泛型编程漫谈

LUNA 吴锡苗

- 以一个简单例子，体现泛型编程的基础用法以及作用
- 介绍泛型编程的定义及其解决的问题
- 对比其在不同语言中实现方案的异同（C++/Java/OC/Swift）
- Java泛型类型擦除以及如何解决
- 逆变、协变、不变
- 探究Swift中协变的实现

举个例子

交换两个Int的值

```
func swapTwoInts(a: inout Int, b: inout Int) {
    let tempA = a
    a = b
    b = tempA
}
```

交换两个String 或者 Double的值

```
func swapTwoDoubles(inout a: Double, inout b: Double) {
    let tempA = a
    a = b
    b = tempA
}
```

```
func swapTwoStrings(inout a: String, inout b: String) {
…
```

考虑使用泛型编程

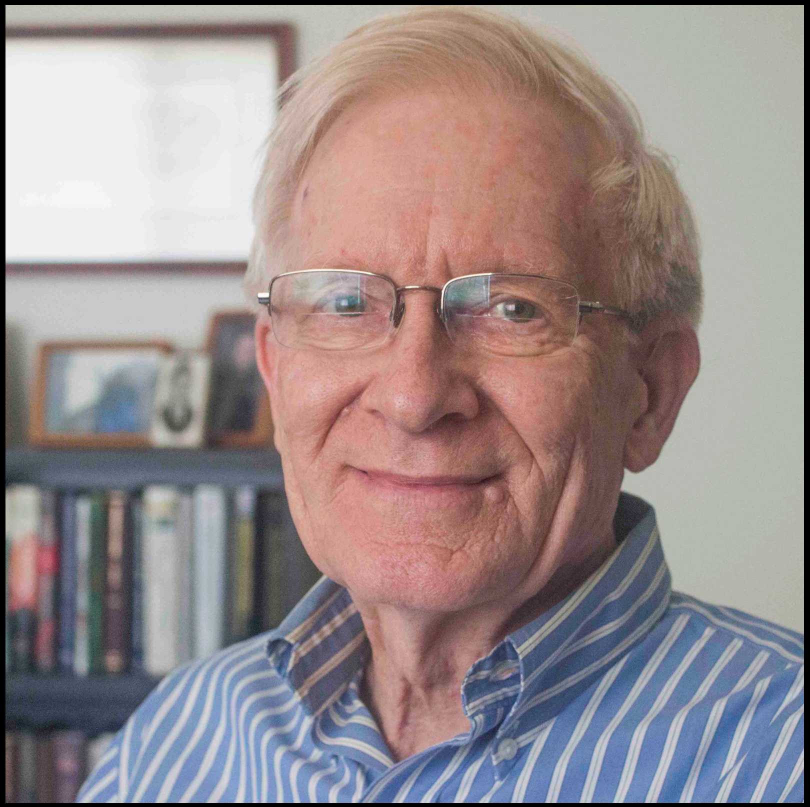
```
func swapTwoValues<T>(inout a: T, inout b: T) {
    let tempA = a
    a = b
    b = tempA
}
```

```
var firstInt = 8
var secondInt = 666
swapTwoValues(&firstInt, &secondInt)
```

```
var firstString = "first"
var secondString = "second"
swapTwoValues(&firstString, &secondString)
```

David R. Musse

Alexander A. Stepanov



Generic programming centers around the idea of abstracting from concrete, efficient algorithms to obtain generic algorithms that can be combined with different data representations to produce a wide variety of useful software.

— Musser, David R.; Stepanov, Alexander A., Generic Programming

泛型编程

泛型编程（generic programming）是一种计算机编程风格。
在泛型编程中，算法是根据稍后要指定的类型编写的，然后在需要时根据作为参数提供的特定类型来实例化。

```swift
func swapTwoValues<T>(inout a: T, inout b: T) {
    let tempA = a
    a = b
    b = tempA
}
```

```swift
var firstString = "first"
var secondString = "second"
swapTwoValues(&firstString, &secondString)
```

泛型编程解决的问题

最初提出时的动机：发明一种语言机制，能够帮助实现一个通用的标准容器库。

使用泛型可以避免重复的代码

一些强类型程序语言支持泛型，其主要目的是加强类型安全及减少类转换的次数

```
UILabel* label = [UILabel new];
NSMutableArray* array = [NSMutableArray new];
[array addObject:@"Hello world!"];
[label setText: (NSString *)array[0]];

NSMutableArray<NSString*>* templateArray = [NSMutableArray new];
[templateArray addObject: @"Hello world!"];
[label setText: templateArray[0]];
```

不同语言对泛型的实现

语言	加入	特性加入时间	语言发布时间	特性
C++	STL	1987	1998	模板编程,编译期根据模板针对不同类型生成不相关的多个独立的代码。
Objective-C	Xcode 7.0	2015	1980	轻量级泛型，编译期只做类型检查，不支持逆变协变
Java	J2SE 5.0	2004	1996	编译期只做类型检查，只生成一份代码，运行时类型被擦除，支持协变和逆变。
Swift	Xcode 6.0	2014	2014	编译期编译成多份不同的代码，容器类型支持协变，不支持逆变

c++模板编程代码示例

```
template<class T>
class TestTemplate {
    T *instance;
    void test() {
        instance->anyFunction();
    }
};
```

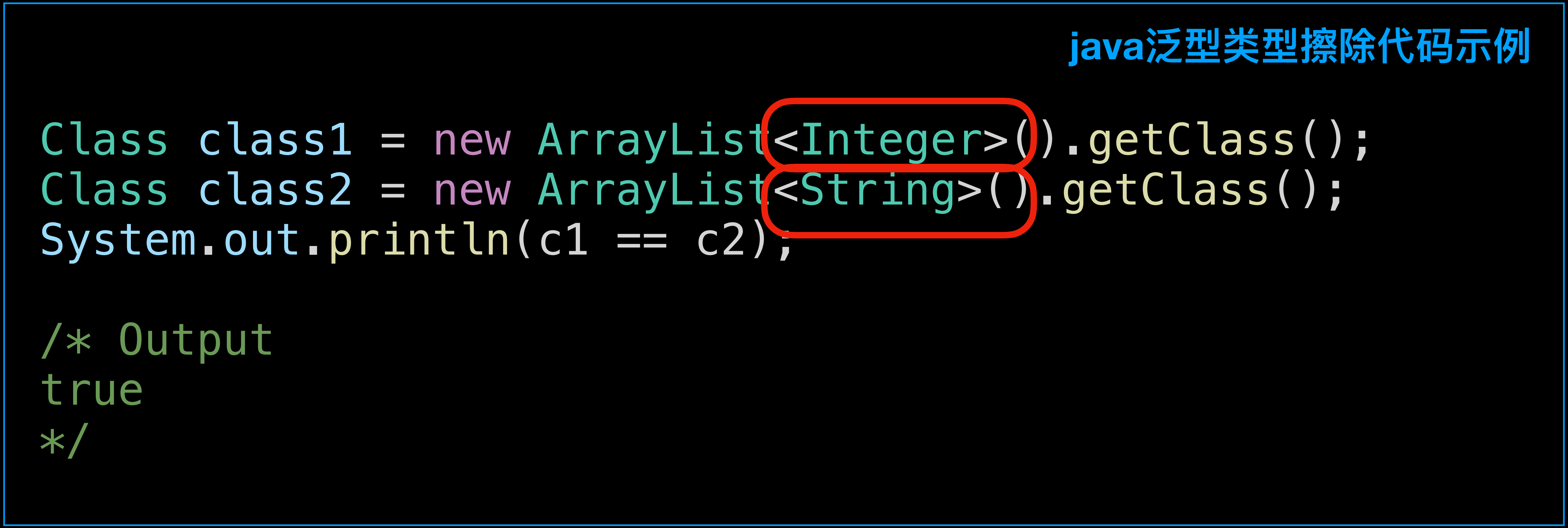
c++使用类的成员作为模板的限制，而不是接口或基类

```
template<class T, int Num>
class TestTemplate {
    T *instance;
    void test() {
        printf("%d", Num);
    }
};
```

可以使用常量作为模板参数

OC轻量级泛型使用示例

```
UILabel* label = [UILabel new];
NSMutableArray* array = [NSMutableArray new];
[array addObject:@"Hello world!"];
[label setText: (NSString *)array[0]];

NSMutableArray<NSString*>* templateArray = [NSMutableArray new];
[templateArray addObject: @"Hello world!"];
[label setText: templateArray[0]];
```



java泛型类型擦除代码示例

```java
Class class1 = new ArrayList<Integer>().getClass();
Class class2 = new ArrayList<String>().getClass();
System.out.println(c1 == c2);

/* Output
true
*/
```

回顾c++对泛型参数的限制

```
template<class T>
class TestTemplate {
    T *instance;
    void test() {
        instance->anyFunction();
    }
};
```

Java有界类型

```
public class AnimalHouse<T extends Animal> {
    List<T> animals;
    void feed() {
        for int i; i < animals.count; i++ {
            animals[i].eat()
        }
    }
}
```

协变（covariance）：父类参数可以用子类替换

逆变（contravariance）：与协变相反

不变（invariance）：不可替换

```
协变
List<Animal> animal = new List<Bird>()

逆变（不能编译通过，仅演示含义）
List<Bird> birds = new List<Animal>()
```

Java协变，参数中的
协变用于读(输入)参数

```
void feedAll(List<Animal> animals) {
    for int i; i < animals.count; i++ {
        animals[i].eat()
    }
}

void feedAll(List<? extends Animal> animals) {
    for int i; i < animals.count; i++ {
        animals[i].eat()
    }
}


func test() {
    List<Animal> animals = [cat1, dog1, bird1];
    feedAll(animals);

    List<Cat> cats ❌ [cat1, cat2, cat3];
        feedAll(ca❌;
}
```

Java逆变，参数中的
逆变用于写(输出)参数

```
void addCat(List<? super Cat> animals) {
    animals.add(new Cat());
}


List<Cat> cats = [cat1, cat2, cat3];
addCat(cats);


List<Animal> animals = [cat1, dog1, bird1];
addCat(animals);


List<ChinaCat> chinaCats = […];
addCat(chinaCats)
```
✗

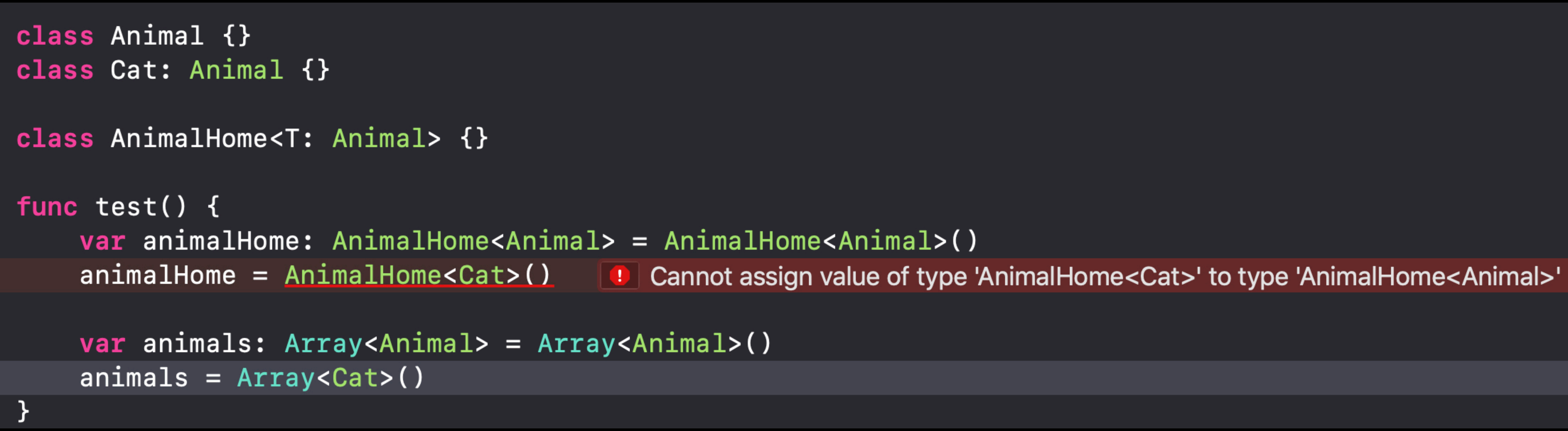
Swift中的协变

```
class Animal {
    func eat() {
        print("eat")
    }
}
```

```
class Cat: Animal {
    func jump() {
    }
}
```

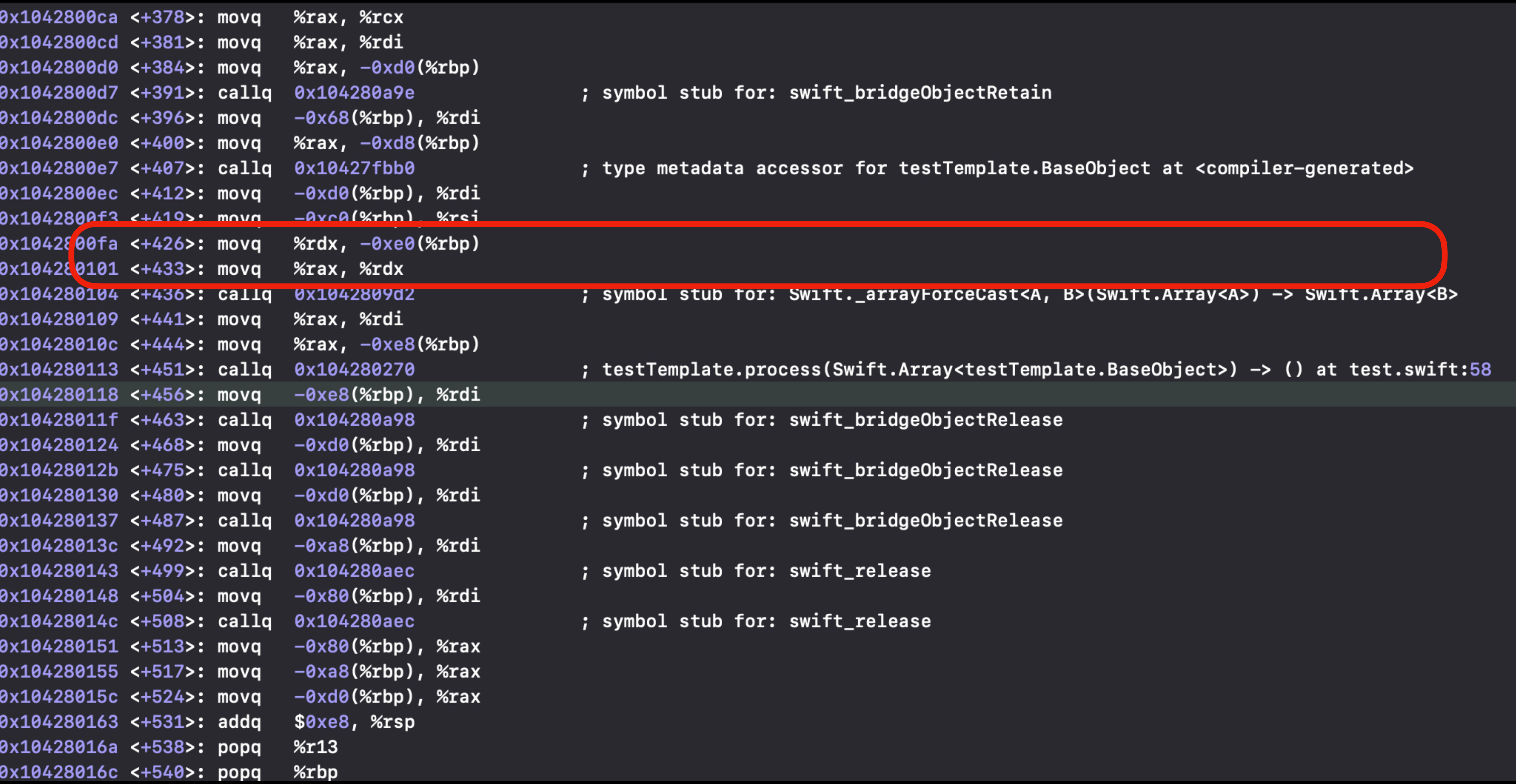
```
//[Animal]等同于 Array<Animal>
func feedAnimal(animals: [Animal]) {
    for animal in animals {
        animal.eat()
    }
}


func test() {
    let cats = [Cat(), Cat(), Cat()]
    feedAnimal(animals: cats)
}
```

协变在Swift中的表现

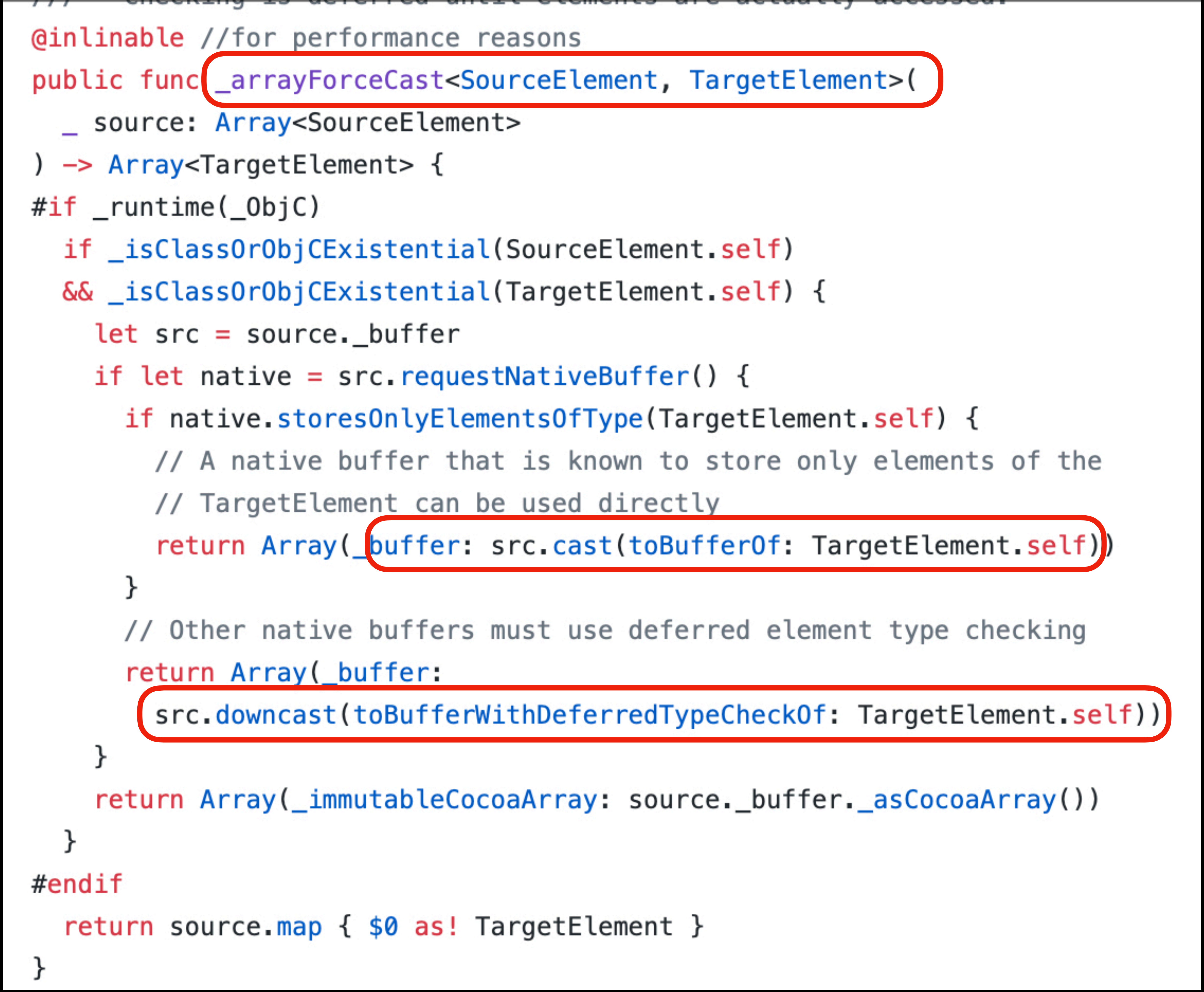


```swift
class Animal {}
class Cat: Animal {}

class AnimalHome<T: Animal> {}

func test() {
    var animalHome: AnimalHome<Animal> = AnimalHome<Animal>()
    animalHome = AnimalHome<Cat>()

    var animals: Array<Animal> = Array<Animal>()
    animals = Array<Cat>()
}
```

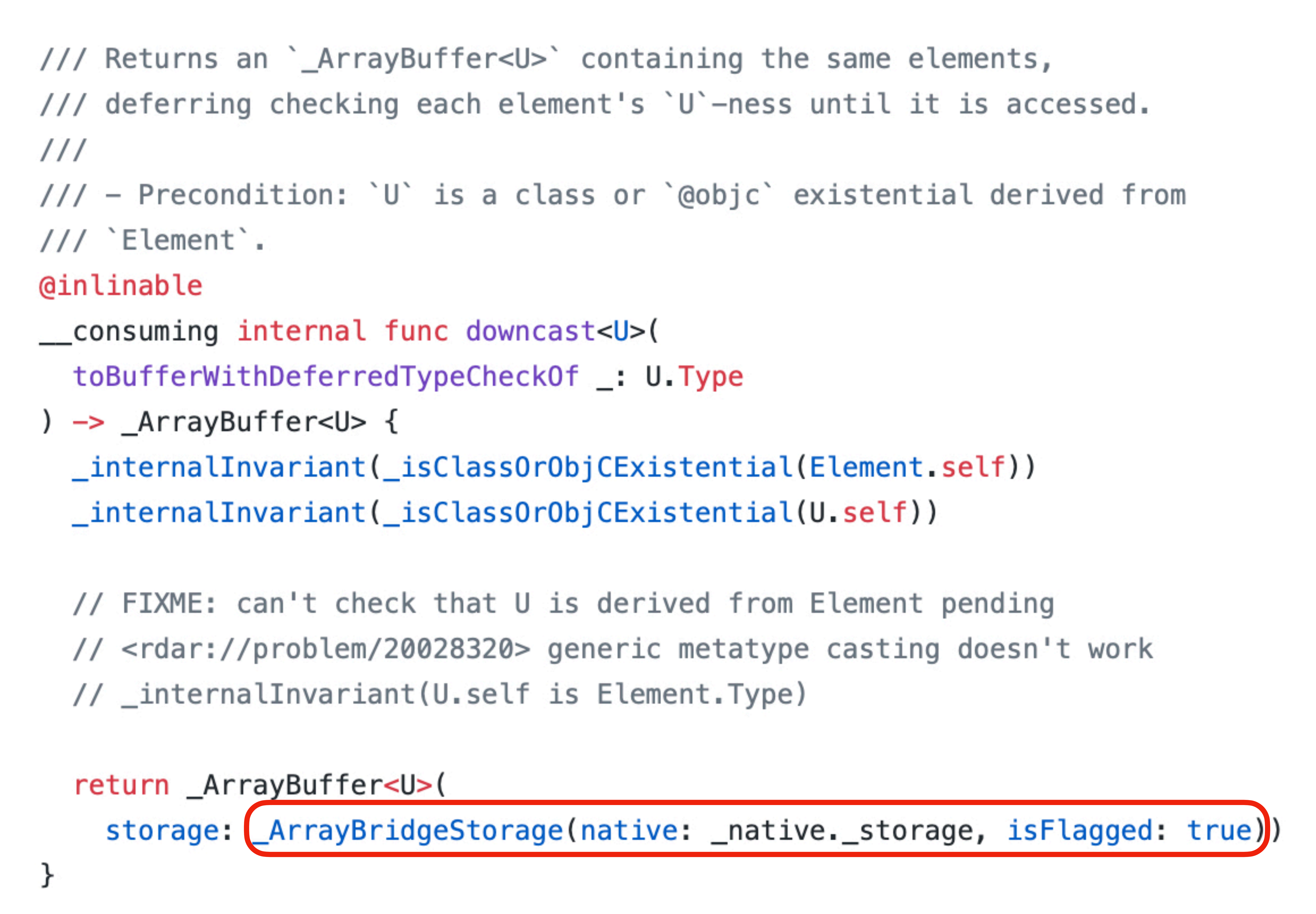
Cannot assign value of type 'AnimalHome<Cat>' to type 'AnimalHome<Animal>'



```
0x1042800ca <+378>: movq   %rax, %rcx
0x1042800cd <+381>: movq   %rax, %rdi
0x1042800d0 <+384>: movq   %rax, -0xd0(%rbp)
0x1042800d7 <+391>: callq  0x104280a9e               ; symbol stub for: swift_bridgeObjectRetain
0x1042800dc <+396>: movq   -0x68(%rbp), %rdi
0x1042800e0 <+400>: movq   %rax, -0xd8(%rbp)
0x1042800e7 <+407>: callq  0x10427fbb0               ; type metadata accessor for testTemplate.BaseObject at <compiler-generated>
0x1042800ec <+412>: movq   -0xd0(%rbp), %rdi
0x1042800f3 <+419>: movq   -0xc0(%rbp), %rsi
0x1042800fa <+426>: movq   %rdx, -0xe0(%rbp)
0x104280101 <+433>: movq   %rax, %rdx
0x104280104 <+436>: callq  0x1042809d2               ; symbol stub for: Swift._arrayForceCast<A, B>(Swift.Array<A>) -> Swift.Array<B>
0x104280109 <+441>: movq   %rax, %rdi
0x10428010c <+444>: movq   %rax, -0xe8(%rbp)
0x104280113 <+451>: callq  0x104280270               ; testTemplate.process(Swift.Array<testTemplate.BaseObject>) -> () at test.swift:58
0x104280118 <+456>: movq   -0xe8(%rbp), %rdi
0x10428011f <+463>: callq  0x104280a98               ; symbol stub for: swift_bridgeObjectRelease
0x104280124 <+468>: movq   -0xd0(%rbp), %rdi
0x10428012b <+475>: callq  0x104280a98               ; symbol stub for: swift_bridgeObjectRelease
0x104280130 <+480>: movq   -0xd0(%rbp), %rdi
0x104280137 <+487>: callq  0x104280a98               ; symbol stub for: swift_bridgeObjectRelease
0x10428013c <+492>: movq   -0xa8(%rbp), %rdi
0x104280143 <+499>: callq  0x104280aec               ; symbol stub for: swift_release
0x104280148 <+504>: movq   -0x80(%rbp), %rdi
0x10428014c <+508>: callq  0x104280aec               ; symbol stub for: swift_release
0x104280151 <+513>: movq   -0x80(%rbp), %rax
0x104280155 <+517>: movq   -0xa8(%rbp), %rax
0x10428015c <+524>: movq   -0xd0(%rbp), %rax
0x104280163 <+531>: addq   $0xe8, %rsp
0x10428016a <+538>: popq   %r13
0x10428016c <+540>: popq   %rbp
```



```swift
/// checking is deferred until elements are actually accessed.
@inlinable //for performance reasons
public func _arrayForceCast<SourceElement, TargetElement>(
  _ source: Array<SourceElement>
) -> Array<TargetElement> {
#if _runtime(_ObjC)
  if _isClassOrObjCExistential(SourceElement.self)
  && _isClassOrObjCExistential(TargetElement.self) {
    let src = source._buffer
    if let native = src.requestNativeBuffer() {
      if native.storesOnlyElementsOfType(TargetElement.self) {
        // A native buffer that is known to store only elements of the
        // TargetElement can be used directly
        return Array(_buffer: src.cast(toBufferOf: TargetElement.self))
      }
      // Other native buffers must use deferred element type checking
      return Array(_buffer:
        src.downcast(toBufferWithDeferredTypeCheckOf: TargetElement.self))
    }
    return Array(_immutableCocoaArray: source._buffer._asCocoaArray())
  }
#endif
  return source.map { $0 as! TargetElement }
}
```



```
/// Returns an `_ArrayBuffer<U>` containing the same elements,
/// deferring checking each element's `U`-ness until it is accessed.
///
/// - Precondition: `U` is a class or `@objc` existential derived from
/// `Element`.
@inlinable
__consuming internal func downcast<U>(
  toBufferWithDeferredTypeCheckOf _: U.Type
) -> _ArrayBuffer<U> {
  _internalInvariant(_isClassOrObjCExistential(Element.self))
  _internalInvariant(_isClassOrObjCExistential(U.self))

  // FIXME: can't check that U is derived from Element pending
  // <rdar://problem/20028320> generic metatype casting doesn't work
  // _internalInvariant(U.self is Element.Type)

  return _ArrayBuffer<U>(
    storage: _ArrayBridgeStorage(native: _native._storage, isFlagged: true))
}
```

谢谢

希望通过这次一起对泛型的了解，可以更加对泛型编程感兴趣，并且更顺手的使用泛型编程这个好工具

下一场：《Swift泛型实战》适合iOS开发以及对swift编程感兴趣的同学将通过实际代码一步一步演示swift中泛型的高级用法

如何实现泛型的extension

- 第一步、声明一个空的protocol
- 第二步、让需要实现extesion的类遵守这个Protocol
- 第三步、对这个protocol实现extension
- 这时protocol中的关联类型Self对应遵守protocol的类的类型
- https://github.com/wudijimao/SwiftGenricExample