/generic>
- [7] Who needs generic? Use instead!, Christoph Berger, 2016, <https://appliedgo.net/generics/>