- Swift中的泛型编程
  - 1.为什么要使用泛型编程
  - 2.Swift中使用泛型
  - 3.Swift中泛型约束

Swift中的泛型编程

1.为什么要使用泛型编程

> 举一个很简单的例子
> 现在要求写一个函数,具有的功能:
> 给你任何类型的数据
> 你都可以将它完整的打印出来
> 且只能使用一个函数

> > 1.第一种方法,使用Swift提供的Any类型

```
func PrintAny(_ object:Any) -> Any{
    print(object)
    return object
}
```

> > 2.使用Swift中的泛型

```
/*
注意的是这里的T很有意思,
它能够帮助我们做出很多事情，后续会详细的讲到
*/
func PrintAny<T>(_ object:T)->T{
    print(object)
    return object
}
```

> 3.两种方法的比较
> 第一种方法使用到了Any类型去定义了一个函数，虽然很好的适应到了我们的需求，但是面对多元化的需求时候，Any类型就无法帮助我们实现了,比如我们要求Any类型对应的值符合我们的Identifiable等协议,这时候很难去实现
> 此外Any类型在编译的时候会跳过类型检查,不够安全

> > 第二种方法使用到了泛型,T代表的是一个模糊的类型,可以理解成我们需要的一个类型，而这个类型可以成为任何类型(根据我们需求来)

这就是我们使用泛型的原因,因为泛型能够写出可重用,支持任意类型的函数

2.Swift中使用泛型

> 1.泛型的使用广度:
> Func,Struct,Class,enums,protocol,extesion均可以使用

> 2.从系统内置函数Swap中观察如何使用泛型
> 源码如下

```
public func swap<T>(_ a:inout T,_ b:inout T){
    /*
    inout 关键词
    指能够从函数内部修改外部实参的值
    在调用的使用 & 指针指向实参地址进行修改

    此外这里的函数参数a,b前面的 “_”的意义是
    在函数调用的时候可以忽略掉参数名，直接赋值
    */

    let p1  = Builtin.addressof(&a)
    let p2 = Builtin.addressof(&b)
    let tmp : T = Builtin.take(p1)
    Builtin.initialize(Builtin.take(p2) as T,p1)
    Builtin.initialize(tmp,p2)
    /* 这里的Builtin我们无需过多了解
    这里很明显能够看出来是将我们的a,b交换位置了
    */
}

// 调用swap
var a : int = 1  // -->这里我们可以将a,b换成任意类型的
var b : int = 2
swap(&a,&b) // swap(&T,&T)
print(a)
//result a = 2
```

> 从上面一例中,我们可以看出来泛型编程能够帮助我们扩大函数的使用范围,让我们不拘泥于单一函数类型,从而做到提高编写的效率

3.Swift中泛型约束

> 1在一开始说的例子里面，讲述了Any和泛型的区别
> 同时我们提到了泛型中,我们可以对于泛型T进行类型限制
> 这里的限制并不指对于数据类型的限制,而是对于该泛型的特性限制,例如你想要这个泛型继承于...,
> 或者符合怎样的协议...

> > 实现方法 --> Where关键词
> > 要求:定义一个结构体
> > 要求内置**第一个参数**的性质符合Hashable协议

```
//首先这里如果我们要求结构体符合Hashable协议
//我们可以写成如下

struct Anystudent<S,T>:Hashable{
    var name : S
    var old : T
}
//但是这里我们要求name符合,old无需符合
//第一种写法
struct Anystudent<S:Hashable,T>{
    var name : S
    var old : T
}
//第二种写法
struct Anystudent<S,T> where S:Hashable{
    var name : S
    var old : T
}
/*
这里我们就对S这个泛型进行了限制
也就是它必须符合我们的Hashbale协议，Anystudent才能被创建
*/
```

> > 定义其他类型也是一样

```swift
class AnyStudent<S,T> where S:Hashable{
    var name : S
    var old : T

    init(name:S,old:T){
        self.name = name
        self.old = old
    }
}
//枚举里面泛型似乎并没有太多意义
enum LoadingState<T> where T:Hashable{
    case success,
    case loading,
    case failure
}

//协议中的泛型不可以<T>的方式去写
protocol Rules{
    associatedtype itemType : Hashable
    //associatedtype关键词去定义它
    mutating func GetSome(_ a : itemType)
    var count : itemType{get}
}

//运用的protocol的时候
//需要使用typealias指出itemType即可
eg:
class StudentRules : Rules{
    typealias itemType = Int
    var count : itemType{
        get{
            return self.hashvalue / 2
        }
    }
}
```