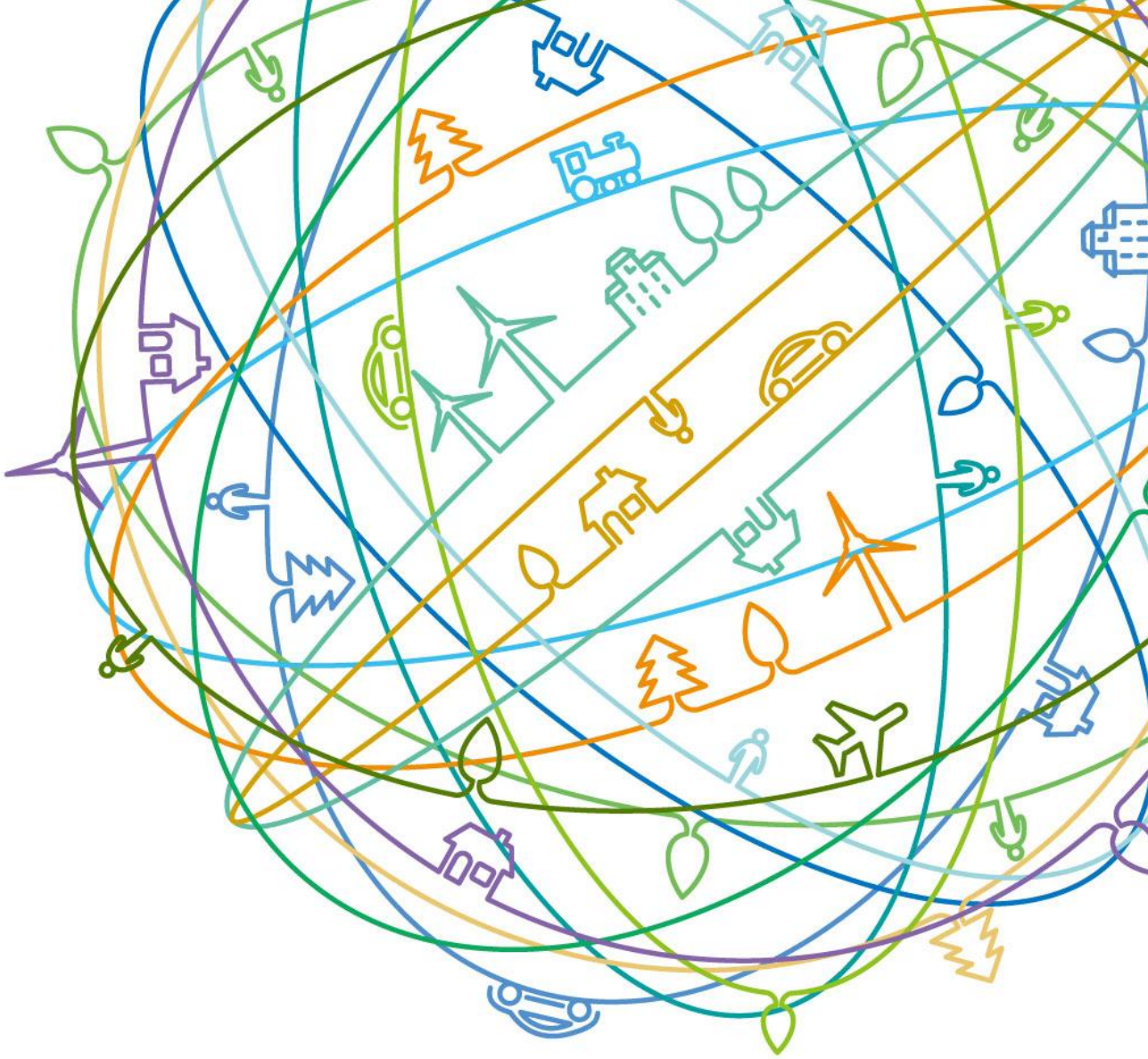
一些有趣的编程语言特性

中央软件院 喻钢/00219053



目录

这是一个有关编程的讲座，面向公司广大程序员。从语言的发展来看一些新的语言特性解决的问题，灵活使用，提供一些思路和大家一起探讨。

面向对象编程

```
class Fruit {
    // return int number of pieces of peel that
    // resulted from the peeling activity
        System.out.println("Peeling is appealing.");
    public int peel() {
        return 1;
    }
}

class Apple extends Fruit {

}

class Example1 {
    public static void main(String[] args) {
        Apple apple = new Apple();
        int pieces = apple.peel();
    }
}
```

```
class Peel {
    private int peelCount;
    public Peel(int peelCount) {
        this.peelCount = peelCount;
    }
    public int getPeelCount() {
        return peelCount;
    }
    //...
}
```

接口改变

```
class Fruit {
    // Return a Peel object that
    // results from the peeling activity.
    public Peel peel(){
        System.out.println("Peeling is appealing.");
        return new Peel(1);
    }
}

//Apple stills compiles and works fine
class Apple extends Fruit{

}

class Example1{
    public static void main(Str
        Apple apple = new Apple();
        int pieces = apple.peel();
    }
}
```

主函数无法编译器

父类接口改变，影响子类到父类的继承体系，子类接口调用也需要修改适配修改，变化的控制难度不小。

面向组合编程

```
class Fruit {
    // return int number of pieces of peel that
    // resulted from the peeling activity
        System.out.println("Peeling is appealing.");
    public int peel() {
        return 1;
    }
}

class Apple extends Fruit {

}

class Example1 {
    public static void main(String[] args) {
        Apple apple = new Apple();
        int pieces = apple.peel();
    }
}
```

```
class Peel {
    private int peelCount;
    public Peel(int peelCount) { ...
    }
    public int getPeelCount() { ...
    }
    //...
}

class Fruit {
    // Return a Peel object that
    // results from the peeling activity.
    public Peel peel(){ ...
    }
}

//Apple stills compiles and works fine
class Apple {
    private Fruit fruit = new Fruit();

    public int peel() {
        Peel peel = fruit.peel();
        return peel.getPeelCount();
    }
}

class Example1{
    public static void main(String[] args) {
        Apple apple = new Apple();
        int pieces = apple.peel();
    }
}
```

用组合取代继承

组合类失去一般性

“适配”类调用接口

组合类“适配”后端类的接口变更，实现“委托”和“隔离”，可以看做是一种修补和桥接。组合模式优于继承。

面向接口编程

```
package main
import "fmt"

type peel interface  {
   Peel() int
}
type fruit struct {
  int peel;
}
type apple struct {
  fruit;
}
func (p *apple) Peel() int {
   fmt.Println("Peeling is appealing");
   return 1;
}
func main() {
    var p apple;
    fmt.Println("apple peelcount is ", p.Peel());
}
```

显式定义的接口

无行为描述的对象实体

对象、接口、消息

```
package main
import "fmt"

type fruit struct {
  peelcount int;
}
type apple struct {
  fruit
}
type peel interface  {
   Peeleel() int
}
func (p *apple) Peel() int{
   fmt.Println("Peeling is appealing");
   return 1;
}
type peelcount interface  {
   Getpeelcount() int
   Setpeelcount(i int)
}
func (p *apple) Getpeelcount() int {
    fmt.Println("Peeling is peelcounting");
    return p.fruit.peelcount
}
func main() {
    var p apple;
    fmt.Println("apple peel is ", p.Peel());
    fmt.Println("apple peelount is ", p.Getpeelcount())
}
```

增加额外的接口描述，刻画对象行为

变更并不影响原有固定关系

接口和对象实体隔离定义，通过接口函数来隐式连接，实现了真正的面向接口编程，避免接口实现继承的尴尬。

接口上的多态

Go

数据和接口定义完全隔离，耦合度降至最低

```go
package main
import "fmt"

type mytype struct {i int}
type mychildtype struct { mytype; j int;}
type myInterface interface {
    Get() int
}
func (p *mytype) Get() int { return p.i }
func (p *mychildtype) Get() int { p.mytype.i++; return p.mytype.Get()}
func getit(x myInterface) { fmt.Println("object value is ", x.Get());}
func main() {
  var p mytype;
  var q mychildtype;
  getit(&p);
  getit(&q);
  getit(&p);
  getit(&q)
}
```

函数参数为接口，更一般的抽象。由具体的接口实现函数实现多态

Rust

```rust
#[derive(Default)]
struct Mytype { i:i32}
#[derive(Default)]
struct Mychildtype { x:Mytype, j:i32}
trait MyInterface {
    fn get(&mut self) -> i32 ;
}
impl MyInterface for Mytype {
    fn get(&mut self) -> i32 { self.i }
}
impl MyInterface for Mychildtype {
    fn get(&mut self) -> i32 { self.x.i += 1; self.j += 1; self.x.get()}
}
fn getit(x: &mut MyInterface) {println!("object value is {}", x.get())}
fn main() {
    let mut p:Mytype = Default::default();
    let mut q:Mychildtype = Default::default();
    getit(&mut p);
    getit(&mut q);
    getit(&mut p);
    getit(&mut q);
}
```

接口上定义的一般函数，将具体个性化的实现延迟到数据绑定的接口实现上，实现了真正的接口级多态。

目录

对象资源控制：让资源自行申请、释放

- 多线程情况下，对文件的写

“C”版本

```cpp
#include <mutex>
#include <iostream>
#include <string>
#include <fstream>
#include <stdexcept>

void write_to_file (const std::string & message) {
    // mutex to protect file access (shared across threads)
    static std::mutex mutex;

    // lock mutex before accessing file
    mutex.lock();

    // try to open file
    std::ofstream file;
    file.open("example.txt");

    if (!file.is_open())
        throw std::runtime_error("unable to open file");

    // write message to file
    file << message << std::endl;

    //close file
    file.close();
    mutex.unlock();
}
```

RAII版本

```cpp
#include <mutex>
#include <iostream>
#include <string>
#include <fstream>
#include <stdexcept>

void write_to_file (const std::string & message) {
    // mutex to protect file access (shared across threads)
    static std::mutex mutex;

    // lock mutex before accessing file
    std::lock_guard<std::mutex> lock(mutex);

    // try to open file
    std::ofstream file("example.txt");
    if (!file.is_open())
        throw std::runtime_error("unable to open file");

    // write message to file
    file << message << std::endl;

    // file will be closed 1st when leaving scope (regardless of exception)
    // mutex will be unlocked 2nd (from lock destructor) when leaving
    // scope (regardless of exception)
}
```

使用RAII（资源获取即初始化）原则管理栈上资源，堆上资源使用智能指针管理，避免重复多次释放。

对象资源控制：排他的写共享

并发的写共享，是一切“罪恶”的根源。很多编程语言解决思路是：控制“写”，但也有控制“共享”的思路

```
use std::thread;
use std::time::Duration;

fn main() {
    let mut data = vec![1, 2, 3];

    for i in 0..3 {
        thread::spawn(move || {
            data[i] += 1;
        });
    }

    thread::sleep(Duration::from_millis(50));
}
```

共享变量的”转移”，保持变量所有权的一致性

```
8:17 error: capture of moved value: `data`
        data[i] += 1;
        ^~~~
```

```
use std::thread;
use std::time::Duration;
use std::rc::Rc;

fn main() {
    let mut data = Rc::new(vec![1, 2, 3]);

    for i in 0..3 {
        // create a new owned reference
        let data_ref = data.clone();

        // use it in a thread
        thread::spawn(move || {
            data_ref[i] += 1;
        });
    }

    thread::sleep(Duration::from_millis(50));
}
```

引用不允许线程转移，非垃圾回收语言，保持变量引用计数归一

```
13:9: 13:22 error: the trait bound `alloc::rc::Rc<collections::vec::Vec<i32>> : core::mar
ker::Send`
            is not satisfied
...
13:9: 13:22 note: `alloc::rc::Rc<collections::vec::Vec<i32>>`
            cannot be sent between threads safely
```

对象资源控制：排他的写共享

```rust
use std::thread;
use std::sync::Arc;
use std::time::Duration;

fn main() {
    let mut data = Arc::new(vec![1, 2, 3]);

    for i in 0..3 {
        let data = data.clone();
        thread::spawn(move || {
            data[i] += 1;
        });
    }

    thread::sleep(Duration::from_millis(50));
}
```

```
<anon>:11:24 error: cannot borrow immutable borrowed content as mutable
<anon>:11                     data[i] += 1;
                              ^~~~
```

引用可以在线程间拷贝，但是不允许引用的写，无法确定“排他写”

```rust
use std::sync::{Arc, Mutex};
use std::thread;
use std::time::Duration;

fn main() {
    let data = Arc::new(Mutex::new(vec![1, 2, 3]));

    for i in 0..3 {
        let data = data.clone();
        thread::spawn(move || {
            let mut data = data.lock().unwrap();
            data[i] += 1;
        });
    }

    thread::sleep(Duration::from_millis(50));
}
```

在数据上绑定互斥，建立互斥的引用拷贝，每个线程在解锁互斥量，保证写访问的排他性

解决并发竞态问题的核心思想：保持资源的所有线程都是读访问，或者，仅有一个线程对共享资源写。

对象资源控制：排他的写共享（C++实现）

Rust

```rust
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let data = Arc::new(Mutex::new(vec![1, 2, 3]));
    let mut threads = vec![];

    for i in 0..3 {
        let data = data.clone();
        threads.push(thread::spawn(move || {
            let mut data = data.lock().unwrap();
            data[i] += 1;
        }));
    }

    for t in threads {
        t.join();
    }
    let data = data.lock().unwrap();
    for i in 0..3 {
        println!("{}", data[i])
    }

}
```

C++11

```cpp
#include <memory> //std::unique_ptr
#include <thread> //std::thread
#include <mutex>  //std::mutex
#include <vector> //std::vector
#include <iostream> //std::count


int main() {
  std::unique_ptr<std::vector<int>> v(new std::vector<int>{1,2,3});
  std::vector<std::thread> threads;
  std::mutex mutex;
  for(auto& elem: *v ) {
    threads.push_back(std::thread([&elem,&mutex] () mutable {
       std::lock_guard<std::mutex> lock(mutex);
       elem++;
    }));
  }
  for (auto& th : threads) th.join();
  for (auto& elem:*v) {
    std::cout << " " << elem << std::endl;
  }

  return 0;
}
```

语言共享安全的要素：读/写区分的变量申明定义，转移语义，数据上的互斥

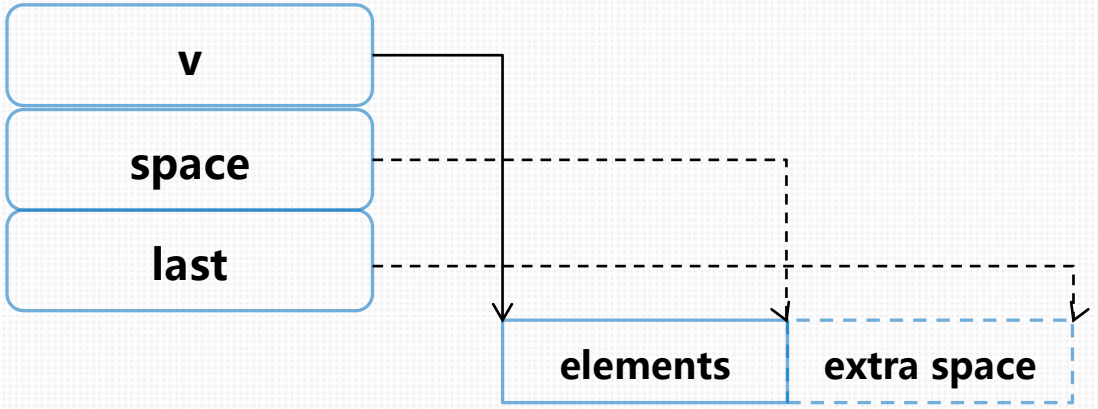
目录

异常的复杂性

软件系统（异常安全）稳定性设计要素：1. 任何情况下，不发生任何资源泄露
2. 关键操作的可恢复性（transaction)
3. 稳定操作的无异常性

- **异常系统并不容易设计**
  - RAII 原则可保证资源的不泄露
  - 关键操作的可恢复性并不容易

C++ vector




```
template<class T, class A>
vector<T, A>::vector(size_type n, const T& val, const A& a)
:alloc(a)
{
    v = alloc.allocate(n);

    iterator p;

    try {
        iterator end = v+n;
        for (p=v; p != end; ++p) alloc.construct(p, val);
        last = space = p;
    }
    catch (...) {
     for( iterator q = v; q != p; ++q) alloc.destroy(q);
     alloc.deallocate(v,n);
     throw;
    }

}
```

设计好的异常处理try-catch系统要保证资源不泄露，回退已完成的中间操作，定位具体的处理点，是一件复杂且棘手的问题

使用二值系统来简化控制流

- **调用程序执行错误，一般而言，只有两种处理措施：**
  - ✓ 函数可能发出异常，这种情况下，我们可以定义调用函数的范围类型为：Result<T,E>。
  - ✓ 程序已经发生严重错误了，无法恢复，必须退出，我们可以给出提示，直接退出 `panic!("boom");`

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

- **二值系统看起来简单，但是最直接问题是“啰嗦”**

```
fn write_info(info: &Info) -> io::Result<()> {
    let mut file = match File::create("my_best_friends") {
        Err(e) => return Err(e),
        Ok(f) => f,
    };
    if let Err(e) = writeln!(&mut file, "name: {}", info.name) {
        return Err(e)
    }
    if let Err(e) = writeln!(&mut file, "age: {}", info.age) {
        return Err(e)
    }
    if let Err(e) = writeln!(&mut file, "rating: {}", info.rating) {
        return Err(e)
    }
    if let Err(_) = writeln!(&mut file, "all is OK") {
        println!("final write failed");
    }
    return Ok(());
}
```

解决方案：宏和组合子(combinator)

- ✓ 有些操作，确定不会错
- ✓ 有些操作，基本确定不会错
  如果错了，直接退出程序，并给出提示
- ✓ 有些操作，需要连续操作下去，又不想写的累赘
- ✓ 有些操作，错了，但是我想恢复，或者给出默认值

流，配合组合子，实现表达式级的精准错误控制，大大方便异常调试

异常系统设计

较大规模的应用，异常需要设计：

- ✓ 扩展基本库异常，增强诊断能力或者提供新特性，定位和调试显得非常重要。
- ✓ 基本库已经在那了，我们需要重用而不是重写
- ✓ 正常的计算流程，不能轻易打断
- ✓ 提供灵活而清晰的接口，不能繁琐

我们以rust的包管理系统Cargo的异常系统设计为例

- 首先是扩展基本异常

```
pub type CargoResult<T> = Result<T, Box<CargoError>>;

// ============================================================================
// CargoError trait

pub trait CargoError: Error + Send + 'static {
    fn is_human(&self) -> bool { false }
    fn cargo_cause(&self) -> Option<&CargoError>{ None }
}

impl Error for Box<CargoError> {
    fn description(&self) -> &str { (**self).description() }
    fn cause(&self) -> Option<&Error> { (**self).cause() }
}

impl CargoError for Box<CargoError> {
    fn is_human(&self) -> bool { (**self).is_human() }
    fn cargo_cause(&self) -> Option<&CargoError> { (**self).cargo_cause() }
}
```

- 扩展异常上，组合实现新的接口

```
pub trait ChainError<T> {
    fn chain_error<E, F>(self, callback: F) -> CargoResult<T>
                          where E: CargoError, F: FnOnce() -> E;
}

#[derive(Debug)]
struct ChainedError<E> {
    error: E,
    cause: Box<CargoError>,
}
```

- 嵌入到传统接口，增强调试

```
impl<T, E: CargoError + 'static> ChainError<T> for Result<T, E> {
    fn chain_error<E2: 'static, C>(self, callback: C) -> CargoResult<T>
                          where E2: CargoError, C: FnOnce() -> E2 {
        self.map_err(move |err| {
            Box::new(ChainedError {
                error: callback(),
                cause: Box::new(err),
            }) as Box<CargoError>
        })
    }
}
```

应用代码模式

```rust
let mut s = String::new();
try!(f.read_to_string(&mut s).chain_error(|| {
    human(format!("failed to read file: {}", f.path().display()))
}));

(|| {
    let table = toml::Value::Table(try!(cargo_toml::parse(&s, f.path(), config)));
    let mut d = toml::Decoder::new(table);
    let v: resolver::EncodableResolve = try!(Decodable::decode(&mut d));
    Ok(Some(try!(v.to_resolve(pkg, config))))
}).chain_error(|| {
    human(format!("failed to parse lock file at: {}", f.path().display()))
})
```

- ✓ 基本库调用，系统接口未变
- ✓ 嵌入到正常逻辑中，增强了诊断
- ✓ 不打断正常执行逻辑，也不啰嗦
- ✓ 异常加强发生在异常路径上，不影响正常业务的性能

```rust
impl<'a, T, F> ChainError<T> for F where F: FnOnce() -> CargoResult<T> {
    fn chain_error<E, C>(self, callback: C) -> CargoResult<T>
                         where E: CargoError, C: FnOnce() -> E {
        self().chain_error(callback)
    }
}
```

异常系统需要设计，并满足四个要素。interface（traits）系统是设计后面的精髓。

目录

编写规格，让测试自动生成

- 单元测试很重要，咋做？

qsort in Haskell

业务执行函数

```
import Test.QuickCheck

-- | sorting algorithm
quicksort :: Ord a => [a] -> [a]
quicksort []     = []
quicksort (p:xs) = (quicksort lesser) ++ [p] ++ (quicksort greater)
    where
      lesser  = filter (< p) xs
      greater = filter (>= p) xs

业务检查函数

-- | check algorithm
ordered :: Ord a => [a] -> Bool
ordered [] = True
ordered [x] = True
ordered (x:y:ys) = (x <= y) && ordered (y:ys)

main :: IO()
main = do
  print $ quicksort ["123","15a","abc","14542","","456"]
  print $ quicksort [121,532,52525,3232131431,-56888,0,-55]
  quickCheck ((\s -> ordered $ quicksort s) :: [String] -> Bool)
  quickCheck ((\s -> ordered $ quicksort s) :: [Int] -> Bool)
```

随机测试框架

多型测试框架

>. 写完函数请尽早测试
>. 写测试规格，别写测试例，让系统自动生成测试例
>. 测试的规格也是函数
>. 越高规格的抽象才也越好写测试函数

一个实际检查的例子

Rust实例

```
extern crate quickcheck;
use quickcheck::quickcheck;

fn sort<T: Clone + Ord + Copy>(list: &[T]) -> Vec<T> {
  if list.is_empty() { vec![] }
  else {
      let (xs,ys):(Vec<T>,Vec<T>) = list.iter().partition(|&x| *x < list[0]);
      let mut result:Vec<T> = sort(&xs);
      result.extend(sort(&ys));
      result
  }
}

fn sorted_and_keeps_length(xs: Vec<isize>) -> bool {
   let ys = sort(&*xs);
   for win in ys.windows(2) {
     if win[0] > win[1] {
      return false
      }
   }
   xs.len() == ys.len()
}

fn main() {
    quickcheck(sorted_and_keeps_length as fn(Vec<isize>) -> bool);
}
```

```
xsymbol:quickcheck gang$ cargo run
   Running `target/debug/quickcheck`

thread 'safe' has overflowed its stack
fatal runtime error: stack overflow
error: An unknown error occurred
```

```
extern crate quickcheck;
use quickcheck::quickcheck;

fn sort<T: Clone + Ord + Copy>(list: &[T]) -> Vec<T> {
  if list.is_empty() { vec![] }
  else {
      let (xs,ys):(Vec<T>,Vec<T>) = list[1..].iter().partition(|&x| *x < list[0]);
      let mut result:Vec<T> = sort(&xs);
      result.extend(sort(&ys));
      result
  }
}

fn sorted_and_keeps_length(xs: Vec<isize>) -> bool {
   let ys = sort(&*xs);
   for win in ys.windows(2) {
     if win[0] > win[1] {
      return false
      }
   }
   xs.len() == ys.len()
}

fn main() {
    quickcheck(sorted_and_keeps_length as fn(Vec<isize>) -> bool);
}
```

```
xsymbol:quickcheck gang$ cargo run
   Running `target/debug/quickcheck`
thread '<main>' panicked at '[quickcheck] TEST FAILED. Arguments: ([0])',
/Users/gang/.cargo/registry/src/github.com-88ac128001ac3a9a/quickcheck-0.2.27/src/tester.rs:116
```

quickcheck自动测试的真正难度在于编写程序的规格（不动点），一旦形成规格，函数的问题还是很快可以暴漏的。

目录

模式匹配，让程序自己匹配

- **一个编程能力测试题：**
  - 写一个程序，去除C/C++源码中的注释
  - 一个著名IT公司的主任工程师笔试上机题
  - 40分钟内完成
  - 单元测试、调试和发布
  - 产品级应用
- **我们该咋做？**

C语言解决方案

- 很自然地，我们想到C的解决方案

```
int tranfer(FILE * in, FILE *out) {
    char c;
    while((c = fgetc(in)) != EOF) {
        if (c == '/') {
            ...
        } else if ( c == '*') {
            ...
        } else {
            ...
        }
    ...
    }
}
```

我们能20分钟内写完程序，15分钟完成调试？

模式匹配语言的解决方案

Haskell解法(list comprehension)

```haskell
module Implement (
  stripped_code,
) where
data R_state = In_Code | In_C_comment | In_Cpp_comment | In_String
stripped_code :: String -> String
stripped_code orig = g1 In_Code [] orig where
g1 _ readed [] = readed
g1 In_Code readed ('/':'/':xs) = g1 In_Cpp_comment readed xs
g1 In_Code readed ('/':'*':xs) = g1 In_C_comment readed xs
g1 In_Code readed ('"':xs) = g1 In_String (readed++['"']) xs
g1 In_Code readed (x:xs) = g1 In_Code (readed++[x]) xs
g1 In_String readed ('"':xs) = g1 In_Code (readed++['"']) xs
g1 In_String readed (x:xs) = g1 In_String (readed++[x]) xs
g1 In_C_comment readed ('*':'/':xs) = g1 In_Code readed xs
g1 In_C_comment readed (x:xs) = g1 In_C_comment readed xs
g1 In_Cpp_comment readed ('\n':xs) = g1 In_Code (readed++['\n']) xs
g1 In_Cpp_comment readed (x:xs) = g1 In_Cpp_comment readed xs
```

模式匹配，很好很强大？

Rust解法(常规)

```rust
fn stripped_code(source:&[u8], result:&mut Vec<u8>) {
  let mut status = Code::InCode;
  let mut next_skip = false;
  if source.is_empty() {}
  else if source.len() == 1 {
    result.push(source[0]);
  }
  else {
   for s in source.windows(2) {
       let z = (s[0] as char, s[1] as char);
       if !next_skip {
           match z {
               ('/' , '/') if status == Code::InCode =>
           { status = Code::InCppComment; next_skip = true; }
               ('/', '*') if status ==  Code::InCode =>
           {status = Code::InCComment; next_skip = true; }
               ('"', _) if status == Code::InCode =>
           { status = Code::InString; result.push(s[0]); next_skip = false; }
               (_, _) if status ==  Code::InCode =>
           { result.push(s[0]); next_skip = false; }
               ('"', _) if status == Code::InString =>
           { status = Code::InCode; result.push(s[0]); next_skip = false; }
               (_, _) if status == Code::InString =>
           { result.push(s[0]); next_skip = false; }
               ( '*', '/') if status == Code::InCComment =>
           { status = Code::InCode; next_skip = true; }
               (_, _) if status ==  Code::InCComment =>
           { next_skip = false; }
               ('\n', _) if status == Code::InCppComment =>
           { status = Code::InCode; next_skip = false; }
               (_, _) if status == Code::InCppComment =>
           { next_skip = false; }
               (_, _) => {}
           }
       } else {
           next_skip = false;
       }
   }
   if status == Code::InCode || status == Code::InString {
       result.push(*source.last().unwrap());
   }
  }
}
```

模式匹配的问题

- 结果： 10分钟就写完，15分钟还没跑完☹
- 太慢了，怎么办？

- Profile

  ghc –make –prof main.hs Common.hs Implement.hs

  ./p1 +RTS –hc –sstderr -RTS

Profiling结果

```
2,731,445,588 bytes allocated in the heap
  2,247,346,036 bytes copied during GC
  8,927,768 bytes maximum residency (256 sample(s))
  5,297,344 bytes maximum slop
 25 MB total memory in use (0 MB lost due to fragmentation)

Generation 0: 2080 collections, 0 parallel, 4.94s, 4.92s elapsed
Generation 1: 256 collections, 0 parallel, 2.42s, 2.52s elapsed

INIT time 0.00s ( 0.00s elapsed)
MUT time 1.80s ( 1.95s elapsed)
GC time 7.36s ( 7.44s elapsed)
RP time 0.00s ( 0.00s elapsed)
PROF time 0.09s ( 0.09s elapsed)
EXIT time 0.00s ( 0.00s elapsed)
Total time 9.25s ( 9.48s elapsed)

%GC time 79.5% (78.5% elapsed)

Alloc rate 1,514,011,096 bytes per MUT second

Productivity 19.5% of total user, 19.0% of total elapsed
```

Profiling分析

- GC太费时了
- 咋办？ --》（++）好用不？
- a:xs 快过 xs++[a]，链表的构造是正向的

```
module Implement (
  stripped_code,
) where
data R_state = In_Code | In_C_comment | In_Cpp_comment | In_String
stripped_code :: String -> String
stripped_code orig = g1 In_Code [] orig where
g2 _ readed [] = reverse readed
g2 In_Code readed ('/':'/':xs) = g2 In_Cpp_comment readed xs
g2 In_Code readed ('/':'*':xs) = g2 In_C_comment readed xs
g2 In_Code readed ('"':xs) = g2 In_String ('"':readed) xs
g2 In_Code readed (x:xs) = g2 In_Code (x:readed) xs
g2 In_String readed ('"':xs) = g2 In_Code ('"':readed) xs
g2 In_String readed (x:xs) = g2 In_String (x:readed) xs
g2 In_C_comment readed ('*':'/':xs) = g2 In_Code readed xs
g2 In_C_comment readed (x:xs) = g2 In_C_comment readed xs
g2 In_Cpp_comment readed ('\n':xs) = g2 In_Code ('\n':readed) xs
g2 In_Cpp_comment readed (x:xs) = g2 In_Cpp_comment readed xs
```

```
20,875,372 bytes allocated in the heap
11,484,236 bytes copied during GC
2,335,096 bytes maximum residency (3 sample(s))
109,996 bytes maximum slop
7 MB total memory in use (0 MB lost due to fragmentation)

Generation 0: 36 collections, 0 parallel, 0.01s, 0.01s elapsed
Generation 1: 3 collections, 0 parallel, 0.01s, 0.01s elapsed

INIT time 0.00s ( 0.00s elapsed)
MUT time 0.02s ( 0.22s elapsed)
GC time 0.02s ( 0.02s elapsed)
RP time 0.00s ( 0.00s elapsed)
PROF time 0.00s ( 0.00s elapsed)
EXIT time 0.00s ( 0.00s elapsed)
Total time 0.04s ( 0.24s elapsed)
%GC time 36.4% (9.5% elapsed)
Alloc rate 745,522,374 bytes per MUT second
Productivity 54.5% of total user, 9.9% of total elapsed
```

对强类型语言，静态编译可以帮助我们解决很多问题，但是动态profiling的能力依旧十分需要，语言的DFX设计也是重要一环。

小结

- 编程并不是一个容易的事情，尤其想写兼顾性能、安全、调测、结构的好代码，语言可以提供很多的帮助，核心还是程序员自己的*思考*
- 写抽象的程序，抽象的层次越高，说明对事物的本质思考越彻底。
- 数据和行为应尽可能分开定义，在接口上建立联系，接口保持稳定。
- 并发安全的思路是保持并发数据的写排他，读共享
- 异常设计要简化，安全是首要问题。建立在二值系统上的控制流合一看起来是有效思路，但仍旧需要其他语言成分的支持，形成一个整体。
- 测试从“小”从“早”做起，自底而上的写程序，自底而上的写测试。思考程序的规格，确保规格来自动测试。如果无法写规格，写断言。

Thank you

www.huawei.com