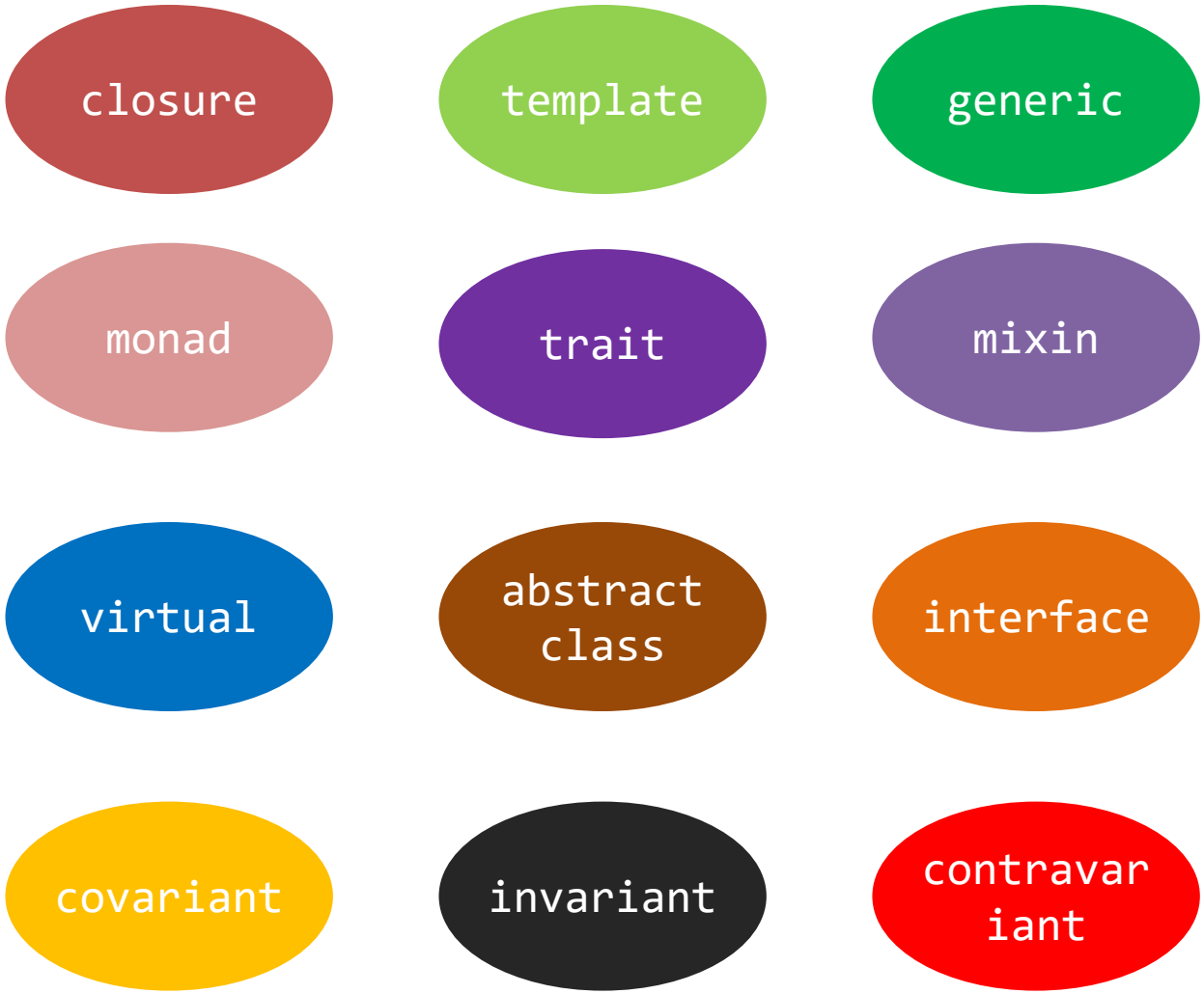
Lecture 6.3

高级类型系统

徐 辉

xuh@fudan.edu.cn


closure
template
generic
monad
trait
mixin
virtual
abstract class
interface
covariant
invariant
contravariant

大纲

- 一、类型代码复用（inheritance）
- 二、统一函数接口（polymorphism）
- 三、动态派发
- 四、函数式编程
- 五、类型协变

代码复用

- 结构体类型代码复用
  - 结构体（对象）继承
- 功能代码复用：
  - Trait、Mix-in

结构体定义和代码复用

```
struct A {
    int a;
    float b;
}

struct B : A {
    void* foo;
}
```

复用primitive type

复用compound type

```
struct A { … }
struct B { … }
struct S : B + A { … }
```

复用多个compound type

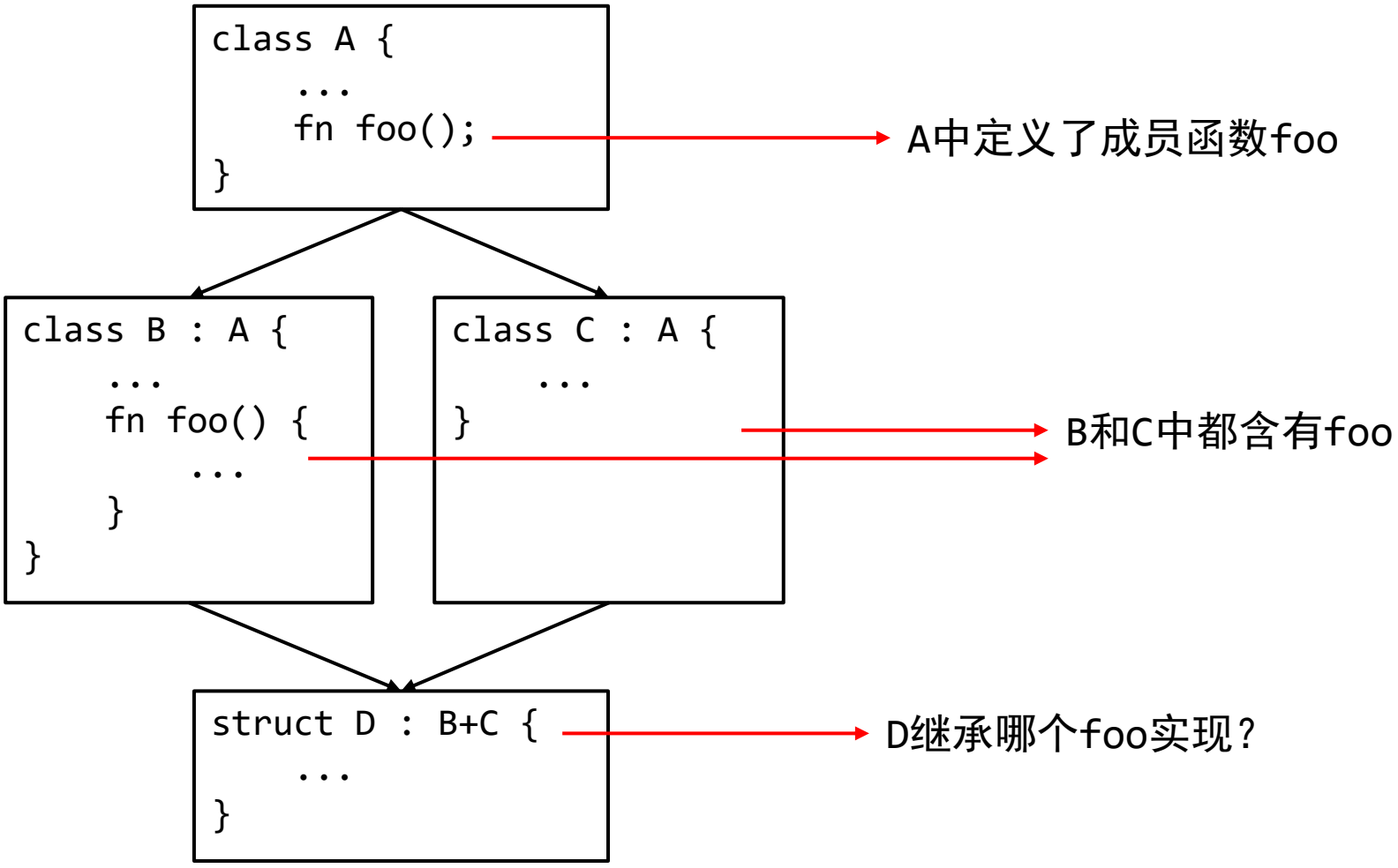
如果如果A和B都包含同一个函数？

Class vs Struct: 函数指针

```
struct A {
    int a;
    void* foo; //空指针
    int (* bar)(int);//函数指针
}

fn any (){...}
A.foo = any;
(* (unsigned int (*)(void)) A.foo)();
A.bar(1);
```

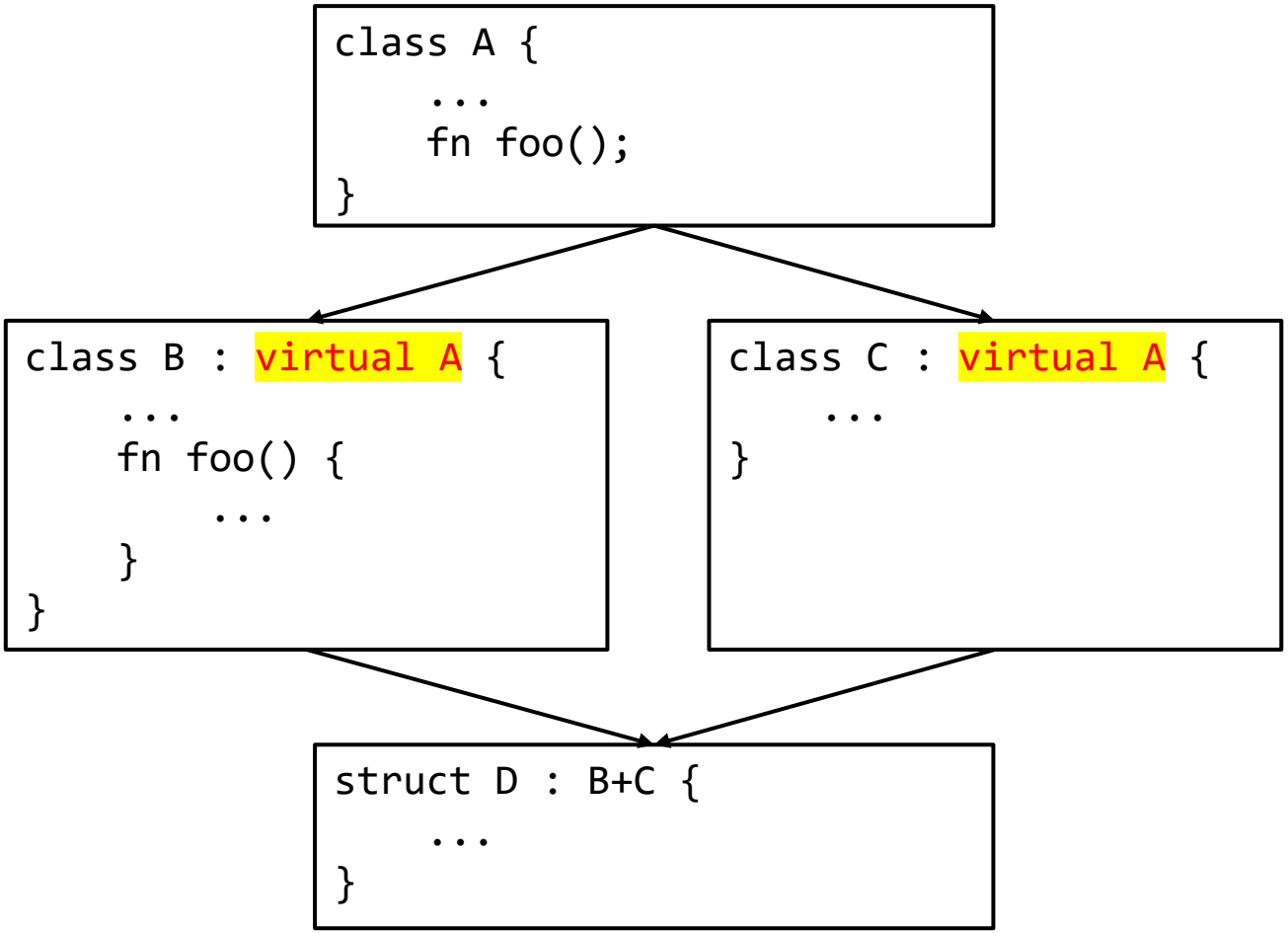
```
class A {
    int a;
    int bar(int b);//成员函数
}

A.bar(1);
```

多继承问题：the diamond problem



解决多继承问题：C++虚拟继承

- 虚拟继承：保证只有一个副本被拷贝



应对多继承问题：Java Interface

- Java语言不支持多继承
  - 规格继承：Interface
  - Interface只包括虚函数，无函数实现
  - 需要为class实现

```
class S {...}

interface A {
    fn foo();
}

interface B {
    fn bar();
}

impl A, B for S {
    fn foo(){...}
    fn bar(){...}
}
```

功能代码复用：Mixin

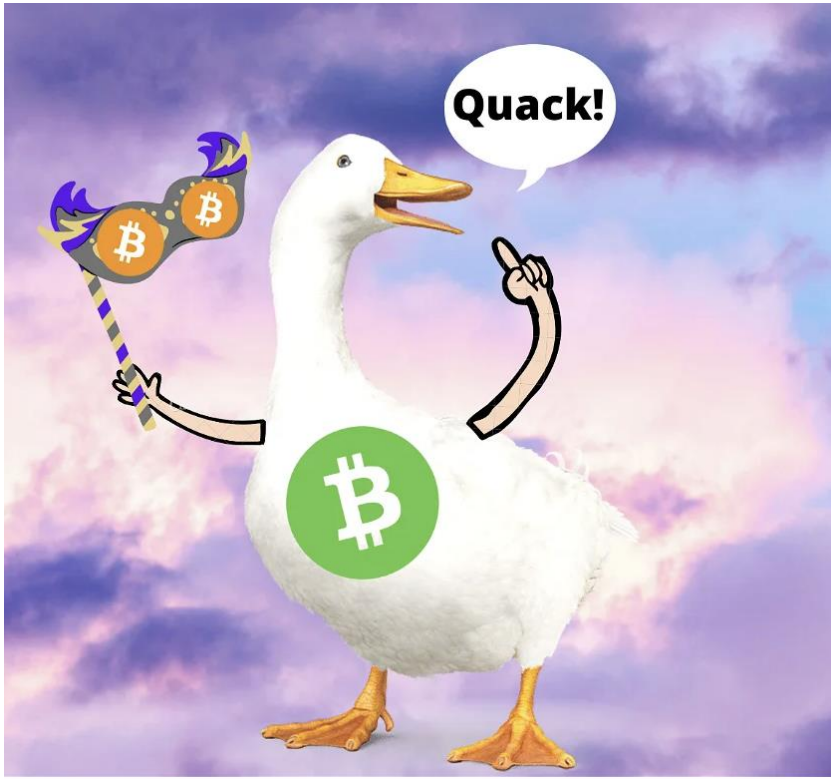
- Mixin：使用其它class中的方法而无需继承

```
interface A {
    fn foo();
}

interface B {
    fn bar();
}

class ImplA : impl A{
    fn foo(){...}
}

class ImplB : impl B{
    fn bar(){...}
}
```

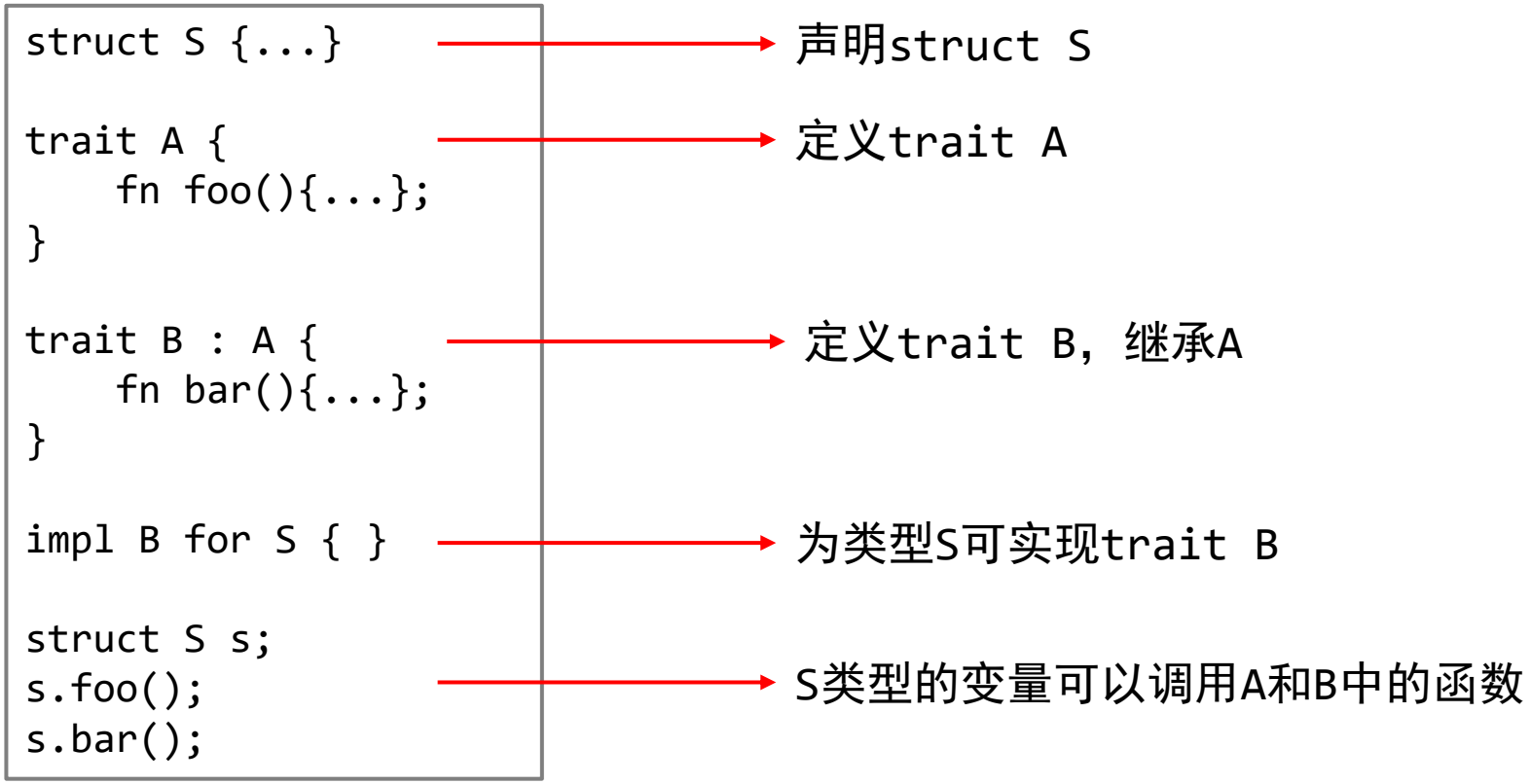
```
class S impl A, B {
    ImplA a;
    ImplB b;

    fn foo(){
        a.foo();
    }

    fn bar(){
        b.bar();
    }
}
```

Duck Typing：数据和功能分离的思想

“If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck”



功能代码复用：Rust Trait



大纲

- 一、类型代码复用（inheritance）
- 二、统一函数接口（polymorphism）
- 三、动态派发
- 四、函数式编程
- 五、协变和逆变

比较两个数的大小，并返回较大的一个

- 泛型参数：
  - C++ 模版（template）
  - Rust 泛型（generic）

```
//C++代码
int max(int x, int y) {
    return (x > y) ? x : y;
}
double max(double x, double y) {
    return (x > y) ? x : y;
}
char max(char x, char y) {
    return (x > y) ? x : y;
}
max(3, 7);
max(3.0, 7.0);
max('g', 'e');
```

```
//C++代码
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

```
//Rust 代码
fn max(x:T, y:T) -> T {
    return if(x > y) x else y;
}
```

如果没有泛型参数？

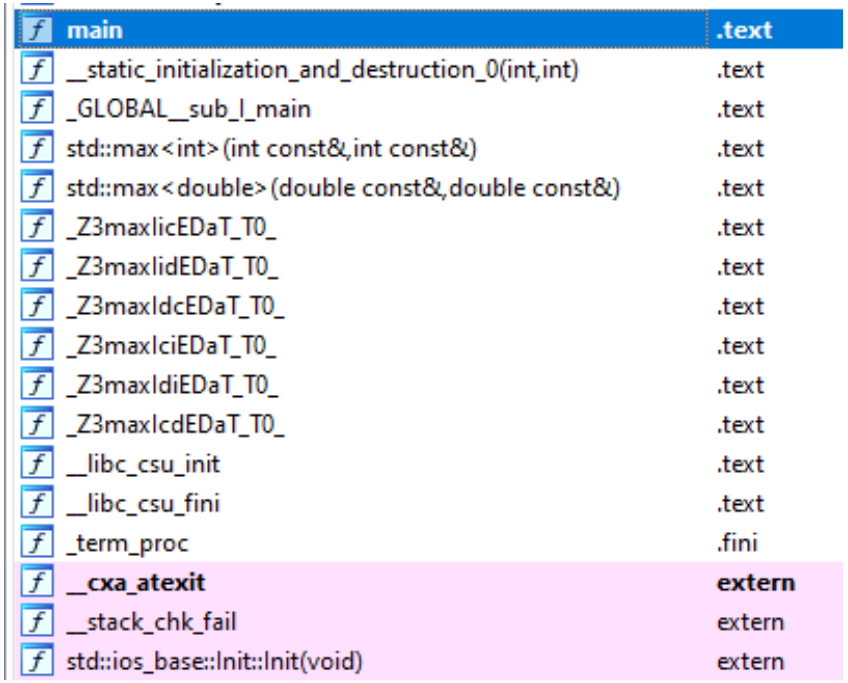
- 编译器对运算符多态的支持
- 基于void可以实现类似的功能

```
int a = 1 + 2;
int = 'a' + 'b';
```

```
void *max(void *x, void *y, int* (*f)(void *, void *)) {
    if (f(x, y) > 0)
        return x;
    else
        return y;
}

int* compare(void *x, void *y) {
    return (* (int *) x > * (int *) y) ? 1 : 0;
}

int *a = 123;
int *b = 234;
int *r = (int *)max(&a, &b, compare);
printf("max = %d\n", *r);
```

多个泛型参数：C++

```
template <typename T, typename G>
auto max(T x, G y) {
    return (x > y) ? x : y;
}

max(3, 7);
max(3.0, 7.0);
max(3, 'g');
max(3, 7.0);
max(3.0, 'g');
max('g', 3);
max(7.0, 3);
max('g', 3.0);
```



泛型的实现

- 编译阶段推导确定具体类型；
- 也可以通过属性指定泛型的具体类型；

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}

max(3, 7);
max(3.0, 7.0);
max('g', 'e');
```

⟦max⟧ = (T,T)→T

T = int
⟦max⟧ = (int,int)→int

```
template <typename T, typename G>
auto max(T x, G y) {
    return (x > y) ? x : y;
}

max<int,char>(123,'g');
max(3, 1.5);
max(3.0, 'g');
```

?

https://en.cppreference.com/w/cpp/language/template_argument_deduction

子类型

- 类型之间存在偏序关系，如$X \leq Y$表示：
  - X是Y的子类型
  - Y是的父类型
- 偏序的特性：
  - 自反性：$X \leq X$
  - 传递性：$X \leq Y$，$Y \leq Z \implies X \leq Z$

Liskov替换原理和类型约束

- 当类型约束为父类型时，可用子类型的对象
- 子类型的数据结构可兼容父类型

```
public class B extends Number {
    ...
}

public class A {
    public <T extends Number> void foo(T t){
        ...
    }
}

Class A a;
Class B b;
a.foo(b);
```

Upcast和Downcast

- Upcasting：如果X>Y，将Y类型转换为X类型
  - Liskov替换原理：一般不存在风险，默认都允许
- Downcasting：如果X>Y，将X类型转换为Y类型
  - 类型检查，如果类型不匹配会抛出异常

```
class Base {};
class Derived : public Derived {};

int main(int argc, const char** argv) {
    Base* base = new Base();
    if(Derived* derived = dynamic_cast<Derived *>(base)){
        ...
    }
}
```

Trait之间可以存在偏序关系

- 但非类型之间的偏序关系

```
struct S { }
trait A { }
trait B : A { }
impl B for S { }
```

=> B < A

```
trait A { }
trait B { }
impl<T> B for T where T:A { }
```

=> B < A

```
struct S1 { }
struct S2 { }
trait A { }
trait B { }
impl A for S2 { }
impl B for S2 { }
impl A for S1 { }
```


=> S2 < S1

Trait用于类型约束

```
trait A { }
trait B : A { }
struct S { }

impl A for S { }
impl B for S { }

fn makeacall<T:A>(s: &T){
    ...
}

fn main() {
  let a = S {};
  makeacall(&a);
}
```

```
trait A { }
trait B { }
struct S { }

impl A for S { }
impl<T> B for T where T:A { }

fn makeacall<T:B>(s: &T){
    ...
}

fn main() {
  let a = S {};
  makeacall(&a);
}
```

Rust使用Trait作为泛型的类型约束

```
trait Countable{ fn getcount(&self) -> u32; }
struct MyList{ val:u32, next:Option<Box<MyList>>, }

impl Countable for MyList {
    fn getcount(&self) -> u32 {
        let mut r = self.val;
        let mut cur = &self.next;
        loop {
            match cur {
                Some(x) => { r = r+x.val; cur = &x.next}
                _ => {break;}
            }
        }
        return r;
    }
}

fn foo<T:Countable>(t: T) { println!("Count: {:?}", t. getcount()); }

fn main() {
    let l = MyList{val:1, next:Some(Box::new(MyList{val:2, next:None}))};
    foo(l);
}
```

基于生命周期的subtype：Rust

- 如果s的生命周期大于t，则s是t的subtype

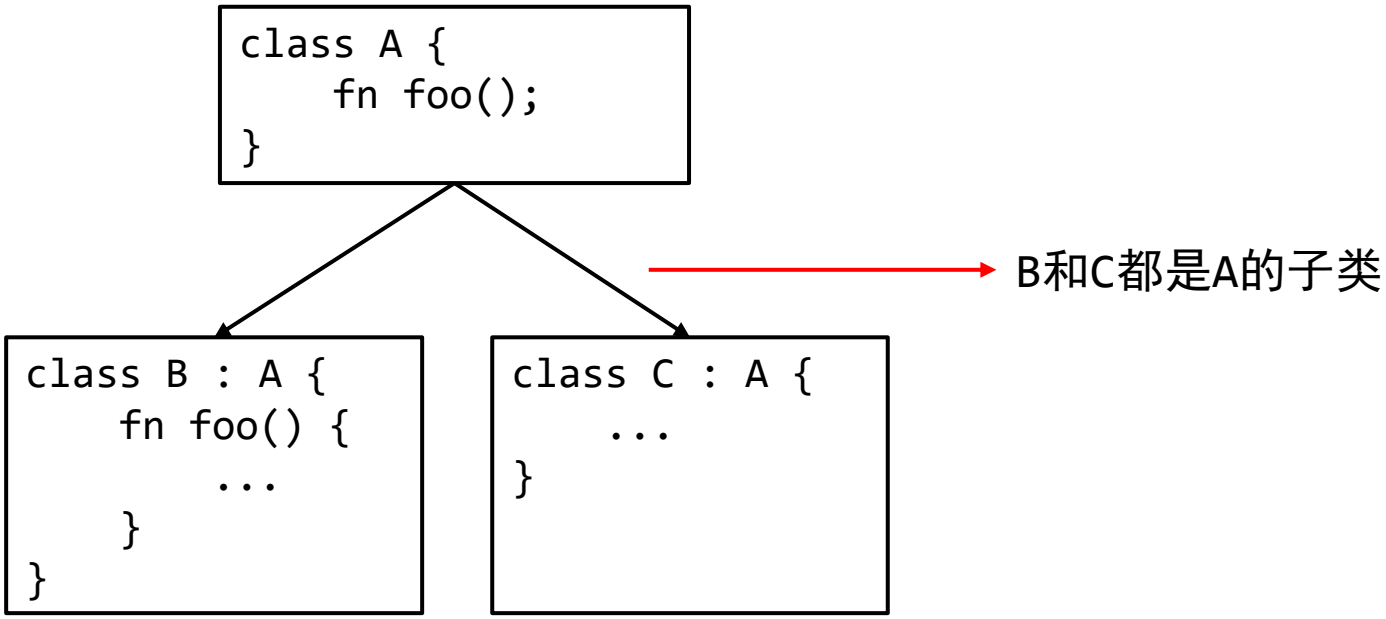
```
fn main() {
    let s: &'static str = "hi";
    let t: &'a str = s;
}
```

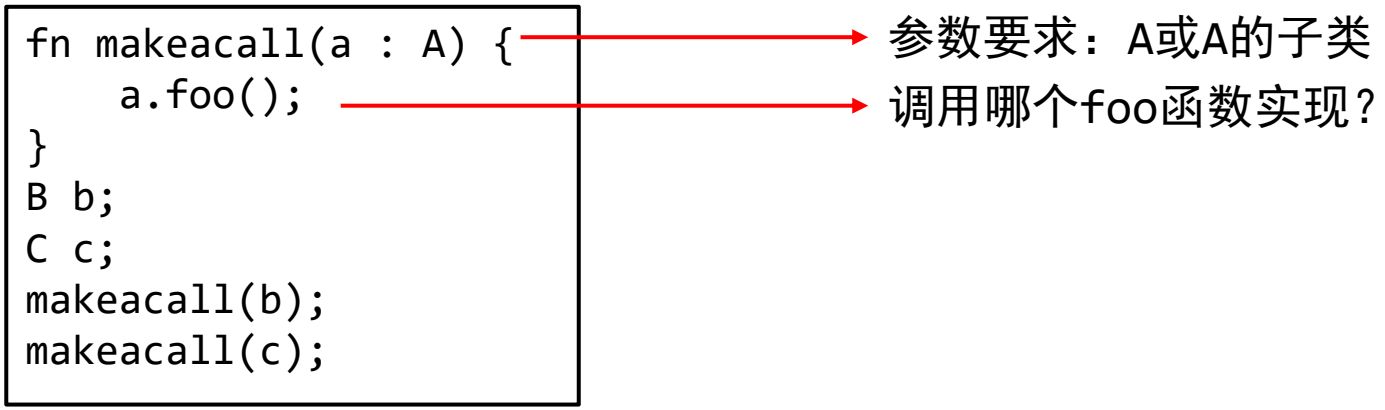
=> s < t

大纲

- 一、类型代码复用（inheritance）
- 二、统一函数接口（polymorphism）
- 三、动态派发
- 四、函数式编程
- 五、类型协变

Liskov替换带来的问题





下面这段代码输出什么？

```
class Base {
public:
  void print(){ cout << "base print" << endl;}
  virtual void speak(){ cout << "base speak" << endl;}
  virtual void shout(){ cout << "base shout" << endl;}
  virtual ~Base(){ cout << "destroying base" << endl;}
};

class Derived : public Base {
public:
  void print(){ cout << "derived print" << endl;}
  virtual void speak(){ cout << "derived speak" << endl;}
  virtual ~Derived(){ cout << "destroying derived" << endl;}
};

void test(Base* bptr){
  bptr->print();
  bptr->speak();
  bptr->shout();
}

int main(){
  Derived dobj;
  test(&dobj);
}
```

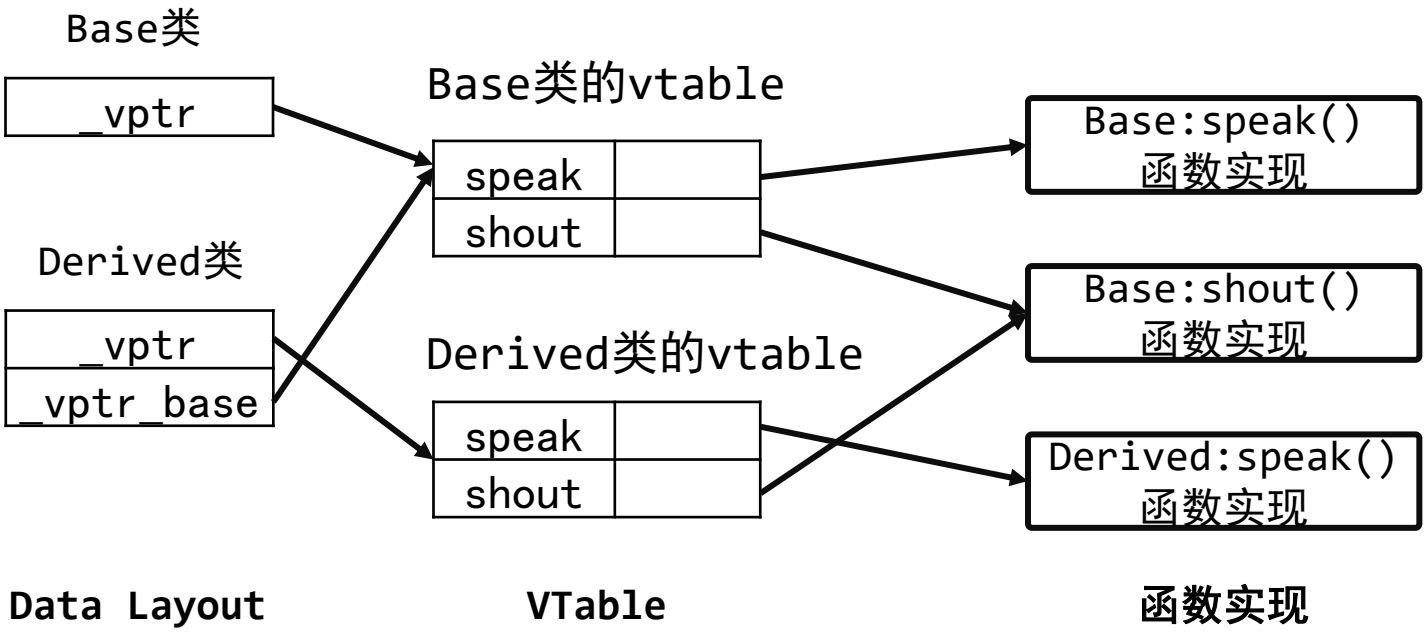
```
base print
derived speak
base shout
destroying derived
destroying base
```

虚函数和动态绑定

- 静态绑定：可在编译时确定执行版本
  - 通过对象类型调用任意函数
  - 调用非虚函数
- 动态绑定：直到运行时才能确定执行版本
  - C++虚函数
  - Rust dynamic trait

C++如何实现动态分发

- 编译器为每个类创建一个虚拟指针（vptr）指向虚拟方法表格（vtable：virtual method table）
- vtable包含每一个可用虚函数以及指向其具体函数实现的指针。



https://itanium-cxx-abi.github.io/cxx-abi/cxx-vtable-ex.html

Clang++的VTable

```
clang++ -Xclang -fdump-vtable-layouts
```

```
Vtable for 'Base' (6 entries).
   0 | offset_to_top (0)
   1 | Base RTTI
       -- (Base, 0) vtable address --
   2 | void Base::speak()
   3 | void Base::shout()
   4 | Base::~Base() [complete]
   5 | Base::~Base() [deleting]

VTable indices for 'Base' (4 entries).
   0 | void Base::speak()
   1 | void Base::shout()
   2 | Base::~Base() [complete]
   3 | Base::~Base() [deleting]

Vtable for 'Derived' (6 entries).
   0 | offset_to_top (0)
   1 | Derived RTTI
       -- (Base, 0) vtable address --
       -- (Derived, 0) vtable address --
   2 | void Derived::speak()
   3 | void Base::shout()
   4 | Derived::~Derived() [complete]
   5 | Derived::~Derived() [deleting]

VTable indices for 'Derived' (3 entries).
   0 | void Derived::speak()
   2 | Derived::~Derived() [complete]
   3 | Derived::~Derived() [deleting]
```

```
.rodata:0000000000402058                public _ZTV7Derived ; weak
.rodata:0000000000402058 ; `vtable for'Derived
.rodata:0000000000402058 _ZTV7Derived   dq 0                    ;
.rodata:0000000000402058                                        ;
.rodata:0000000000402060                dq offset _ZTI7Derived  ;
.rodata:0000000000402068 off_402068     dq offset _ZN7Derived5spea
.rodata:0000000000402068                                        ;
.rodata:0000000000402068                                        ;
.rodata:0000000000402070                dq offset _ZN4Base5shoutEv
.rodata:0000000000402078                dq offset _ZN7DerivedD2Ev
.rodata:0000000000402080                dq offset _ZN7DerivedD0Ev
.rodata:0000000000402088                public _ZTS7Derived ; weak
.rodata:0000000000402088 ; `typeinfo name for'Derived
.rodata:0000000000402088 _ZTS7Derived   db '7Derived',0         ;
.rodata:0000000000402091                public _ZTS4Base ; weak
.rodata:0000000000402091 ; `typeinfo name for'Base
.rodata:0000000000402091 _ZTS4Base      db '4Base',0            ;
.rodata:0000000000402097                align 8
.rodata:0000000000402098                public _ZTI4Base ; weak
.rodata:0000000000402098 ; `typeinfo for'Base
.rodata:0000000000402098 _ZTI4Base      dq offset unk_403D50    ;
.rodata:0000000000402098                                        ;
.rodata:00000000004020A0                dq offset _ZTS4Base     ;
.rodata:00000000004020A8                public _ZTI7Derived ; weak
.rodata:00000000004020A8 ; `typeinfo for'Derived
.rodata:00000000004020A8 _ZTI7Derived   dq offset unk_403DA8    ;
.rodata:00000000004020B0                dq offset _ZTS7Derived  ;
.rodata:00000000004020B8                dq offset _ZTI4Base     ;
.rodata:00000000004020C0                public _ZTV4Base ; weak
.rodata:00000000004020C0 ; `vtable for'Base
.rodata:00000000004020C0 _ZTV4Base      dq 0                    ;
.rodata:00000000004020C0                                        ;
.rodata:00000000004020C8                dq offset _ZTI4Base     ;
```

RTTI(run-time type identification)

LLVM IR + Assembly Code

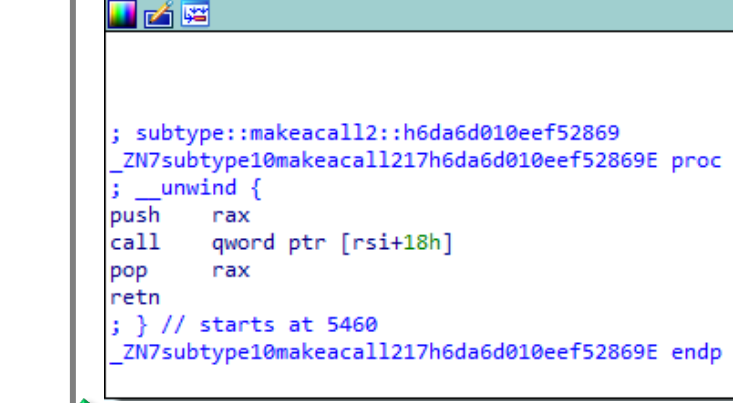
```
%class.Derived = type { %class.Base }
%class.Base = type { i32 (...)** }

  %1 = alloca %class.Derived, align 8
  %2 = alloca %class.Base*, align 8
  %3 = alloca i8*
  %4 = alloca i32
  call void @_ZN7DerivedC2Ev(%class.Derived* %1) #3
  %5 = bitcast %class.Derived* %1 to %class.Base*
  store %class.Base* %5, %class.Base** %2, align 8
  invoke void @_ZN7Derived5printEv(%class.Derived* %1)
          to label %6 unwind label %21
6:                                                ; preds = %0
  %7 = load %class.Base*, %class.Base** %2, align 8
  invoke void @_ZN4Base5printEv(%class.Base* %7)
          to label %8 unwind label %21
8:                                                ; preds = %6
  %9 = load %class.Base*, %class.Base** %2, align 8
  %10 = bitcast %class.Base* %9 to void (%class.Base*)***
  %11 = load void (%class.Base*)**, void (%class.Base*)*** %10, align 8
  %12 = getelementptr inbounds void (%class.Base*)*
        , void (%class.Base*)** %11, i64 0
  %13 = load void (%class.Base*)*, void (%class.Base*)** %12, align 8
  invoke void %13(%class.Base* %9)
          to label %14 unwind label %21
14:                                               ; preds = %8
  %15 = load %class.Base*, %class.Base** %2, align 8
  %16 = bitcast %class.Base* %15 to void (%class.Base*)***
  %17 = load void (%class.Base*)**, void (%class.Base*)*** %16, align 8
  %18 = getelementptr inbounds void (%class.Base*)*
        , void (%class.Base*)** %17, i64 1
  %19 = load void (%class.Base*)*, void (%class.Base*)** %18, align 8
  invoke void %19(%class.Base* %15)
          to label %20 unwind label %21
```

```
# %bb.0:pushq   %rbp
        movq    %rsp, %rbp
        subq    $48, %rsp
        leaq    -8(%rbp), %rax
        movq    %rax, %rdi
        movq    %rax, -40(%rbp)
        callq   _ZN7DerivedC2Ev
        movq    -40(%rbp), %rax
        movq    %rax, -16(%rbp)
.Ltmp0: movq    %rax, %rdi
        callq   _ZN7Derived5printEv
.Ltmp1: jmp     .LBB1_1
.LBB1_1:movq    -16(%rbp), %rdi
.Ltmp2: callq   _ZN4Base5printEv
.Ltmp3: jmp     .LBB1_2
.LBB1_2:movq    -16(%rbp), %rax
        movq    (%rax), %rcx
        movq    (%rcx), %rcx
.Ltmp4: movq    %rax, %rdi
        callq   *%rcx
.Ltmp5: jmp     .LBB1_3
.LBB1_3:movq    -16(%rbp), %rax
        movq    (%rax), %rcx
        movq    8(%rcx), %rcx
.Ltmp6: movq    %rax, %rdi
        callq   *%rcx
.Ltmp7: jmp     .LBB1_4
.LBB1_4:leaq    -8(%rbp), %rdi
        callq   _ZN7DerivedD2Ev
        xorl    %eax, %eax
        addq    $48, %rsp
        popq    %rbp
        retq
```

Rust Dyn Trait

- dyn Trait表示任意实现了trait的类型
  - 类似T:Base
  - 但当成动态类型编译，使用vtable寻址

```
trait A {
    fn a(&self) { println!("super a"); }
}
trait B : A{
    fn b(&self) { println!("sub b"); }
}
struct S { }

impl A for S { }
impl B for S { }

//fn makeacall1<T:A>(s: &T){ s.a() }
fn makeacall2(s: &dyn A){ s.a() }

fn main() {
  let s = S {};
  makeacall2(&s);
}
```



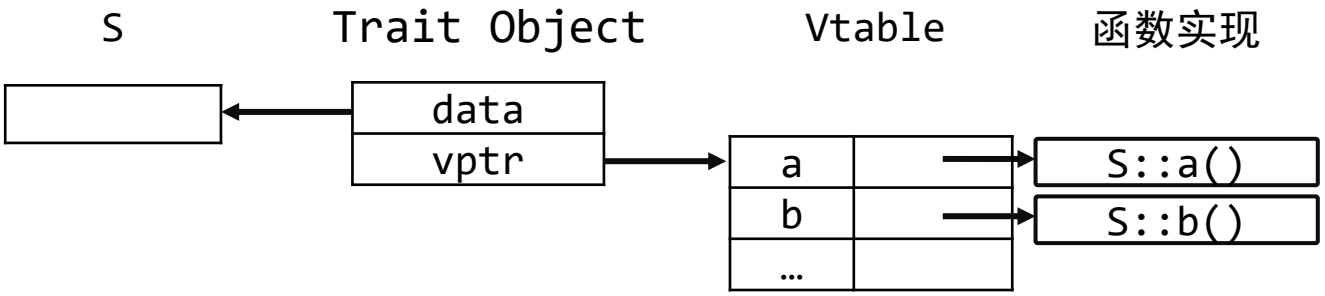
```
; subtype::makeacall1::hb802f2baed532665
_ZN7subtype10makeacall117hb802f2baed532665E proc n
; __unwind {
push    rax
call    _ZN7subtype4Base4base17h9f72e927683f8486E
pop     rax
retn
; } // starts at 5450
_ZN7subtype10makeacall117hb802f2baed532665E endp
```



```
; subtype::makeacall2::h6da6d010eef52869
_ZN7subtype10makeacall217h6da6d010eef52869E proc
; __unwind {
push    rax
call    qword ptr [rsi+18h]
pop     rax
retn
; } // starts at 5460
_ZN7subtype10makeacall217h6da6d010eef52869E endp
```

Dyn Trait vs 虚函数

- Dyn Trait只有一个vtable，不支持upcast



大纲

- 一、类型代码复用（inheritance）
- 二、统一函数接口（polymorphism）
- 三、动态派发
- 四、函数式编程
- 五、类型协变

函数式编程的特性

- 函数是一等公民（first class citizen）
  - 可用作变量赋值、参数传递、返回值
- 高阶函数
  - 函数可以作为参数或返回值，如g(x)=f(x)或y=f(g(x))
- 在命令式编程语言中：
  - C++ lambda calculus
  - Rust closure

函数类型

- Rust使用关键词fn作为函数类型标识

```
use std::fmt::Display;

fn main() {
    fn foo<T:Display>(x:T) { println!("{}",x); }
    let fn1 = &mut foo::<i32>;
    fn1(1);
    type Binop = fn(i32) -> ();
    let fn2:Binop = foo::<i32>;
    fn2(2);
}
```

使用函数作参数

```
fn add(a:i32, b:i32) -> i32 {
    a + b
}

fn hofn<F>(v1:i32, v2:i32, f: F) -> i32
    where F: Fn(i32, i32) -> i32 {
    f(v1,v2)
}

fn main() {
    hofn(1, 2, add);
}
```

函数参数

Rust Closure：匿名函数

- 自动捕获环境变量
- 参数传递：||

```
fn hofn<F>(v1:i32, v2:i32, f: F) -> i32
    where F: Fn(i32, i32) -> i32
{
    f(v1,v2)
}

fn main() {
    let i = 10;
    let cl = move |a, b| {a+b+i};
    let result = hofn(20, 10, cl);
}
```

a和b是参数，i是捕获变量

使用函数作返回值

```
fn hofn(len:u32) -> Box<dyn Fn(u32) -> u32> {
    let vec:Vec<u32> = (1..len).collect();
    let sum:u32 = vec.iter().sum();
    Box::new(move |x| {
        sum + x
    })
}

fn main() {
    hofn(10)(10);
}
```

Monad

- 函数式编程常用的一种模式
- 将返回值封装在含有功能代码的结构体中
  - wrap,unwrap
  - iter,map,contains...

```
enum Result<T, E> {
   Ok(T),
   Err(E),
}
```

```
fn foo(v:i32) -> Result<i32, &'static str> {
    match v {
        0 => Err("invalid header length"),
        _ => Ok(v),
    }
}

let r = foo(1);
match r {
    Ok(v) => println!("{v:?}"),
    Err(e) => println!("error: {e:?}"),
}
```

Monad: Rust Option

```
pub enum Option<T> {
    None,
    Some(T),
}
```

```
fn foo(v: int) -> Option<int> {
    match v {
        0 => None,
        _ => Some(v)
    }
}

let x = foo(1);
let y = match x {
    Some(x) => x,
    None    => 0,
};
```

高阶函数典型应用

- 通过Iterator实现容器的filter、map等功能

```
fn main() {
    let mut v:Vec<u32> = (1..100).collect();
    let iter1 = v.iter();
    let sum1:u32 = iter1().sum();

    let iter2 = v.iter().filter(|x| *x % 2 as u32 == 0);
    let sum2:u32 = iter2().sum();
    println!("sum = {:?}{:?}", sum1, sum2);

    let v2: Vec<_> = v1.iter().map(|x| x + 1).collect();
    println!(”v2 = {:?}", v2);
}
```

Iterator的优点

- 循环会做两次边界检测
  - 循环条件检查
  - 越界检查
- Iterator只检查一次

```
fn main() {
    let len = 1000000;
    let mut vec:Vec<usize> = (1..len).collect();
    let start = Instant::now();
    for i in vec.iter_mut(){
        *i += 1;
    }
    println!("{:?}", start.elapsed().as_nanos());

    let start = Instant::now();
    for i in 0..len-1 {
        vec[i] = vec[i]-1;
    }
    println!("{:?}", start.elapsed().as_nanos());
}
```

```
#:./iterator
14253222
57399993
```

函数式的优点：延迟计算

- 非必要，不计算
- 传统的控制流语句可以达到延迟计算的效果
  - `Lazy: If cond then y=eva(expr)`
  - `Eager: x=eva(expr), if(cond) then y=x`
- 函数式编程：
  - 变量=函数
  - 赋值时不会执行函数（evaluation）
  - 计算（evaluate）一次，永远使用
    - 有些语言模式默认采用该模式
    - 有些语言需要程序员手动实现，如Rust

延迟计算示例

```
use std::collections::HashMap;
use std::{thread,time};

fn main() {
    let mut hmap = HashMap::new();
    let mut insert = |x: i32| {
        println!("enter closure...");
        match hmap.get(&x) {
            Some(&val) => (),
            _ => {
                    thread::sleep(time::Duration::new(5,0));
                    hmap.insert(x, "123");
            }
        };
    };

    println!("Before insertion...");
    insert(1);
    println!("After the first insertion...");
    insert(1);
    println!("After the second insertion...");
}
```

大纲

- 一、类型代码复用（inheritance）
- 二、统一函数接口（polymorphism）
- 三、动态派发
- 四、函数式编程
- 五、类型协变

子类型关系是否会自动传播？

- 如果A是B的子类型，那么
  - A型数组和B型数组的关系？
  - List<A>和List<B>呢？

```
foo(b:B){
    ...
}

foo(l:List<B>){
    ...
}
```

协变关系：covariance

- 如果A是B的子类型，T<A>是T<B>的子类型
- 动态类型检查

```
//Java代码
String[] a = new String[1];
Object[] b = a;
b[0] = 1;
```

运行时报错

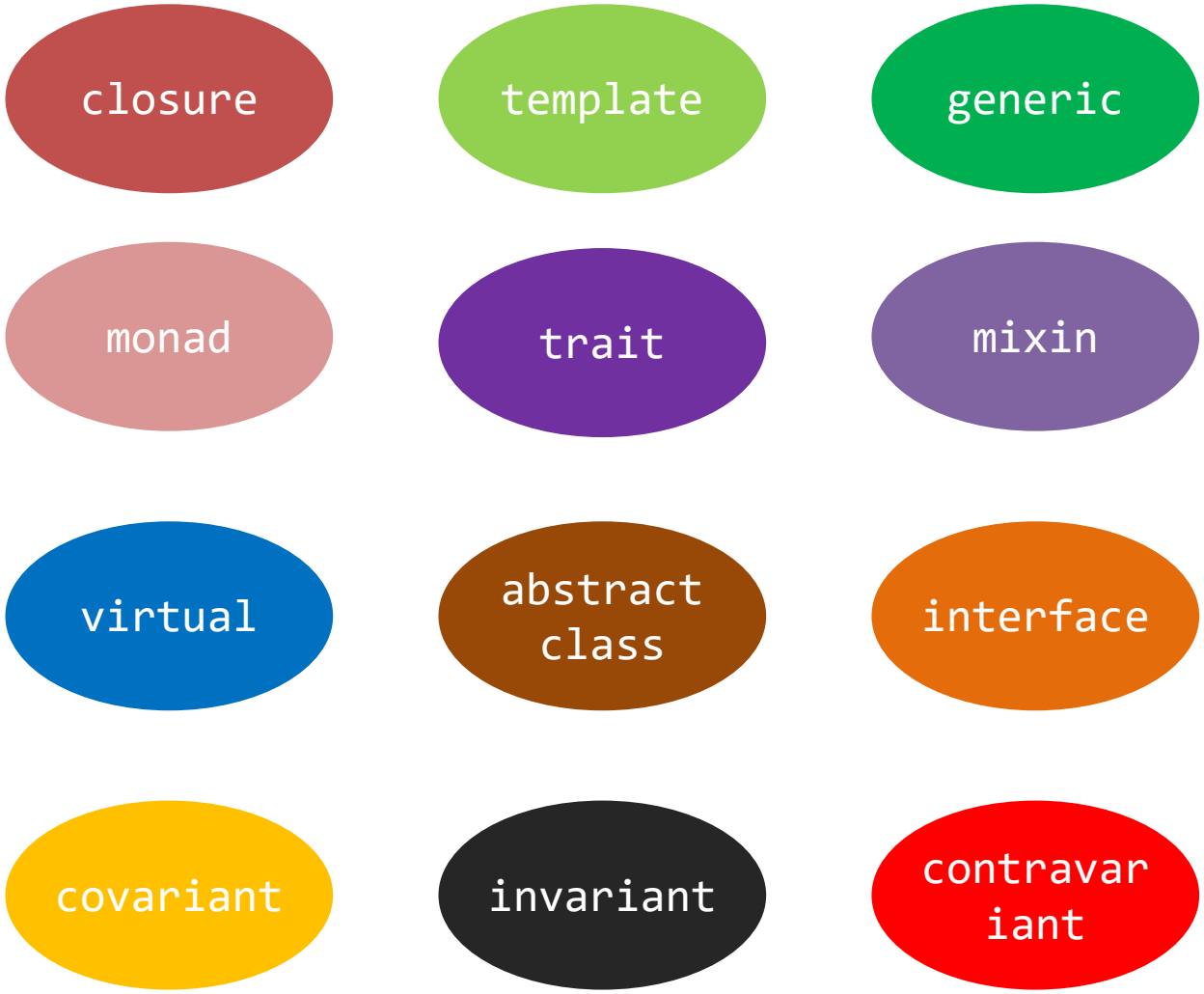
逆变关系：contravariance

- 如果A是B的子类型，T<A>是T<B>的子类型
- 典型逆变关系：函数参数

```
//Rust代码

fn test(f:fn(A)->()){
    ...
}

fn foo(a:A) {
    ...
}

fn bar(b:B) {
    ...
}

test(bar)
```

bar是foo的子类型

总结


closure
template
generic
monad
trait
mixin
virtual
abstract class
interface
covariant
invariant
contravariant