泛型

@M了个J

https://github.com/CoderMJLee

http://cnblogs.com/mjios

小码哥教育
SEEMYGO

实力IT教育 www.520it.com

码拉松

泛型（Generics）

- 泛型可以将类型参数化，提高代码复用率，减少代码量

```swift
func swapValues<T>(_ a: inout T, _ b: inout T) {
    (a, b) = (b, a)
}
```

```swift
var i1 = 10
var i2 = 20
swapValues(&i1, &i2)

var d1 = 10.0
var d2 = 20.0
swapValues(&d1, &d2)

struct Date {
    var year = 0, month = 0, day = 0
}
var dd1 = Date(year: 2011, month: 9, day: 10)
var dd2 = Date(year: 2012, month: 10, day: 11)
swapValues(&dd1, &dd2)
```

泛型

```
class Stack<E> {
    var elements = [E]()
    func push(_ element: E) { elements.append(element) }
    func pop() -> E { elements.removeLast() }
    func top() -> E { elements.last! }
    func size() -> Int { elements.count }
}
```

```
var stack = Stack<Int>()
stack.push(11)
stack.push(22)
stack.push(33)
print(stack.top()) // 33
print(stack.pop()) // 33
print(stack.pop()) // 22
print(stack.pop()) // 11
print(stack.size()) // 0
```

```
class SubStack<E> : Stack<E> { }
```

```
struct Stack<E> {
    var elements = [E]()
    mutating func push(_ element: E) { elements.append(element) }
    mutating func pop() -> E { elements.removeLast() }
    func top() -> E { elements.last! }
    func size() -> Int { elements.count }
}
```

```
enum Score<T> {
    case point(T)
    case grade(String)
}
let score0 = Score<Int>.point(100)
let score1 = Score.point(99)
let score2 = Score.point(99.5)
let score3 = Score<Int>.grade("A")
```

关联类型（Associated Type）

- 关联类型的作用：给协议中用到的类型定义一个占位名称
- 协议中可以拥有多个关联类型

```swift
protocol Stackable {
    associatedtype Element // 关联类型
    mutating func push(_ element: Element)
    mutating func pop() -> Element
    func top() -> Element
    func size() -> Int
}
```

```swift
class Stack<E> : Stackable {
    // typealias Element = E
    var elements = [E]()
    func push(_ element: E) {
        elements.append(element)
    }
    func pop() -> E { elements.removeLast() }
    func top() -> E { elements.last! }
    func size() -> Int { elements.count }
}
```

```swift
class StringStack : Stackable {
    // 给关联类型设定真实类型
    // typealias Element = String
    var elements = [String]()
    func push(_ element: String) { elements.append(element) }
    func pop() -> String { elements.removeLast() }
    func top() -> String { elements.last! }
    func size() -> Int { elements.count }
}
var ss = StringStack()
ss.push("Jack")
ss.push("Rose")
```

类型约束

```swift
protocol Runnable { }
class Person { }
func swapValues<T : Person & Runnable>(_ a: inout T, _ b: inout T) {
    (a, b) = (b, a)
}
```

```swift
protocol Stackable {
    associatedtype Element: Equatable
}
class Stack<E : Equatable> : Stackable { }
```

```swift
func equal<S1: Stackable, S2: Stackable>(_ s1: S1, _ s2: S2) -> Bool
    where S1.Element == S2.Element, S1.Element : Hashable {
    return false
}
```

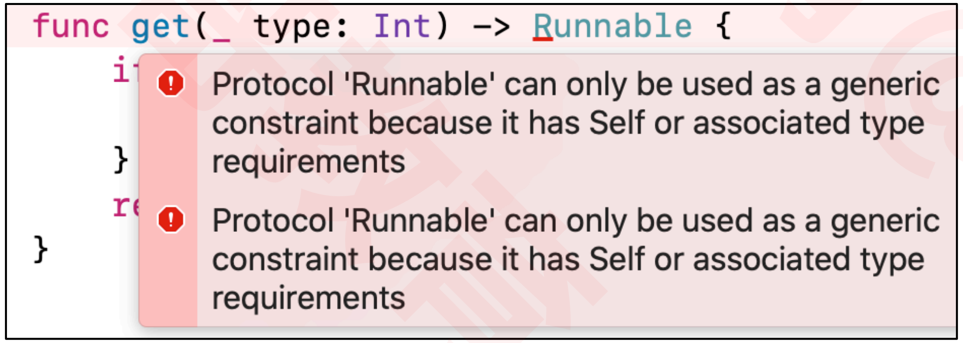
```swift
var stack1 = Stack<Int>()
var stack2 = Stack<String>()
// error: requires the types 'Int' and 'String' be equivalent
equal(stack1, stack2)
```

协议作返回值类型

```
protocol Runnable { }
class Person : Runnable { }
class Car : Runnable { }

func get(_ type: Int) -> Runnable {
    if type == 0 {
        return Person()
    }
    return Car()
}

var r1 = get(0)
var r2 = get(1)
```

- 如果协议中有associatedtype或者使用了Self作参数

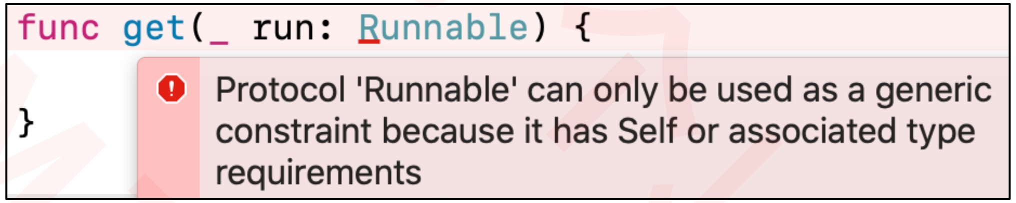
```
protocol Runnable {
    associatedtype Speed
    var speed: Speed { get }
}
class Person : Runnable {
    var speed: Double { 0.0 }
}
class Car : Runnable {
    var speed: Int { 0 }
}
```



```
func get(_ type: Int) -> Runnable {
    i
    }
    re
}
```

Protocol 'Runnable' can only be used as a generic constraint because it has Self or associated type requirements

Protocol 'Runnable' can only be used as a generic constraint because it has Self or associated type requirements



```
func get(_ run: Runnable) {
}
```

Protocol 'Runnable' can only be used as a generic constraint because it has Self or associated type requirements

协议作返回值类型

- 解决方案①：使用泛型

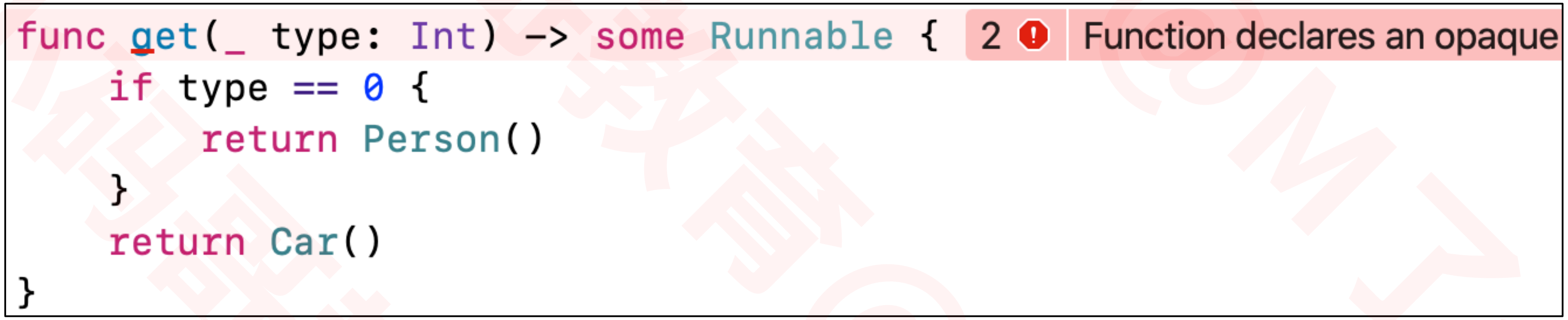
```swift
func get<T : Runnable>(_ type: Int) -> T {
    if type == 0 {
        return Person() as! T
    }
    return Car() as! T
}
var r1: Person = get(0)
var r2: Car = get(1)
```

协议作返回值类型

- 解决方案②：使用some关键字（Opaque Type，不透明类型）

```
func get(_ type: Int) -> some Runnable { Car() }
var r1 = get(0)
var r2 = get(1)
```

- some限制只能返回一种类型



```
func get(_ type: Int) -> some Runnable {  2 ❗ Function declares an opaque
    if type == 0 {
        return Person()
    }
    return Car()
}
```

可选项的本质

- 可选项的本质是enum类型

```
public enum Optional<Wrapped> : ExpressibleByNilLiteral {
    case none
    case some(Wrapped)
    public init(_ some: Wrapped)
}
```

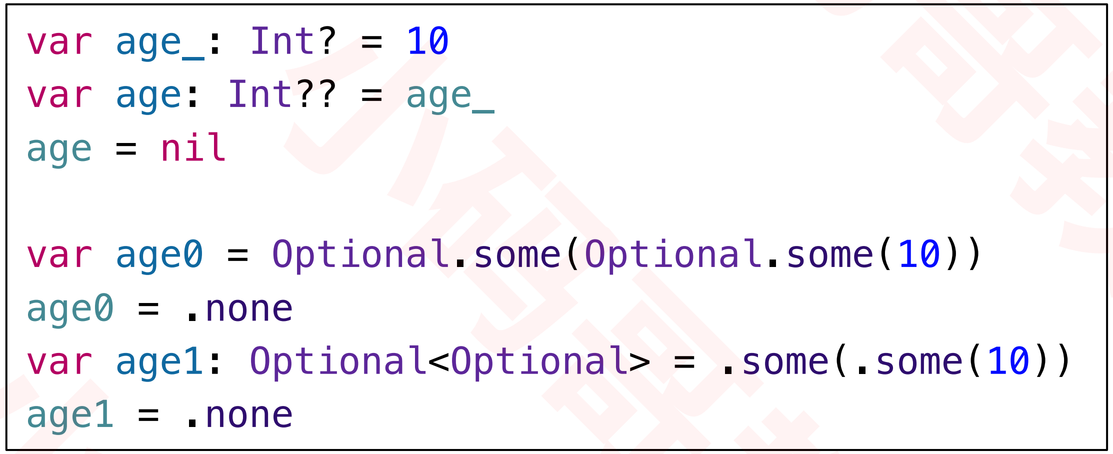
```
var age: Int? = .none
age = 10
age = .some(20)
age = nil
```

```
var age: Int? = 10
var age0: Optional<Int> = Optional<Int>.some(10)
var age1: Optional = .some(10)
var age2 = Optional.some(10)
var age3 = Optional(10)
age = nil
age3 = .none
```

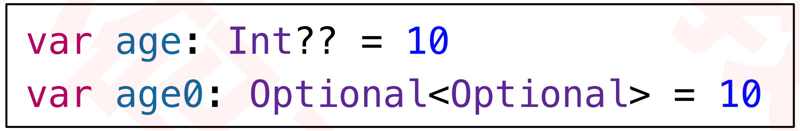
```
var age: Int? = nil
var age0 = Optional<Int>.none
var age1: Optional<Int> = .none
```

```
switch age {
case let v?:
    print("some", v)
case nil:
    print("none")
}

switch age {
case let .some(v):
    print("some", v)
case .none:
    print("none")
}
```

可选项的本质



```
var age_: Int? = 10
var age: Int?? = age_
age = nil

var age0 = Optional.some(Optional.some(10))
age0 = .none
var age1: Optional<Optional> = .some(.some(10))
age1 = .none
```



```
var age: Int?? = 10
var age0: Optional<Optional> = 10
```