计算概论A—实验班

函数式程序设计

胡振江，张 伟

北京大学 信息科学技术学院 计算机科学技术系

2021年09～2021年12月

函数式编程思想
在其他编程语言中的应用

0	Introduction
1	FP in C++ 2020
2	FP in JavaScript

以下内容

不会出现在

本课程的任何考试/作业中

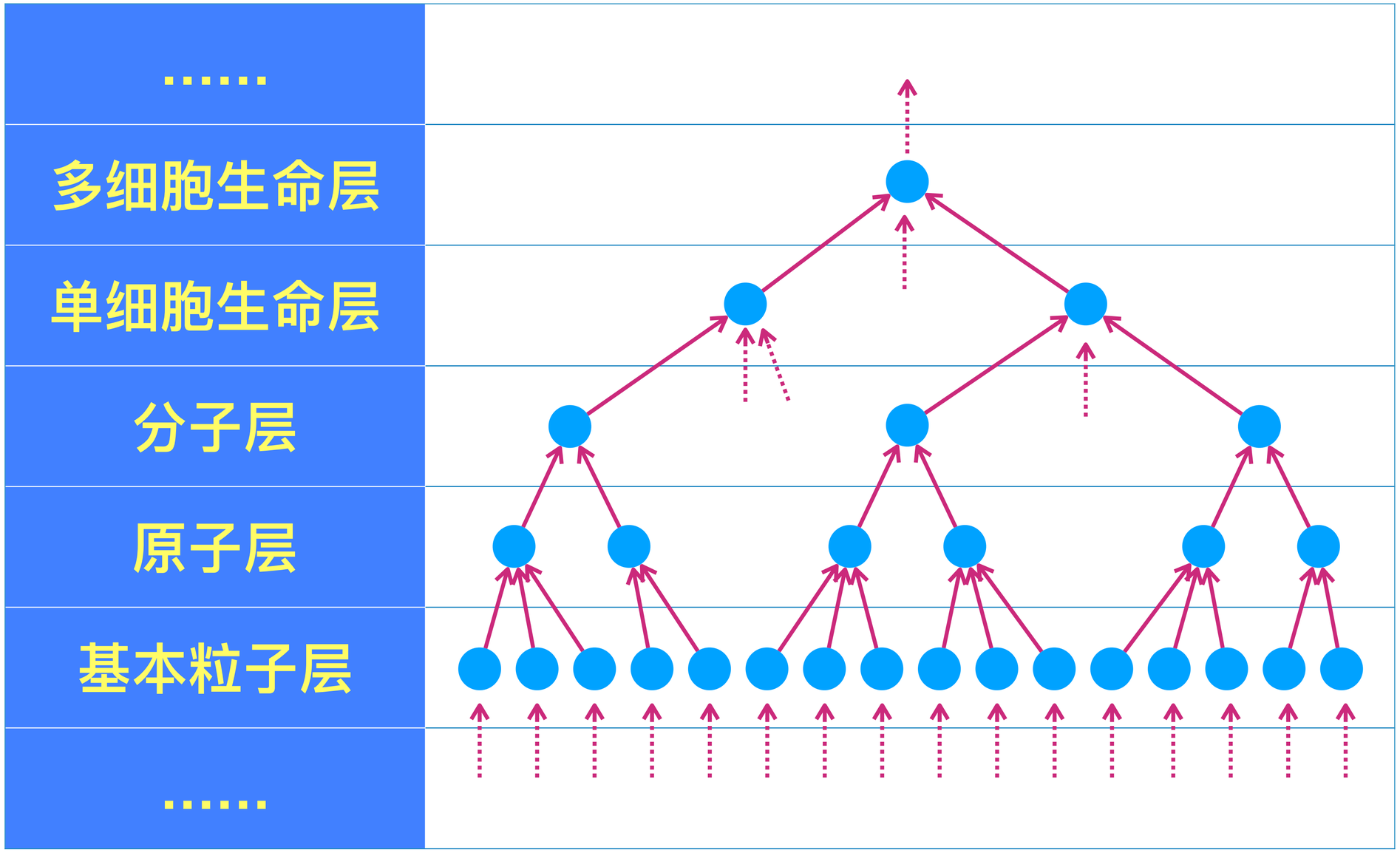
0. Introduction

Haskell类型系统的一种理解方式

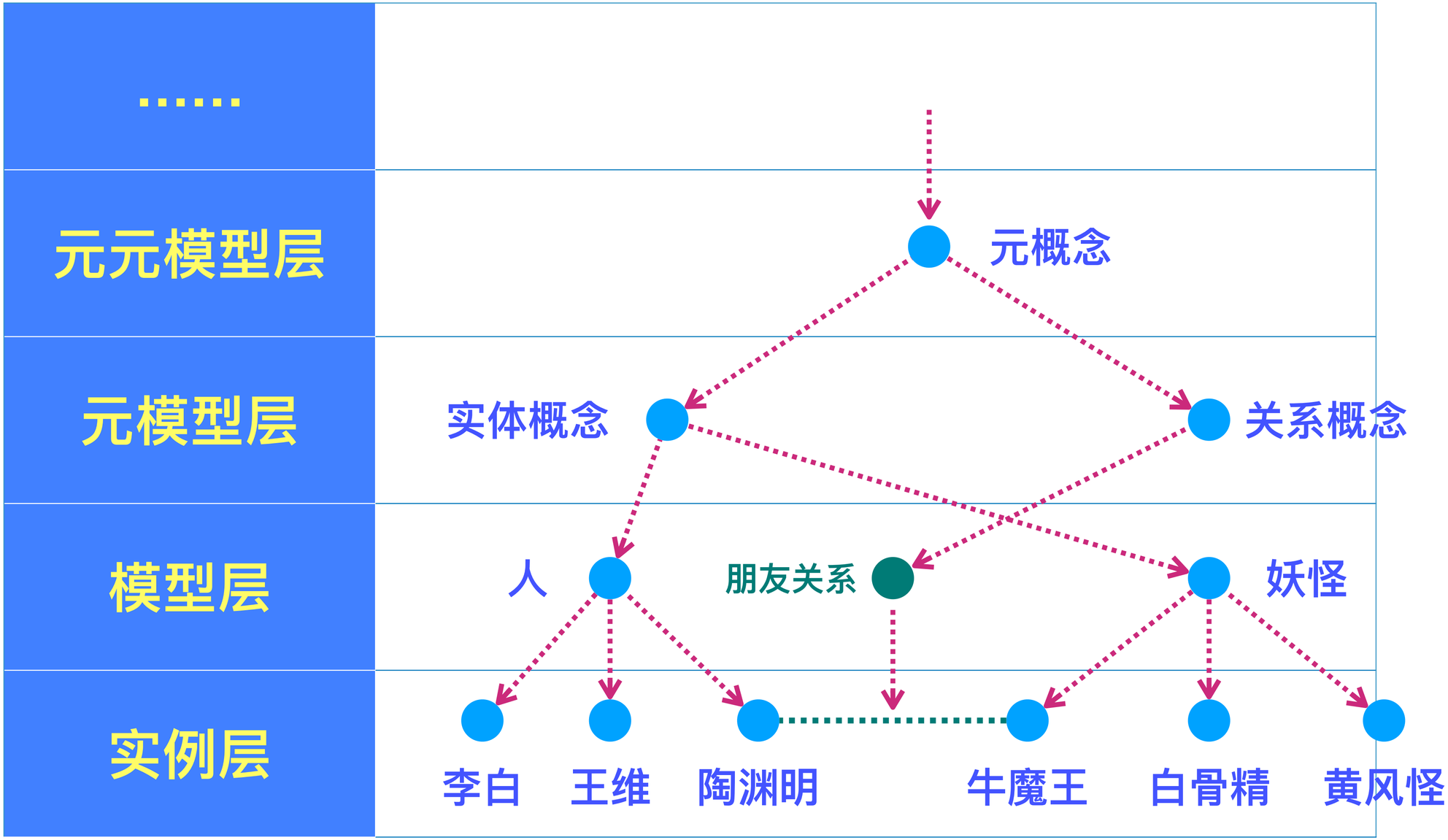
Haskell类型系统的一种理解方式

我们的周围存在两种层次性	
物理空间中的 一种层次性	思维空间中的 一种层次性

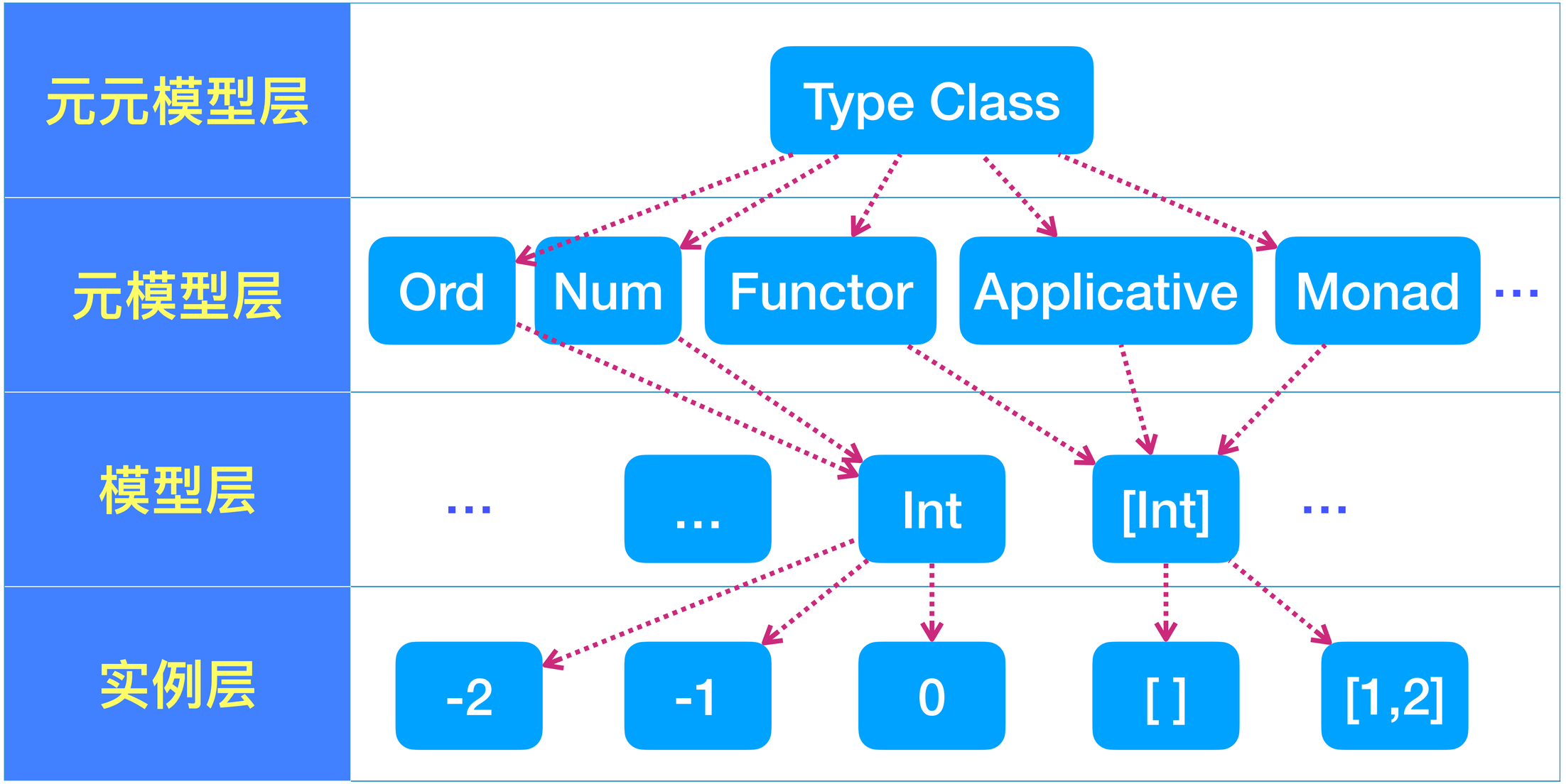
物理空间：基于物质的组成关系形成的层次性



思维空间：基于概念外延与概念实例的实例化关系形成的层次性



Haskell的类型系统涉及四个层次



关于程序设计语言的一种观点

关于程序设计语言的一种观点

每一种语言 都或多或少反映了 某种关于信息处理的世界观
在我们还没有真正理解信息处理技术的终极真理之前， 每一种世界观都应该被认真对待
在实际软件开发活动中，要根据实际情况， 灵活选择一或多种世界观，实现对当前问题的有效求解

在软件领域中，世界观 也被称为 范型（Paradigm）

几种 经典世界观/范型 及 代表性语言

ID	世界观	代表性语言	特点
1	函数式编程	Haskell	• 以数学函数作为表示问题的基本方式 • 以问题分解作为求解问题的基本手段 • 以数学家书写数学公式的方式进行编程 • 避免程序员手工管理程序执行的中间状态
2	面向类的编程	经典C++语言（部分）	• 以类作为组织和复用代码的基本方式 • 通过类的实例化产生对象(object)：一个对象中封装了一组状态以及对状态的修改操作
3	面向对象的编程	经典JavaScript语言	• **没有**类，**只有**对象 • 通过基于原型的继承（prototype-based inheritance）实现对象功能的灵活复用

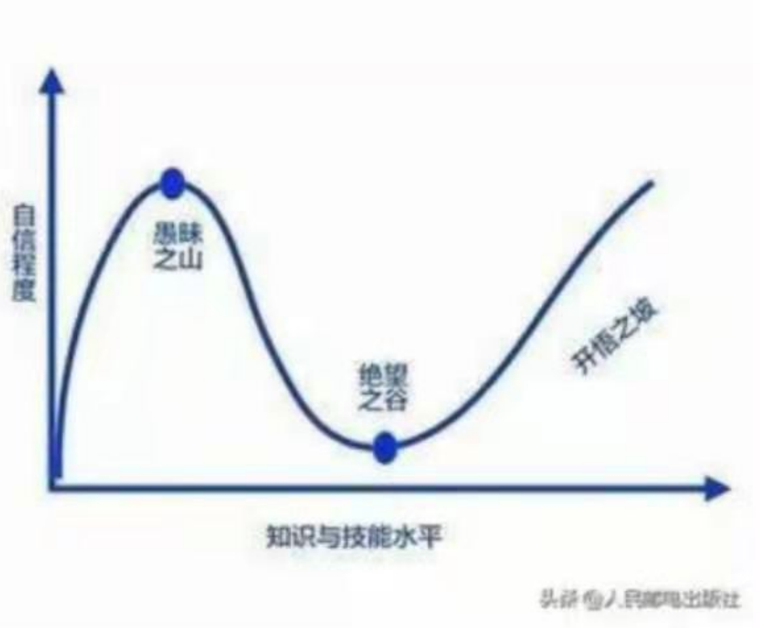
Haskell、C++、JavaScript 人设

Haskell	C++	JavaScript
出身名门 阳春白雪 象牙塔	在实践中发展 兼容并包 / 万金油 沉重的历史包袱	出身草莽 家道中落 早产儿 王侯将相、宁有种乎
• 一群具有良好数学基础的科学家在象牙塔中玩票的产物。 • 他们只关注理论的纯洁性，似乎并不关注实用性	• 以工程师为主导 • 只要能解决实际问题，任何特性都可以加入到语言中 • 学习成本太高了	• Web领域中的汇编语言 • 客户端、服务器端通吃 • 如果想了解详情，可以选修下学期的课程《JavaScript语言Web程序设计》 • 也可随时访问这门课的在线讲义：https://www.yuque.com/nrutas/js

30年沉淀——最复杂的语言

如果10分满分，我对C++的了解是7分。——Bjarne Stroustrup

有一个专属于C++的梗，即当你自称“精通C++”时，你一定还处于学习的愚昧之峰。



你是在愚昧之峰还是开悟之坡?

C++有着许多独立于其他语言的特性，但随着语言成熟度、兼容性以及稳定性而来的，是语言的复杂性。

编程语言的一种发展趋势

主流编程语言都在向 多范型 发展

一个人的智力是否属于上乘

要看其头脑中能否同时容纳两种相反的思想而无碍其处世行事

函数式范型逐渐渗透到各种语言中

0. Introduction

THE END

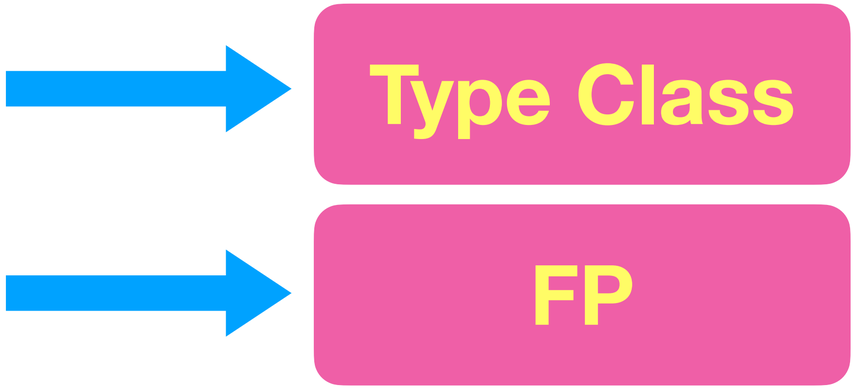
1. FP in C++ 2020

C++语言的年龄比我们都大
为什么它还这么拼?

后浪太汹涌

C++ 2020 四大新特性（The Big Four）

1	Concept
2	Range
3	Coroutine
4	Module



其中，两个特性

与Haskell具有良好的对应关系

下面，我们重点介绍 Range 这一概念

一个方便的C++在线编程环境 https://wandbox.org

C++

gcc HEAD 11.0.0 20201

Load template

- [x] Warnings
- [ ] Optimization
- [ ] Verbose

Boost 1.73.0

- [ ] Sprout
- [ ] MessagePack

C++2a(GNU)

no pedantic

Compiler options:

Runtime options...

Corporate Sponsors:

セオライド・テクノロジー㈱
株式会社フィックスターズ
株式会社ドットインストール
BASSDRUM株式会社

Personal Sponsors:

voluntas @ignis_fatuus ブン
@Linda_pp
清楚なC++メイドBOT
@tzik_tack 長谷川一輝 wraith13

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    string name;

    cin >> name;

    cout << "Hello, " << name << ".";

}
```

```
$ g++ prog.cc -Wall -Wextra -I/opt/wandbox/boost-1.73.0/gcc-head/include -std=gnu++2a
```

Stdin

```
Dog
```

Run (or Ctrl+Enter)

#1

Share by

Code

```
Start

Hello, Dog.

0

Finish
```

Editor settings:

editor: default

tab: 4-spaces

tab width: 4

- [x] Smart Indent
- [ ] No Wrap

C++语言中的Lambda函数

函数

函数是数学中的一个基本概念
$y = f(x_1, x_2, x_3, \ldots, x_n)$
若干自变量 与 一个因变量 之间 的 因果关系
给定自变量的一种取值 会得到一个唯一确定的因变量取值

C++语言提供了若干种机制

使得程序员可以定义数学意义上的函数及其变体

Lambda函数是其中的一种

Lambda函数: 示例

计算两个整数的和

```
1  #include <iostream>
2
3  int main()
4  {
5    int a, b, c;
6
7    std::cin >> a >> b;
8
9    c = a + b;
10
11   std::cout << c;
12 }
```

如何把“两个整数相加”的功能
封装在一个函数里

计算两个整数的和-Lambda版本

```
1  #include <iostream>
2
3  auto add = [](int x, int y)
4  {
5    return x + y;
6  };                          函数定义
7
8  int main()
9  {
10   int a, b, c;
11
12   std::cin >> a >> b;
13
14   c = add(a, b);            函数调用
15
16   std::cout << c;
17 }
```

Lambda函数: 示例 详细说明

计算两个整数的和-Lambda版本

```
1  #include <iostream>
2
3  auto add = [](int x, int y)
4  {
5    return x + y;
6  };
7
8  int main()
9  {
10   int a, b, c;
11
12   std::cin >> a >> b;
13
14   c = add(a, b);
15
16   std::cout << c;
17 }
```

也可以给出明确的类型

std::function< int (int, int)>

此时，需要添加#include<functional>

```
auto add
```

lambda函数定义完成后
形成了一个值
这个值被赋值给变量add

```
[](int x, int y)
{
  return x + y;
}
```

严格意义上，这才是lambda函数的定义

这是对lambda函数的调用

调用的行为大致等价于以下代码

```
int x = a, y = b;
return x + y;
```

Lambda函数: 示例

计算两个整数的和-Lambda版本

```
1  #include <iostream>
2
3  auto add = [](int x, int y)
4  {
5    return x + y;
6  };
7
8  int main()
9  {
10   int a, b, c;
11
12   std::cin >> a >> b;
13
14   c = add(a, b);
15
16   std::cout << c;
17 }
```

如果我想定义
“两个浮点数相加”的函数呢

这算是问题吗

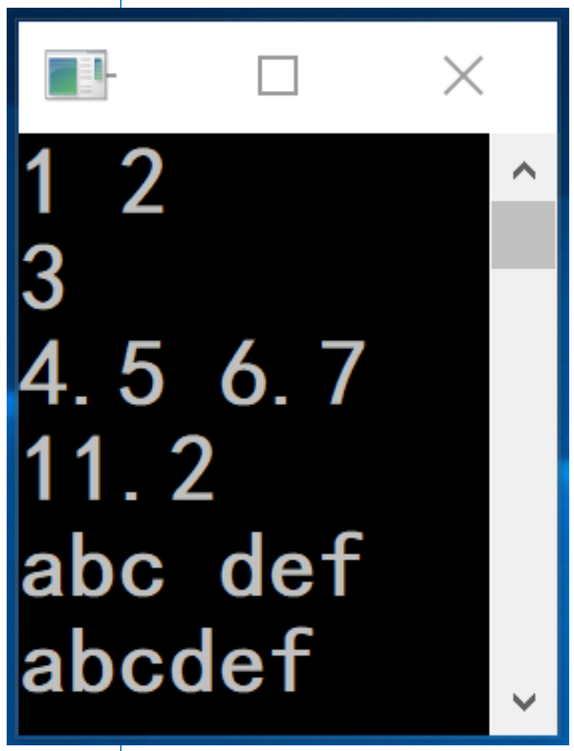
```
auto add_d = [](double x, double y)
{
  return x + y;
};
...
double c = add_d(1.2, 3.4);
```

两个字：**我懒**

Lambda函数示例: 两个东西相加

编译器会在函数调用时
自动判断出传入参数的类型

```
1  #include <iostream>
2  #include <string>
3
4  auto add = [](auto x, auto y){ return x + y; };
5
6  int main()
7  {
8    int a, b; std::cin >> a >> b;
9
10   std::cout << add(a, b) << "\n";
11
12   double u, v; std::cin >> u >> v;
13
14   std::cout << add(u, v) << "\n";
15
16   std::string p, q; std::cin >> p >> q;
17
18   std::cout << add(p, q) << "\n";
19
20 }
```



C++语言入门

Range

开胃小菜

Range让C++中的排序更简洁

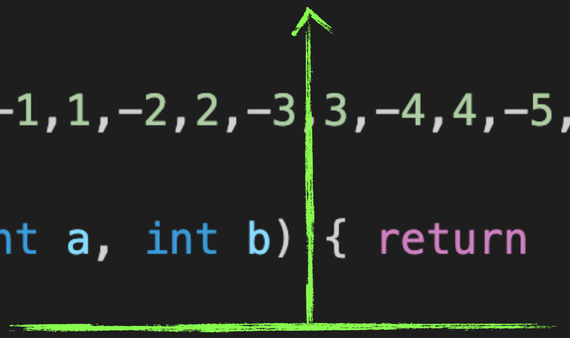
```cpp
#include <vector>
#include <ranges>
#include <algorithm>
#include <iostream>

using namespace std;

int main()
{
    vector<int> ints{0,-1,1,-2,2,-3,3,-4,4,-5,5};

    auto compare = [](int a, int b) { return  a < b; };

    ranges::stable_sort(ints, compare);

    for(auto x : ints) cout << x << " ";

    cout << "\n";
}
```

```
-5 -4 -3 -2 -1 0 1 2 3 4 5
```

Range让C++中的排序更简洁

```cpp
#include <vector>
#include <ranges>
#include <algorithm>
#include <iostream>

using namespace std;

int main()
{
    vector<int> ints{0,-1,1,-2,2,-3,3,-4,4,-5,5};

    auto compare = [](int a, int b) { return  a < b; };

    ranges::stable_sort(ints | views::drop(3), compare);

    for(auto x : ints) cout << x << " ";

    cout << "\n";
}
```

等价于：views::drop(ints, 3)

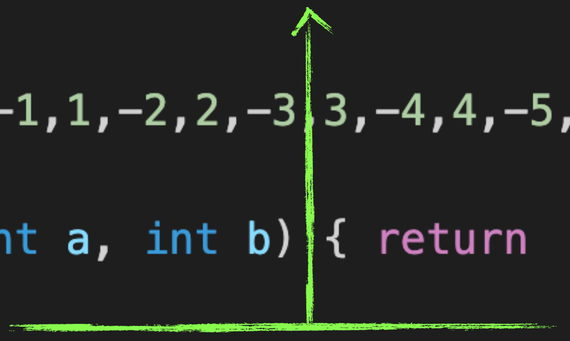


0 -1 1 -5 -4 -3 -2 2 3 4 5

Range让C++中的排序更简洁

```cpp
#include <vector>
#include <ranges>
#include <algorithm>
#include <iostream>

using namespace std;

int main()
{
    vector<int> ints{0,-1,1,-2,2,-3,3,-4,4,-5,5};

    auto compare = [](int a, int b) { return  a < b; };

    ranges::stable_sort(ints | views::take(5), compare);

    for(auto x : ints) cout << x << " ";

    cout << "\n";
}
```

等价于：views::take(ints, 5)

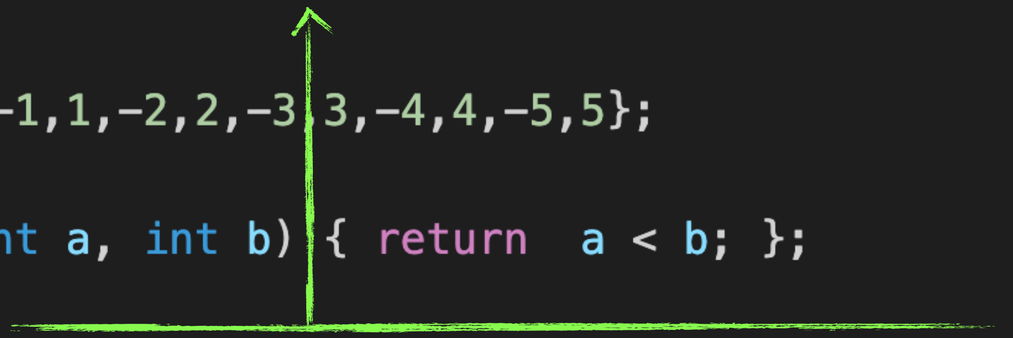


-2 -1 0 1 2 -3 3 -4 4 -5 5

Range让C++中的排序更简洁

```cpp
#include <vector>
#include <ranges>
#include <algorithm>
#include <iostream>

using namespace std;

int main()
{
    vector<int> ints{0,-1,1,-2,2,-3,3,-4,4,-5,5};

    auto compare = [](int a, int b) { return  a < b; };

    ranges::stable_sort(ints | views::drop(3) | views::take(6), compare);

    for(auto x : ints) cout << x << " ";

    cout << "\n";
}
```

等价于：`views::take(views::drop(ints, 3), 6)`



0 -1 1 -4 -3 -2 2 3 4 -5 5

另一个例子

```cpp
#include <vector>
#include <ranges>
#include <iostream>

using namespace std;

int main()
{
    vector<int> ints = {1, 2, 3, 4, 5, 6};

    auto rst = ints | views::filter([](int n){ return n % 2 == 0; })
                    | views::transform([](int n){ return n * 2; });

    for (auto v: rst) cout << v << " "; // 4 8 12

    cout << "\n";
}
```

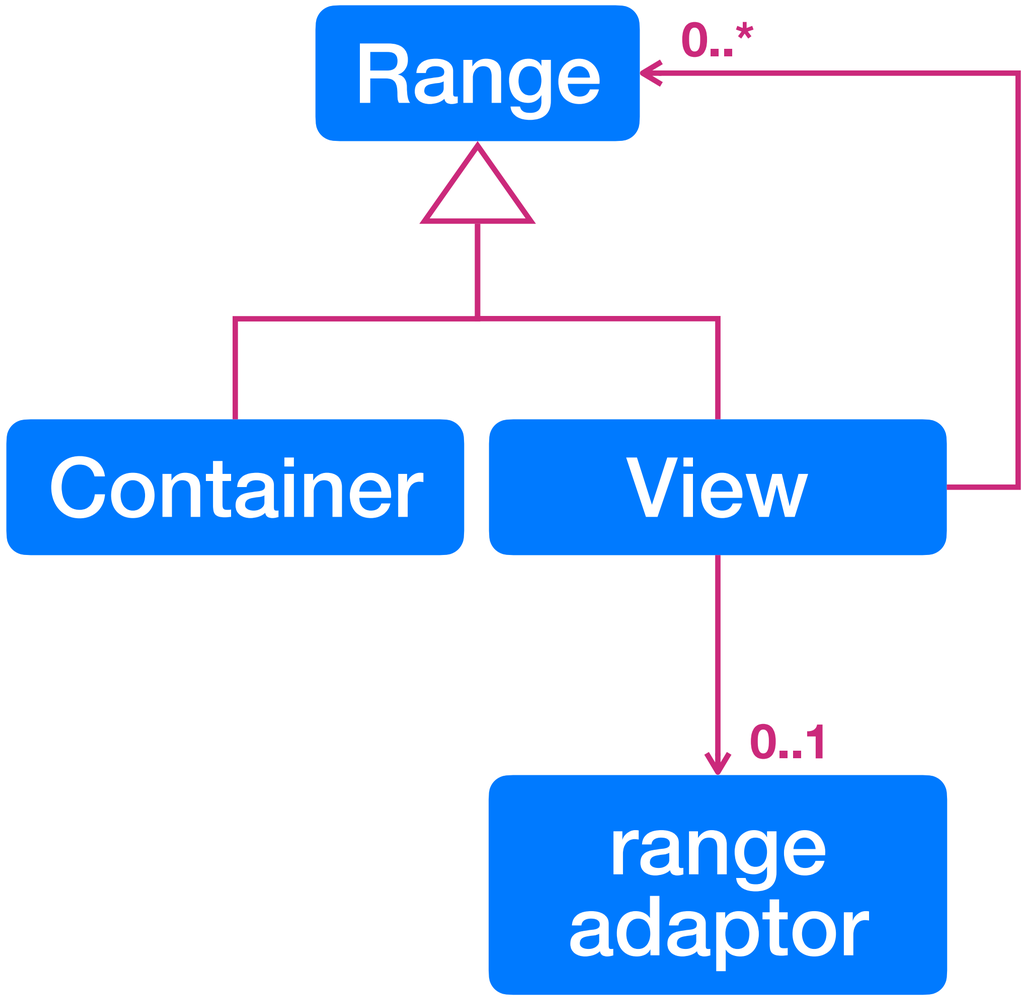
管道操作符 | ：函数组合 (function composition)

```
vec | foo | bar(3) | baz(7)
```


```
baz( bar( foo(vec), 3), 7 )
```

Range到底是什么？

Range到底是什么?



Range	a collection of iterable items with the same type
	或者是一个Container，或者是一个View
Container	例如：一个string，或者一个vector
	C++语言的STL库中存在众多的Container
View	作用在0或多个Range上（通常为1个）
	具有0或1个作用在上述Range上的adaptor
	具有惰性求值的特点

若干 构造基本view的方法/方式

View	描述	构造方法/方式
empty view	不包含任何元素的view	`views::empty`
single view	仅包含一个元素的view	`views::single`
iota view	从一个初始值出发，连续递增的view	`views::iota`

view::empty view::single

```cpp
#include <iostream>
#include <ranges>

using namespace std;

int main()
{
    auto e = views::empty<int>;
    cout << "e.size() = " << e.size() << "\n"; // e.size() = 0

    auto s = views::single('a');
    for(auto x : s) cout << x << " \n"; // a
}
```

一个整数类型的empty view

包含一个元素'a'的view

view::iota

```cpp
#include <iostream>
#include <ranges>

using namespace std;

int main()
{
    for(auto x : views::iota(1, 10)) cout << x << " "; // 1 2 3 4 5 6 7 8 9

    cout << "\n";

    for(auto x : views::iota(1) | views::take(9)) cout << x << " "; // 1 2 3 4 5 6 7 8 9

    cout << "\n";

    for(auto x : views::iota('a') | views::take(26)) cout << x << " "; // a b c ... x y z
}
```

整数区间 [1, 10)

整数区间 [1, +∞)

字符区间 ['a', +∞)

一些 Range adaptor

Adaptor	描述
`reverse`	a view that iterates over the elements of another bidirectional view in reverse order.

type in haskell
`reverse :: [a] -> [a]`

reverse

```cpp
#include <iostream>
#include <ranges>
#include <vector>

using namespace std;

int main()
{
    for(auto x : views::iota(1, 11) | views::reverse)
    {
        cout << x << " "; // 10 9 8 7 6 5 4 3 2 1
    }
}
```

Adaptor	描述
`transform`	a view of a sequence that applies a transformation function to each element.
`filter`	a view that consists of the elements of a range that satisfies a predicate.

type in haskell
`transform :: [a] -> (a -> b) -> [b]`
`filter :: [a] -> (a -> Bool) -> [a]`

transform filter

```
 6  int main()
 7  {
 8      auto is_even = [](auto x){ return x % 2 == 0; };
 9      auto plus_1  = [](auto x){ return x + 1; };
10
11      auto view_1 = views::iota(1, 11) | views::filter(is_even);
12
13      for(auto x : view_1) cout << x << " "; // 2 4 6 8 10
14
15      cout << "\n";
16
17      auto view_2 = view_1 | views::transform(plus_1);
18
19      for(auto x : view_2) cout << x << " "; // 3 5 7 9 11
20  }
```

Adaptor	描述
`take`	a view consisting of the first N elements of another view
`take_while`	a view consisting of the initial elements of another view, until the first element on which a predicate returns false

type in haskell
`take :: [a] -> Int -> [a]`
`take_while :: [a] -> (a -> Bool) -> [a]`

take take_while

```
7   int main()
8   {
9       vector<int> vec = {1,2,3,4,5,4,3,2,1};
10
11      for(auto x : vec | views::take(4)) cout << x << " "; // 1 2 3 4
12
13      cout << "\n";
14
15      auto le_4 = [](auto x){ return x <= 4; };
16
17      for(auto x : vec | views::take_while(le_4)) cout << x << " "; // 1 2 3 4
18  }
```

Adaptor	描述
`drop`	a view consisting of elements of another view, skipping the first N elements.
`drop_while`	a view consisting of the elements of another view, skipping the initial subsequence of elements until the first element where the predicate returns false.

type in haskell
`drop :: [a] -> Int -> [a]`
`drop_while :: [a] -> (a -> Bool) -> [a]`

drop drop_while

```
 7  int main()
 8  {
 9      vector<int> vec = {1,2,3,4,5,4,3,2,1};
10
11      for(auto x : vec | views::drop(4)) cout << x << " "; // 5 4 3 2 1
12
13      cout << "\n";
14
15      auto le_4 = [](auto x){ return x <= 4; };
16
17      for(auto x : vec | views::drop_while(le_4)) cout << x << " "; // 5 4 3 2 1
18  }
```

你知道下面这个程序的功能吗

```
5   using namespace std;
6
7   auto is_space(char c){ return c == ' '; };
8
9   int main()
10  {
11      string str;
12
13      getline(cin, str);
14
15      auto view = str | views::drop_while(is_space)
16                      | views::reverse
17                      | views::drop_while(is_space)
18                      | views::reverse;
19      string rst;
20
21      for (auto c : view) rst.push_back(c);
22
23      cout << rst;
24  }
```

Adaptor	描述
`join`	a view consisting of the sequence obtained from flattening a view of ranges.

type in haskell
`join :: [[a]] -> [a]`

join

```cpp
using namespace std;

template <typename T, typename R>
vector<T> to_vector(R range)
{
    vector<T> rst;

    for(auto x : range) rst.push_back(x);

    return rst;
}

int main()
{
    auto vec1 = to_vector<int>(views::iota(1) | views::take(5));
    auto vec2 = to_vector<int>(views::iota(6) | views::take(5));
    auto vec3 = to_vector<int>(views::iota(11)| views::take(5));

    vector<vector<int>> vvi{vec1, vec2, vec3};

    auto join_view = vvi | views::join;

    for(auto c : join_view) cout << c << " ";
    // 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
}
```

join

```cpp
#include <iostream>
#include <ranges>
#include <vector>
#include <string>

using namespace std;

int main()
{
    vector<string> ss{"hello", " ", "world", "!"};

    for (auto ch : ss | views::join) cout << ch; // hello world!
}
```

Adaptor	描述
`split`	a view over the subranges obtained from splitting another view using a delimiter.

type in haskell
`split :: [a] -> a -> [[a]]`

split

```
6   using namespace std;
7
8   int main()
9   {
10      string sentence{"the quick brown fox"};
11
12      for (auto word : sentence | views::split(' ')) {
13          for (char ch : word) cout << ch;
14          cout << '\n';
15      }
16      // the
17      // quick
18      // brown
19      // fox
20  }
```

Adaptor	描述
`keys`	takes a view consisting of pair-like values and produces a view of the first elements of each pair.
`values`	takes a view consisting of pair-like values and produces a view of the second elements of each pair.

type in haskell
`keys :: [(a,b)] -> [a]`
`values :: [(a,b)] -> [a]`

keys values

```cpp
int main()
{
    auto history_info = vector<pair<string, int>>{
        {"张衡", 78},
        {"王唯", 701},
        {"李白", 701},
        {"杜甫", 712},
        {"曹雪芹", 1715}
    };

    auto names = history_info | views::keys;
    auto years = history_info | views::values;

    for(auto & name : names) cout << name << " "; // 张衡 王唯 李白 杜甫 曹雪芹
    cout << "\n";

    for(auto & year : years) cout << year << " "; // 78 701 701 712 1715
    cout << "\n";
}
```

Adaptor	描述
`elements`	takes a view consisting of tuple-like values and a number N and produces a view of N'th element of each tuple.

type in haskell
`elements :: [(a_1,a_2,…,a_N)] -> Int -> [a_i]`

elements

```cpp
int main()
{
    auto history_info = vector<tuple<string, int, int>>{
        {"张衡",   78,   139},
        {"王唯",   701,  761},
        {"李白",   701,  762},
        {"杜甫",   712,  770},
        {"曹雪芹", 1715, 1763}
    };

    auto names   = history_info | views::elements<0>;
    auto b_years = history_info | views::elements<1>;
    auto e_years = history_info | views::elements<2>;

    for(auto & name : names) cout << name << " "; // 张衡 王唯 李白 杜甫 曹雪芹
    cout << "\n";

    for(auto & begin : b_years) cout << begin << " "; // 78 701 701 712 1715
    cout << "\n";

    for(auto & end : e_years) cout << end << " "; // 139 761 762 770 1763
    cout << "\n";
}
```

以上是C++2020版本的range库中包含的大部分range adoptors

我能自定义一个range adopter吗?

能，但是学习成本非常高！

一个自定义adatopr的实现代码：custom_take

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <ranges>
#include <concepts>
#include <algorithm>
#include <assert.h>

namespace rg = std::ranges;

template<rg::input_range R> requires rg::view<R>
class custom_take_view : public rg::view_interface<custom_take_view<R>>{
private:
    R                                       base_ {};
    std::iter_difference_t<rg::iterator_t<R>> count_ {};
    rg::iterator_t<R>                       iter_ {std::begin(base_)};
public:
    custom_take_view() = default;

    constexpr custom_take_view(R base, std::iter_difference_t<rg::iterator_t<R>> count)
        : base_(base), count_(count), iter_(std::begin(base_)){}

    constexpr R base() const & {return base_;}
    constexpr R base() && {return std::move(base_);}

    constexpr auto begin() const {return iter_;}
    constexpr auto end() const { return std::next(iter_, count_); }

    constexpr auto size() const requires rg::sized_range<const R>{
        const auto s = rg::size(base_);
        const auto c = static_cast<decltype(s)>(count_);
        return s < c ? 0 : s - c;
    }
};
```

```cpp
template<class R>
custom_take_view(R&& base, std::iter_difference_t<rg::iterator_t<R>>)
    -> custom_take_view<rg::views::all_t<R>>;

namespace details{
    struct custom_take_range_adaptor_closure{
        std::size_t count_;
        constexpr custom_take_range_adaptor_closure(std::size_t count): count_(count){}
        template <rg::viewable_range R>
        constexpr auto operator()(R && r) const{
            return custom_take_view(std::forward<R>(r), count_);
        }
    } ;

    struct custom_take_range_adaptor{
        template<rg::viewable_range R>
        constexpr auto operator () (R && r, std::iter_difference_t<rg::iterator_t<R>> count){
            return custom_take_view( std::forward<R>(r), count ) ;
        }

        constexpr auto operator () (std::size_t count){
            return custom_take_range_adaptor_closure(count);
        }
    };

    template <rg::viewable_range R>
    constexpr auto operator | (R&& r, custom_take_range_adaptor_closure const & a){
        return a(std::forward<R>(r)) ;
    }
}

namespace views{
    details::custom_take_range_adaptor custom_take;
}
```

https://mariusbancila.ro/blog/2020/06/06/a-custom-cpp20-range-view/

在你对C++的相关细节没有足够的了解之前
暂时放弃自定义range adoptor的想法吧

但还有一个好消息

C++2020 range库 来源于 range-v3

(https://ericniebler.github.io/range-v3/)

Supported Compilers

The code is known to work on the following compilers:

- clang 5.0
- GCC 6.5
- Clang/LLVM 6 (or later) on Windows
- MSVC VS2019, with `/permissive-` and either `/std:c++latest` or `/std:c++17`

range-v3 提供了一组更有趣的range adaptor

(需要把range-v3的源代码包含在本地C++编译器的include path中)

view::replace

把序列中特定的元素替换为另一个特定的元素

```
1  #include <range/v3/all.hpp>
2  #include <iostream>
3  #include <string>
4
5  using namespace ranges;
6
7  int main()
8  {
9    std::string s{"hello, world, who am I?"};
10
11   std::string r = s | view::replace(',', ' ');
12
13   std::cout << r;
14 }
```

把序列中的逗号替换为空格

```
hello  world  who am I
```

view::replace_if

把序列中满足特定条件的元素替换为另一个特定的元素

```cpp
#include <range/v3/all.hpp>
#include <iostream>
#include <string>

using namespace ranges;

auto idn = [](char c)
{
  return (c < 'a' || c > 'z')&&(c < 'A' || c > 'Z');
};

int main()
{
  std::string s{"hello, world, who am I?"};

  std::string r = s | view::replace_if(idn, '*');

  std::cout << r;
}
```

把序列中所有非字母的字符替换为星号

```
he|lo**wor|d**who*am*|*
```

view::intersperse

在序列中的相邻元素间插入一个特定的元素

```
1  #include <range/v3/all.hpp>
2  #include <iostream>
3  #include <vector>
4
5  using namespace ranges;
6
7  int main()
8  {
9    auto v = view::ints(1) | view::take(10)
10                           | view::intersperse(0);
11
12   for(auto d : v) std::cout << d << ' ';
13 }
```

```
1 0 2 0 3 0 4 0 5 0 6 0 7 0 8 0 9 0 10
```

view::unique

去除序列中相邻的重复元素

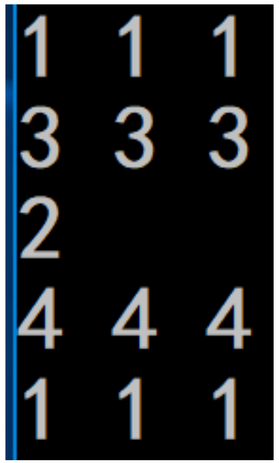
```cpp
#include <range/v3/all.hpp>
#include <iostream>
#include <vector>

using namespace ranges;

auto xiaoyu6 = [](auto i){ return i < 6; };

int main()
{
  std::vector<int> v {1,1,1,3,3,3,2,4,4,4,1,1,1};

  std::vector<int> r = v | view::unique;

  for(auto d : r) std::cout << d << ' ';
}
```

```
1 3 2 4 1
```

view::group_by

把序列中的相邻元素按照某个条件分成很多子序列

```cpp
#include <range/v3/all.hpp>
#include <iostream>
#include <vector>

using namespace ranges;

auto dengy = [](const auto x, const auto y){ return x == y; };

int main()
{
  std::vector<int> v {1,1,1,3,3,3,2,4,4,4,1,1,1};

  auto r = v | view::group_by(dengy);

  for(auto d : r)
  {
    for(auto e : d) std::cout << e << ' ';
    std::cout << std::endl;
  }
}
```



view::group_by

把序列中的相邻元素按照某个条件分成很多子序列

```cpp
#include <range/v3/all.hpp>
#include <iostream>
#include <vector>

using namespace ranges;

auto idn = [](const auto x, const auto y)
{
  return (x >= 0 && y >= 0)||(x < 0 && y < 0);
};

int main()
{
  std::vector<int> v {1,2,3,-3,-4,-5,2,4,-4,-5,7,-8,9};

  auto r = v | view::group_by(idn);

  for(auto d : r)
  {
    for(auto e : d) std::cout << e << ' ';
    std::cout << std::endl;
  }
}
```

```
1 2 3
-3 -4 -5
2 4
-4 -5
7
-8
9
```

view::adjacent_filter 邻接元素过滤器

- 从前向后依次访问一个序列中的每一对邻接元素 (a,b)
  - 判断(a,b)是否满足特定的条件（通过一个lambda函数进行判断）
    - 如果满足该条件，则继续访问下一对邻接元素 (b,b后面的元素)
    - 否则，删除b元素， 然后访问下一对邻接元素 (a,b后面的元素)

```cpp
#include <range/v3/all.hpp>
#include <iostream>
#include <string>

using namespace ranges;

auto idn = [](char a, char b){ return a != b; };

int main()
{
  std::string s{"hello,   world."};

  std::string r = s | view::adjacent_filter(idn);

  std::cout << r;
}
```

这里是三个连续的空格

```
helo,  world.
```

一个小功能：删除字符串中的冗余空格

1. 字符串前后存在的空格
2. 两个单词之间存在的冗余空格

```cpp
#include <range/v3/all.hpp>
#include <iostream>
#include <string>

using namespace ranges;

auto is_kg = [](char c){ return c == ' '; };
auto is_flxkg = [](char a, char b){ return (a != b)||(b != ' '); };

int main()
{
  std::string s{"   hello,   world.   "};

  std::string r = s | view::drop_while(is_kg)
                    | view::reverse
                    | view::drop_while(is_kg)
                    | view::reverse
                    | view::adjacent_filter(is_flxkg);

  std::cout << r;
}
```

```
hello, world.
```

view::adjacent_remove_if 邻接元素过滤器

view::adjacent_filter 的反操作

- 从前向后依次访问一个序列中的每一对邻接元素 (a,b)
  - 判断(a,b)是否满足特定的条件（通过一个lambda函数进行判断）
    - 如果不满足该条件，则继续访问下一对邻接元素 (b,b后面的元素)
    - 否则，删除a元素， 然后访问下一对邻接元素 (b,b后面的元素)

```
1  #include <range/v3/all.hpp>
2  #include <iostream>
3  #include <string>
4
5  using namespace ranges;
6
7  auto idn = [](char a, char b){ return (a == b)&&(a == ' '); };
8
9  int main()
10 {
11   std::string s{"hello,   world."};
12
13   std::string r = s | view::adjacent_remove_if(idn);
14
15   std::cout << r;
16 }
```

这里是三个连续的空格

```
hello, world.
```

view::join

给定一个序列，其中的每个元素又是一个序列

将这个序列中的所有序列串在一起，形成一个更长的序列

输入	3行字符串（可能含空格）	输出	这3行字符串中空格字符的总数

```
1  #include <range/v3/all.hpp>
2  #include <iostream>
3  #include <string>
4  #include <vector>
5
6  using namespace ranges;
7
8  int main()
9  {
10   std::vector<std::string> vs(3);
11   for(auto &d : vs) std::getline(std::cin, d);
12
13   std::cout << count(view::join(vs), ' ');
14 }
```

view::concat

给定多个序列，把这些序列串在一起，形成一个更长的序列

输入	3行字符串（可能含空格）	输出	这3行字符串中空格字符的总数

```
1  #include <range/v3/all.hpp>
2  #include <iostream>
3  #include <string>
4  #include <vector>
5
6  using namespace ranges;
7
8  int main()
9  {
10   std::vector<std::string> vs(3);
11   for(auto &d : vs) std::getline(std::cin, d);
12
13   std::cout << count(view::concat(vs.at(0),vs.at(1),vs.at(2)), ' ');
14 }
```

view::zip

把两个序列，像拉链一样，缝合成一个序列

```cpp
#include <range/v3/all.hpp>
#include <iostream>
#include <string>
#include <vector>

using namespace ranges;

int main()
{
  auto view1 = view::ints(1);

  auto view2 = view::iota('a') | view::take(26);

  auto zip_view = view::zip(view1, view2);

  for(const auto &pair : zip_view)
  {
    std::cout << pair.first << ' ' << pair.second << std::endl;
  }
}
```

```
1 a
2 b
3 c
4 d
5 e
6 f
7 g
8 h
9 i
10 j
11 k
12 l
13 m
14 n
15 o
16 p
17 q
18 r
19 s
20 t
21 u
22 v
23 w
24 x
25 y
26 z
```

view::zip_with

把两个序列，像拉链一样，缝合成一个序列

```cpp
#include <range/v3/all.hpp>
#include <iostream>
#include <string>
#include <vector>

using namespace ranges;

int main()
{
  auto view1 = view::ints(1);

  auto view2 = view::iota('a') | view::take(26);

  auto zipper = [](int i, char c) { return std::to_string(i) + c; };

  auto zip_view = view::zip_with(zipper, view1, view2);

  for(const auto &d : zip_view) std::cout << d << std::endl;
}
```

to_string

名字空间std中的一个函数

把一个整型数或浮点数转成

对应的十进制形式的字符串

```
1a
2b
3c
4d
5e
6f
7g
8h
9i
10j
11k
12l
13m
14n
15o
16p
17q
18r
19s
20t
21u
22v
23w
24x
25y
26z
```

以上是range-v3库中
包含的一些更复杂的range adoptors

1. FP in C++ 2020

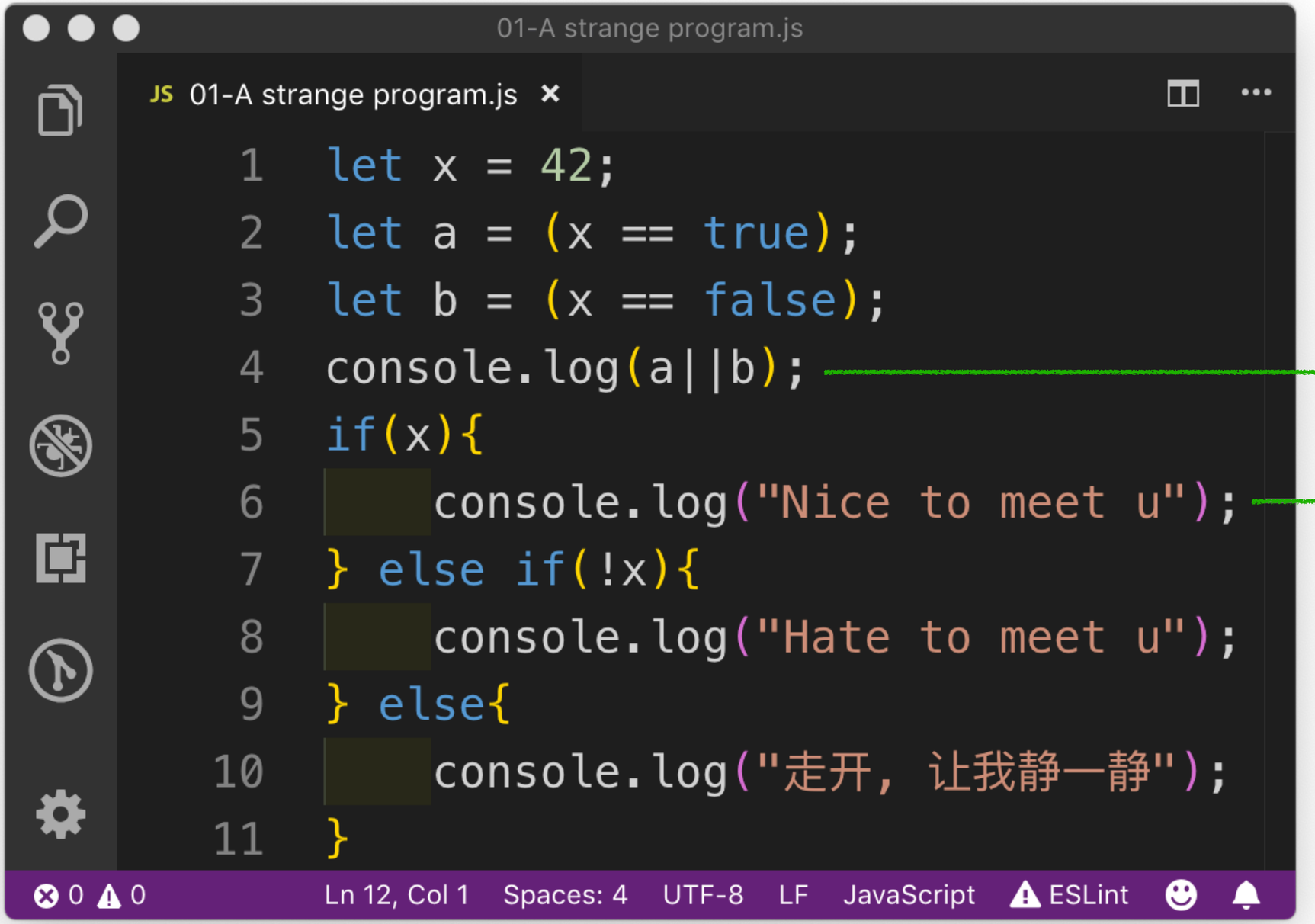
暂时就到这里

若干观点和想法	
1	中国互联网大厂中的绝大多数程序员，对FP的理解，比你们中的大多数人，要差很远
2	你是否可以开发一个程序库/框架，让这些程序员使用C++语言进行FP的学习成本更低
3	给定一段Haskell程序，能否把它自动转化为对应的C++程序

2. FP in JavaScript

如果你苦C++久矣
JS可能就是你的菜

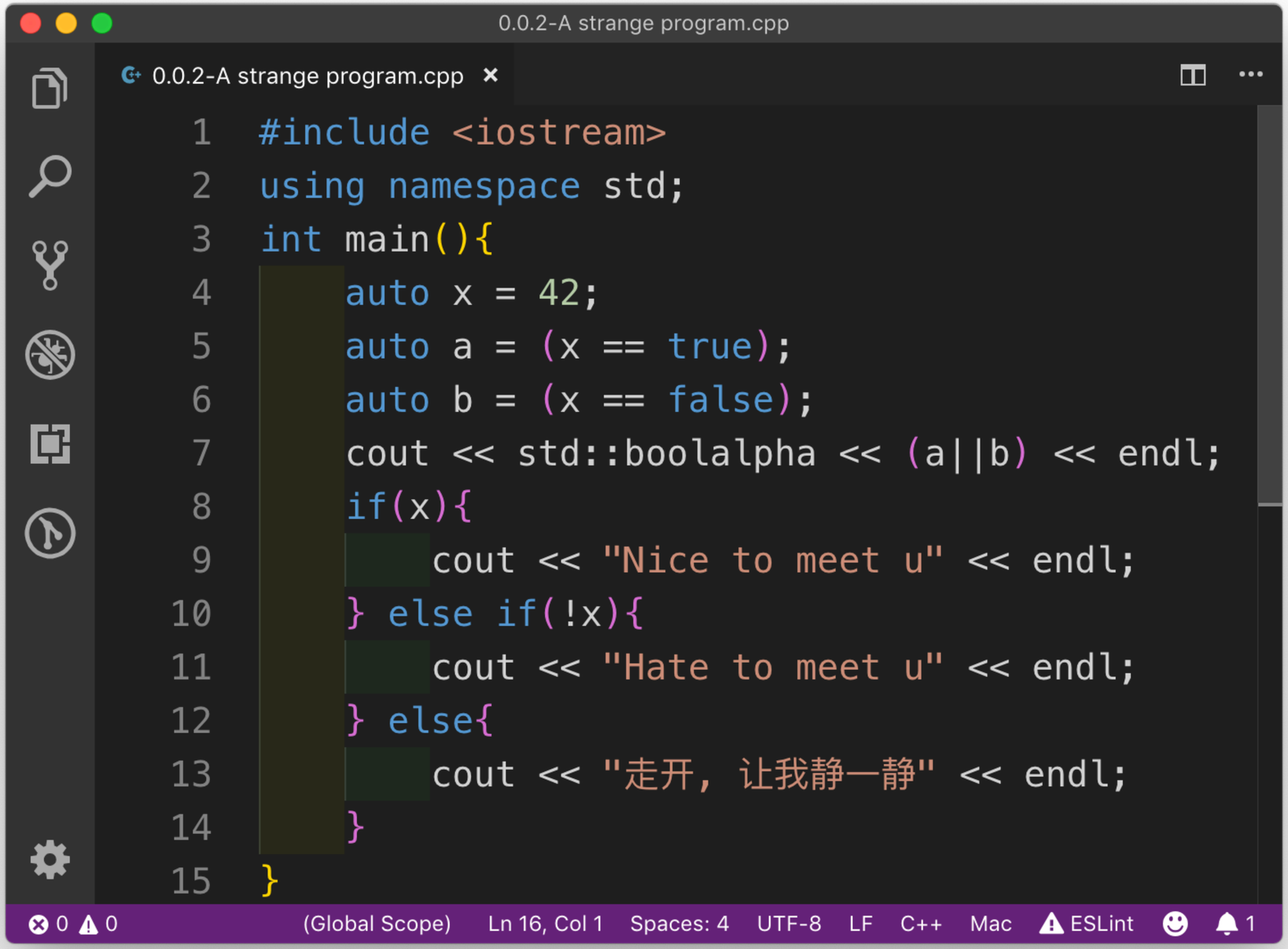
JS：一种奇怪的语言



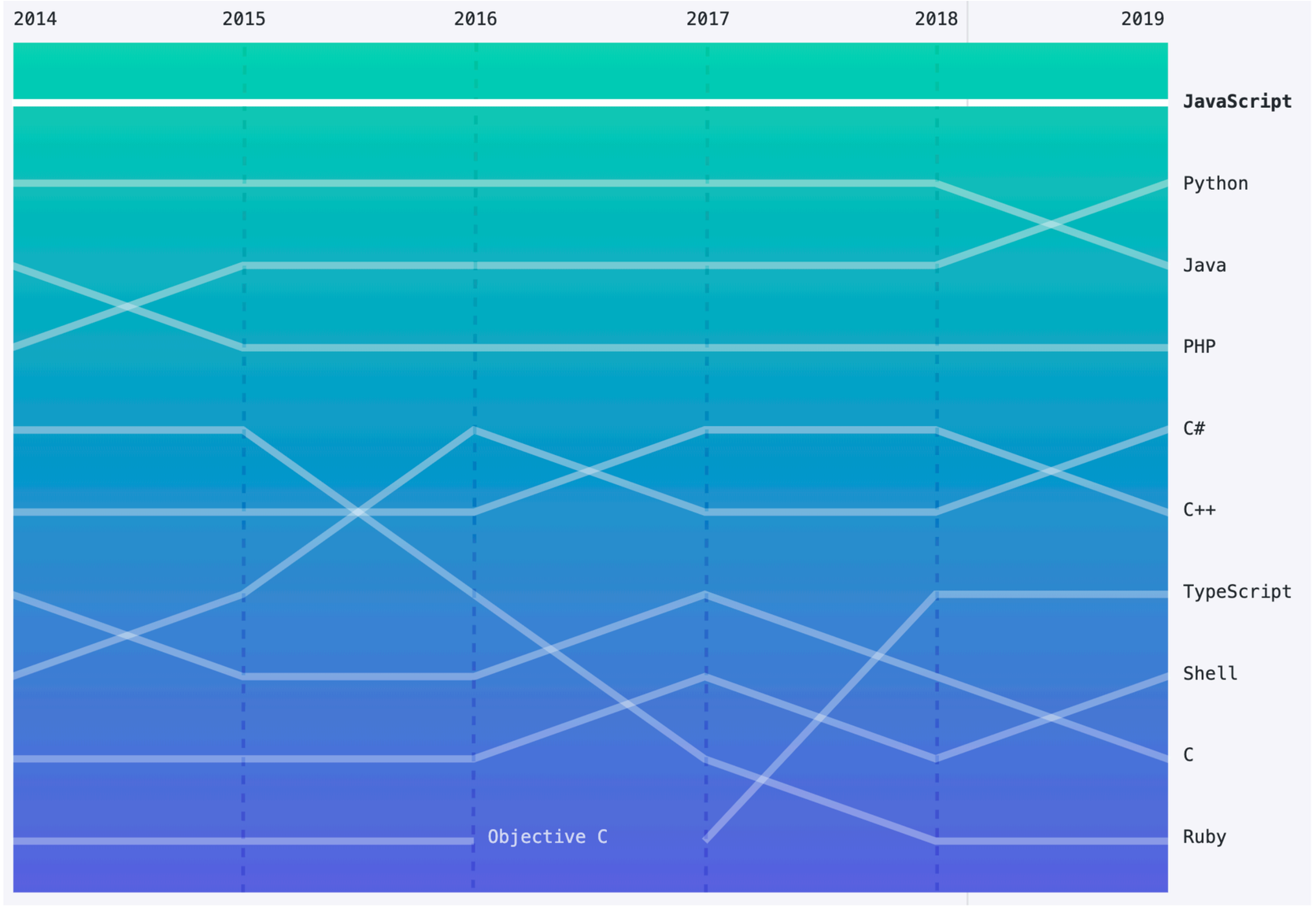
false

Nice to meet u

其实这一点都不奇怪，因为C++也是如此



GitHub 2019 年度报告：编程语言 TOP10



JavaScript 语言的应用领域

浏览器端应用
Browser-side app.

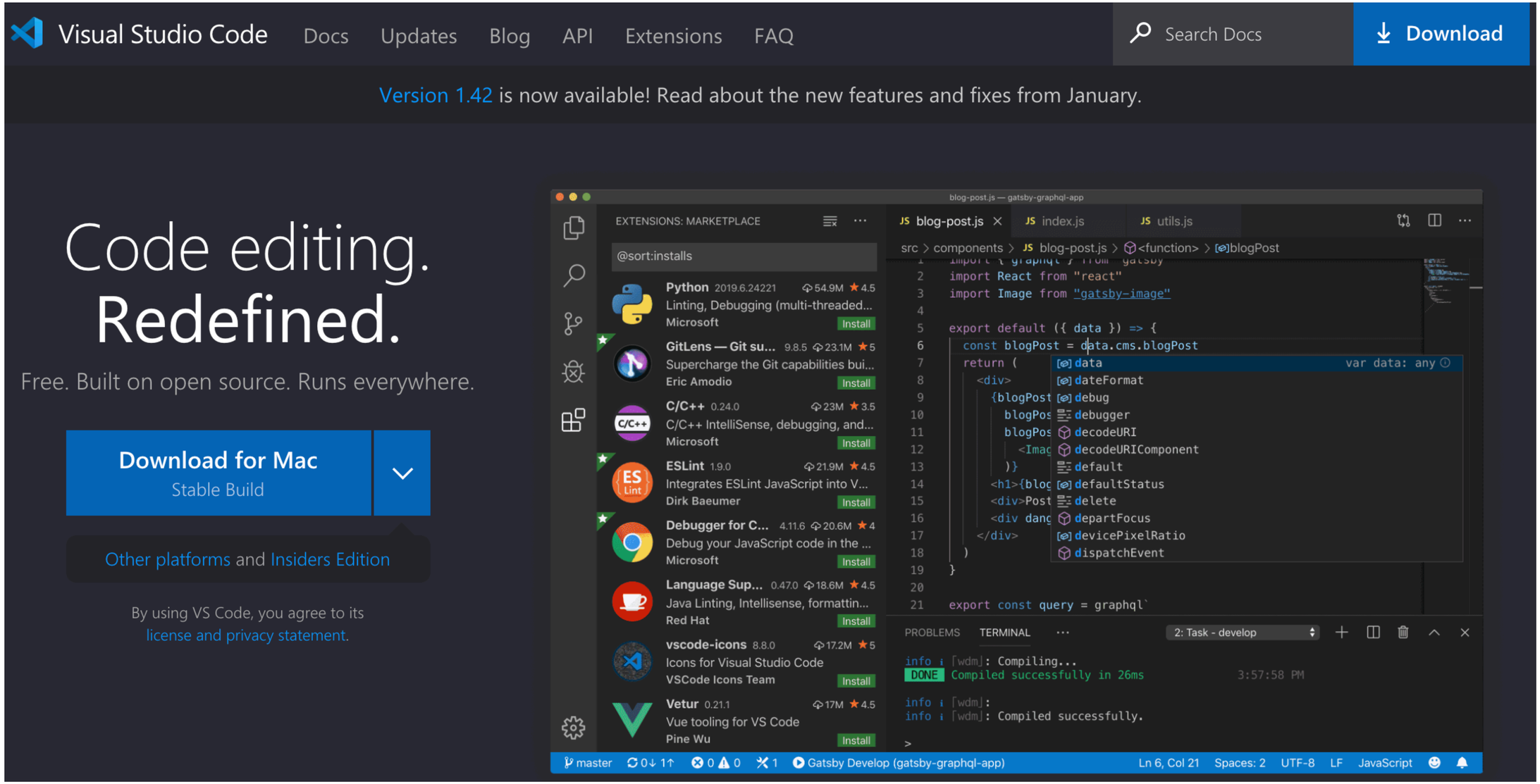
服务器端应用
Server-side app.

本地应用
Native app.

嵌入式软件
Embedded app.

应用举例

Visual Studio Code
—个用JS编写的程序集成开发环境



第一个JS程序

JS宿主环境中存在的一个object
控制台：可以输入/输出文本信息

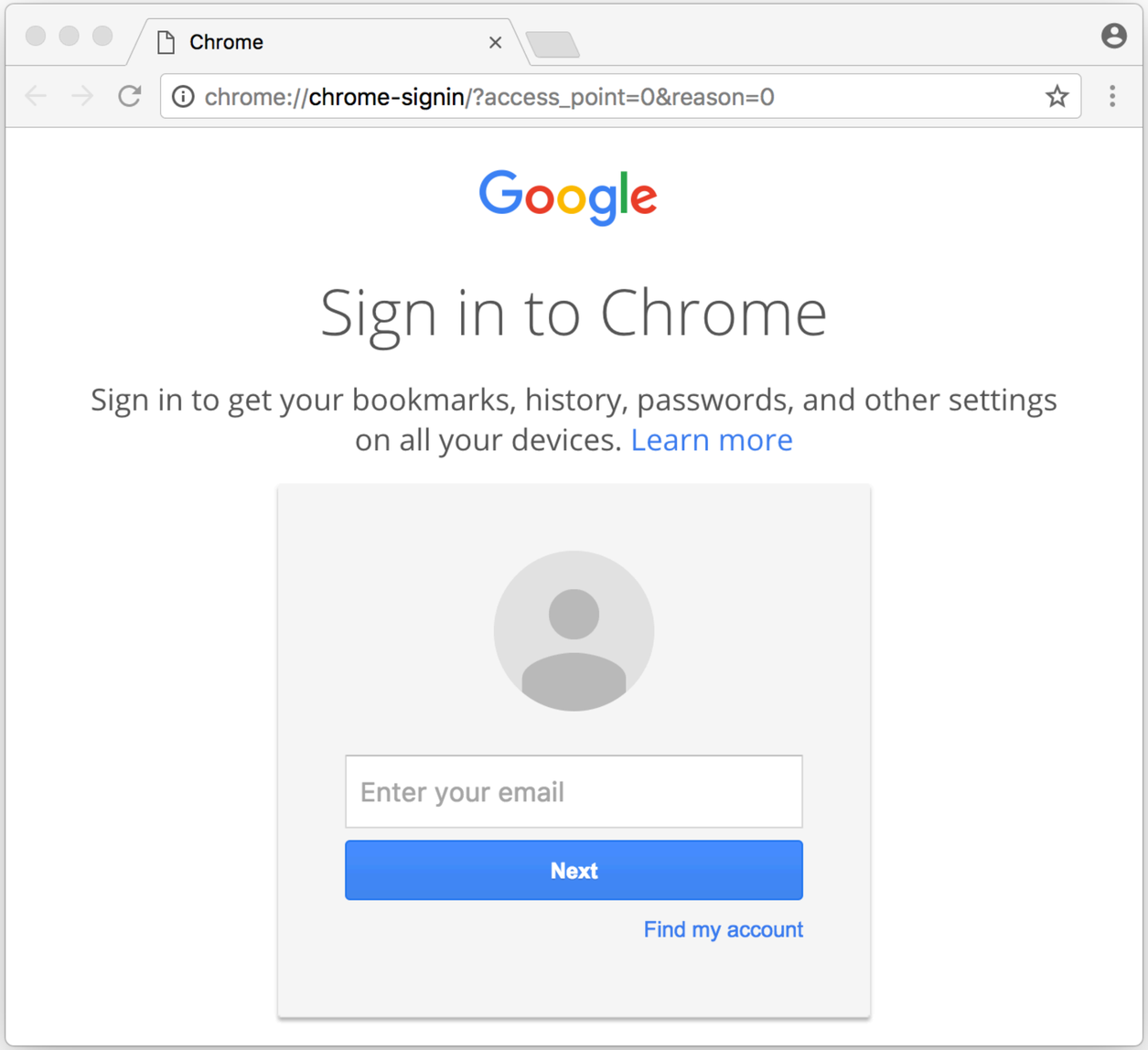
```
console.log("Hello World!");
```

控制台上附着的一个function
向控制台输出一行文本信息

怎么运行这个JS程序呢

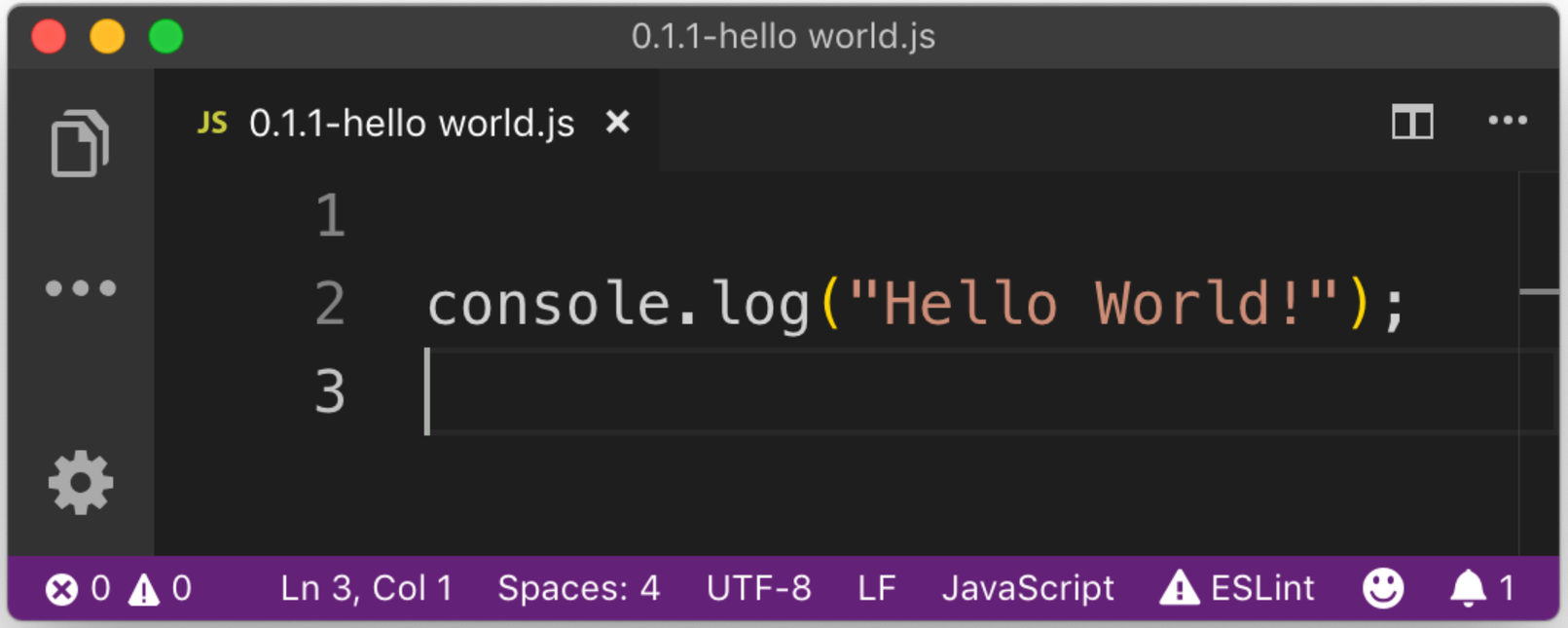
这个事情说来话长

第一种方式：在浏览器中运行



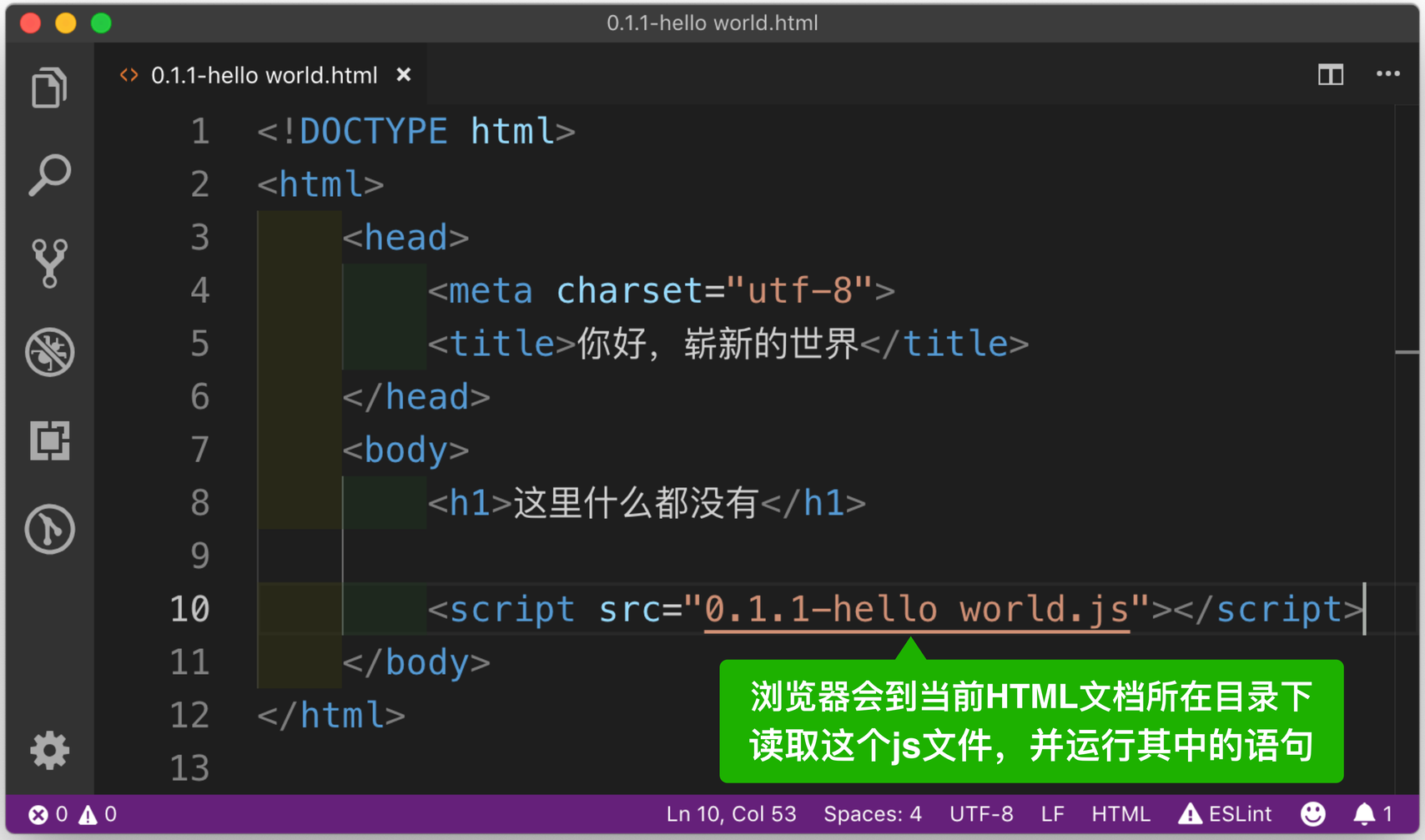
第一种方式：在浏览器中运行

第一步：把JS程序存储为一个扩展名为js的文本文件



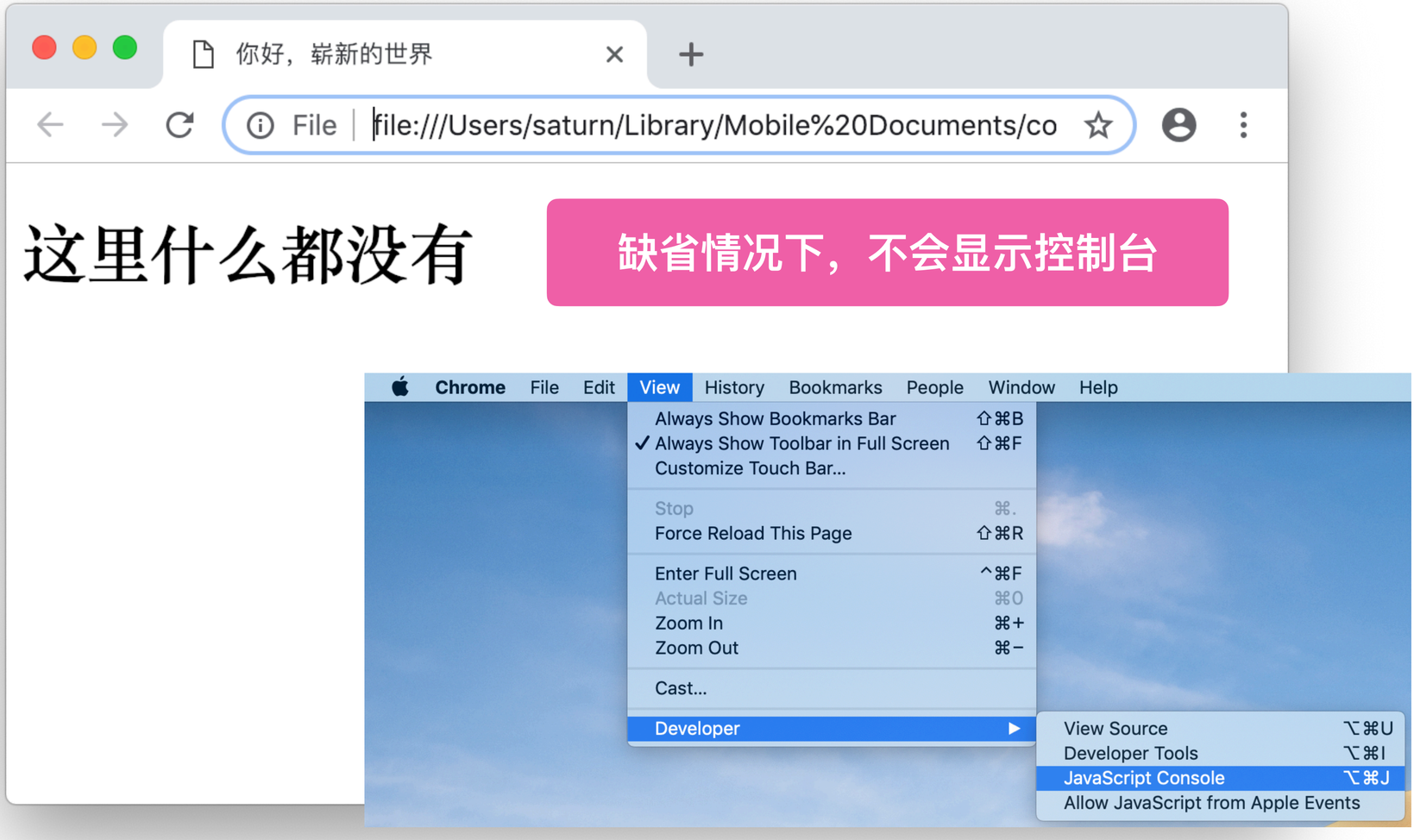
第一种方式：在浏览器中运行

第二步：把js文件关联到一个HTML文件上



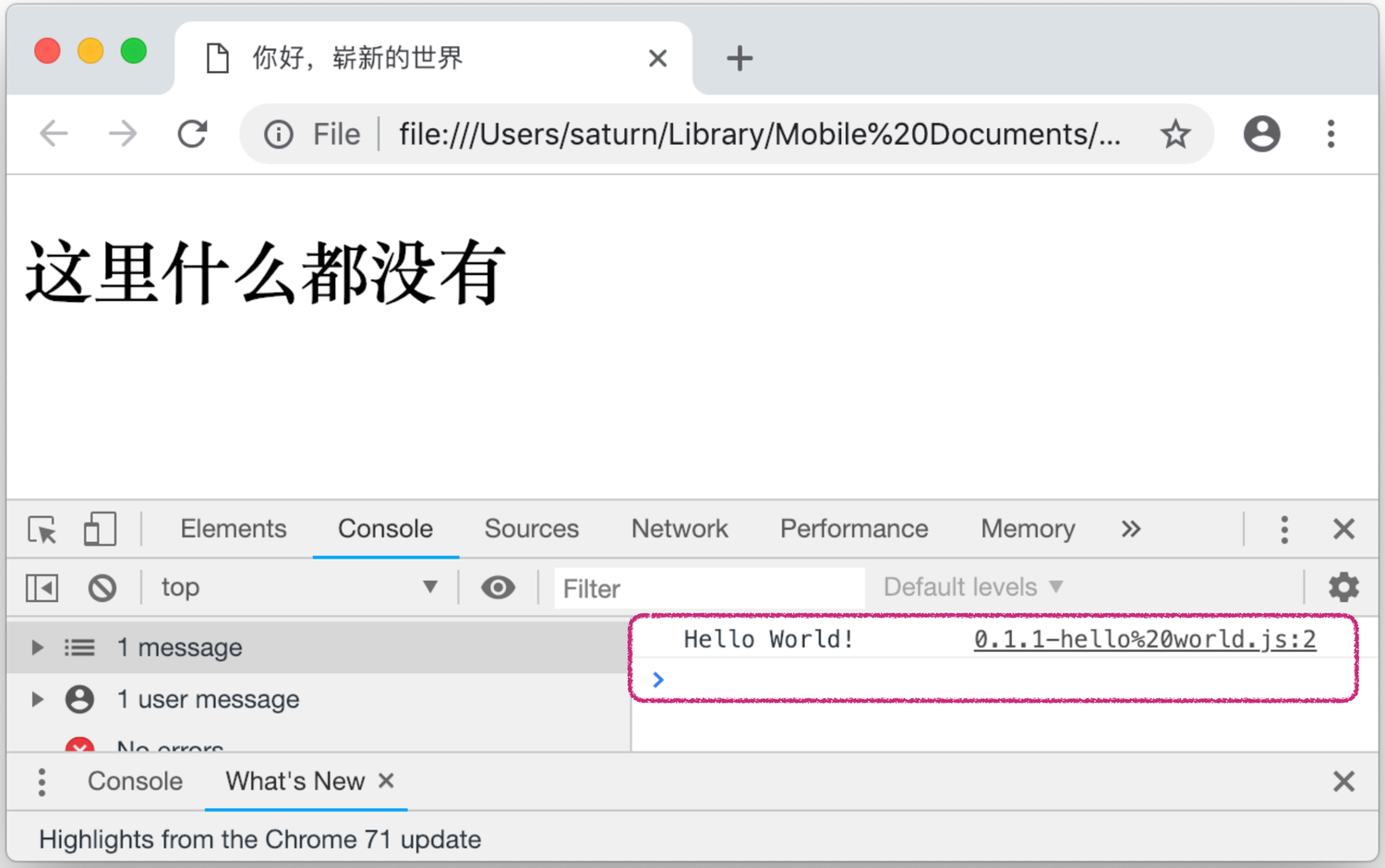
第一种方式：在浏览器中运行

第三步：在浏览器中打开这个HTML文件

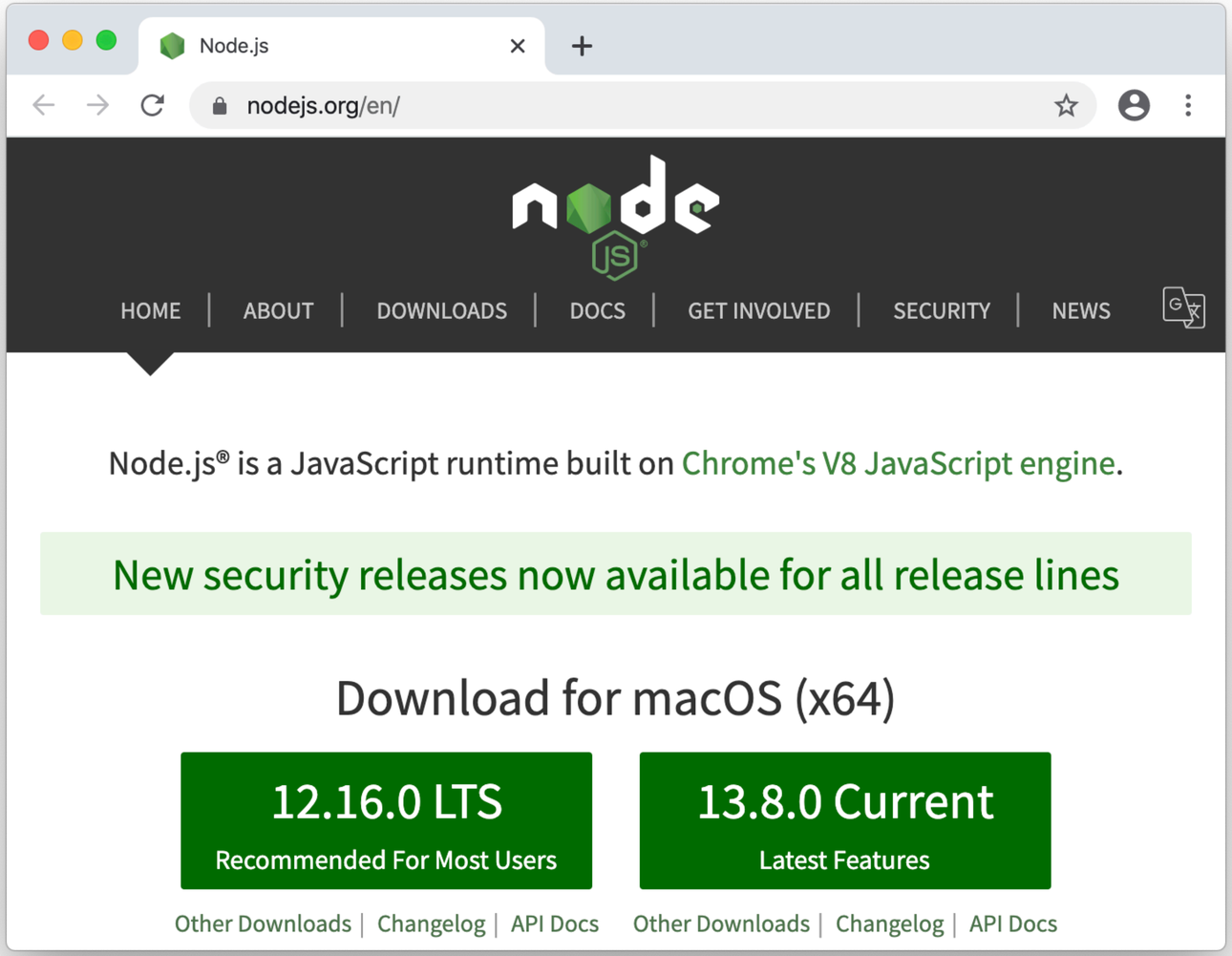


第一种方式：在浏览器中运行

第三步：在浏览器中打开这个HTML文件

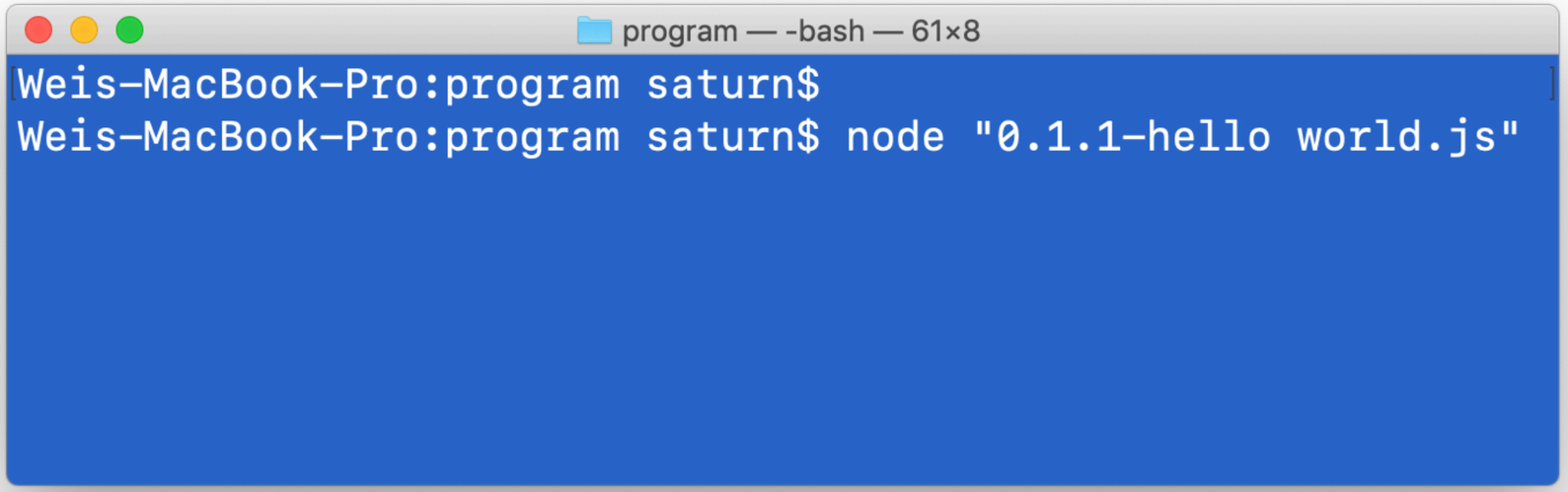


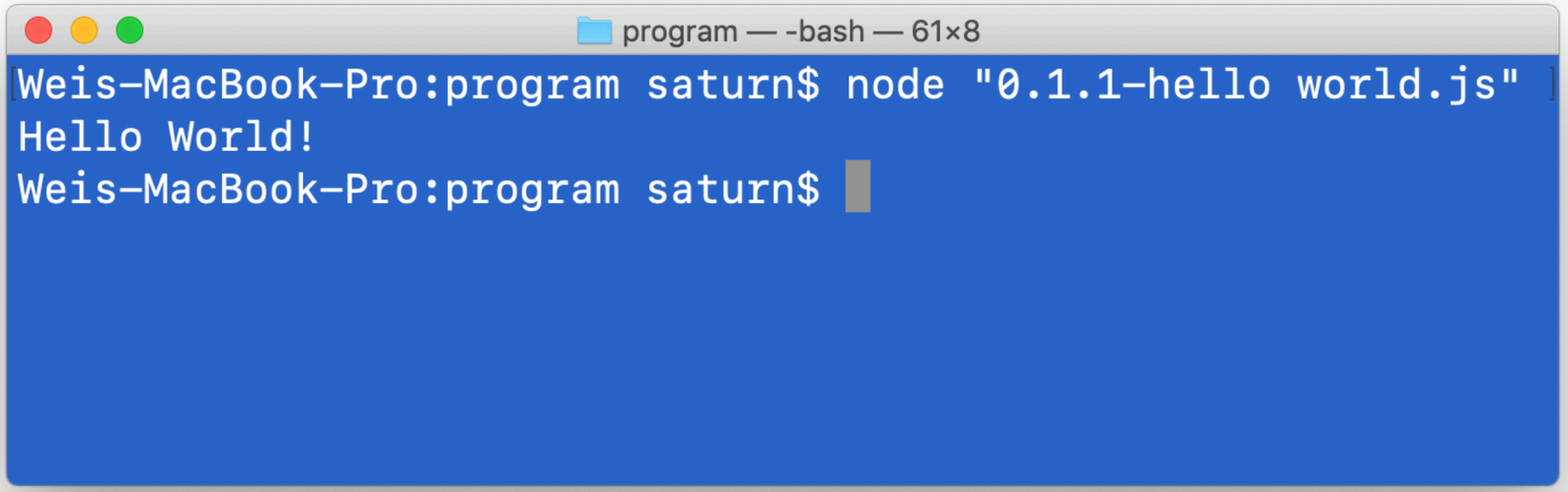
第二种方式：在 命令行/服务器环境中 运行



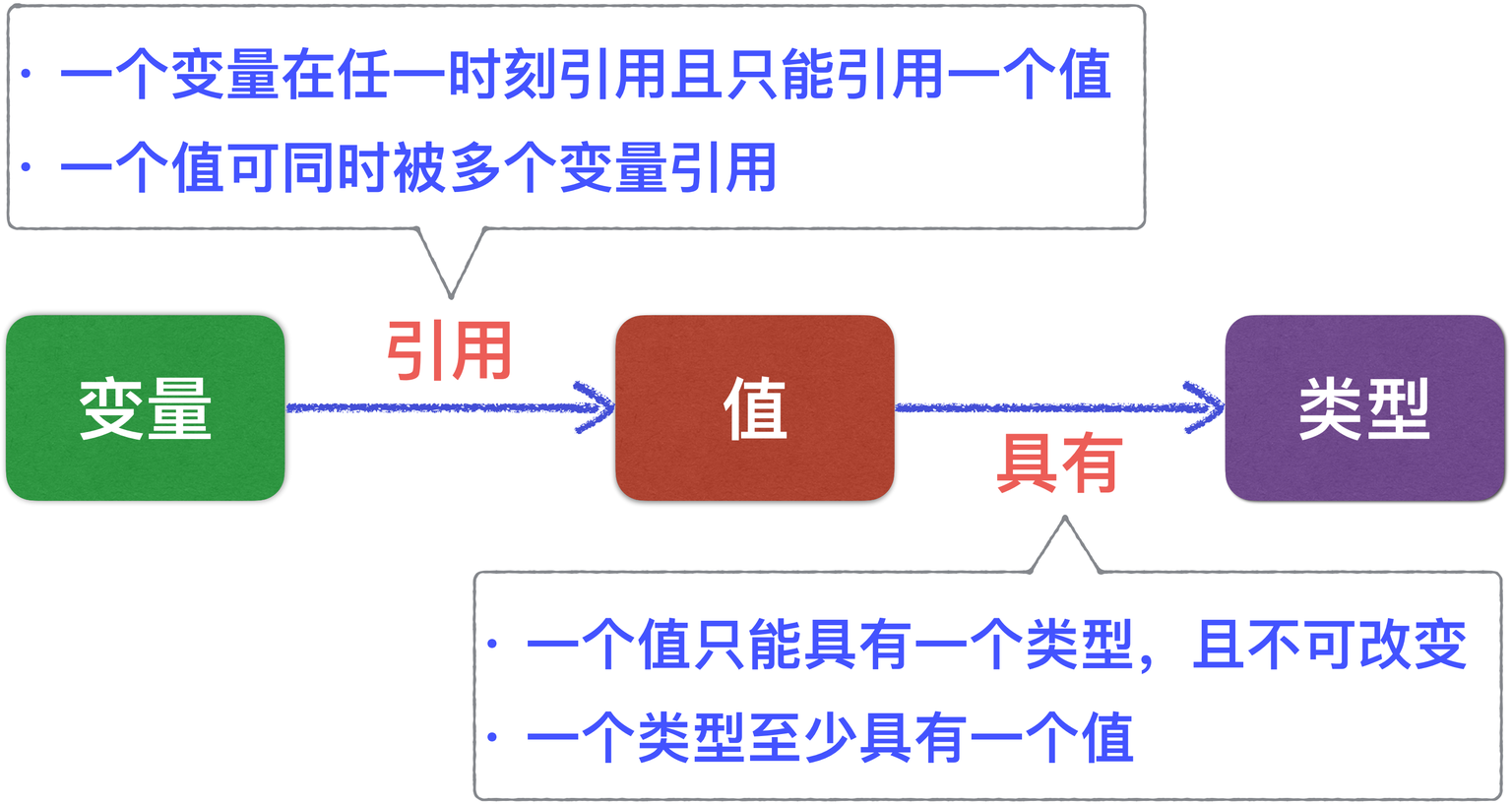
Node.js： JS服务器端的运行环境

第二种方式：在 命令行/服务器环境中 运行





三个基本概念



在JS中，变量没有类型，但值却是有类型的

如何声明一个变量

变量声明关键词 + 变量名

- let
- const

命名规则（可能不是完全准确）

1. 首字符:字母、下划线、美元符号
2. 其它:字母、下划线、美元符、数字
3. 不能与关键词和保留词重名

```
let $hello;
let 变量;
const _world = 5;
```

1. 使用const声明的变量，在声明时必须赋一个值
2. 然后，变量与值之间的引用关系就无法改变了

JS中的值具有七种类型（来源：ES10）

ID	Name	名称	分类	重要性质
1	null	空	primitive type （原子类型）	所有原子类型的值 在声明后 都不会再发生变化 *All primitive type values are immutable*
2	undefined	未定义		
3	boolean	布尔		
4	number	数字		
5	string	字符串		
6	symbol	符号		
7	object	对象	non-primitive type	

佛系排序法

这是我学习编程20多年来，看到的最搞笑的代码

为了能get到这个笑点
你需要知道JS中的一个函数

一个函数

一个非负整数：毫秒数

```
const id = setTimeout(func, delay);
```

行为：延迟delay毫秒后触发函数func

这种理解，严格而言，是错误的

为了能get到这个笑点
你需要知道JS中如何声明的lambda function

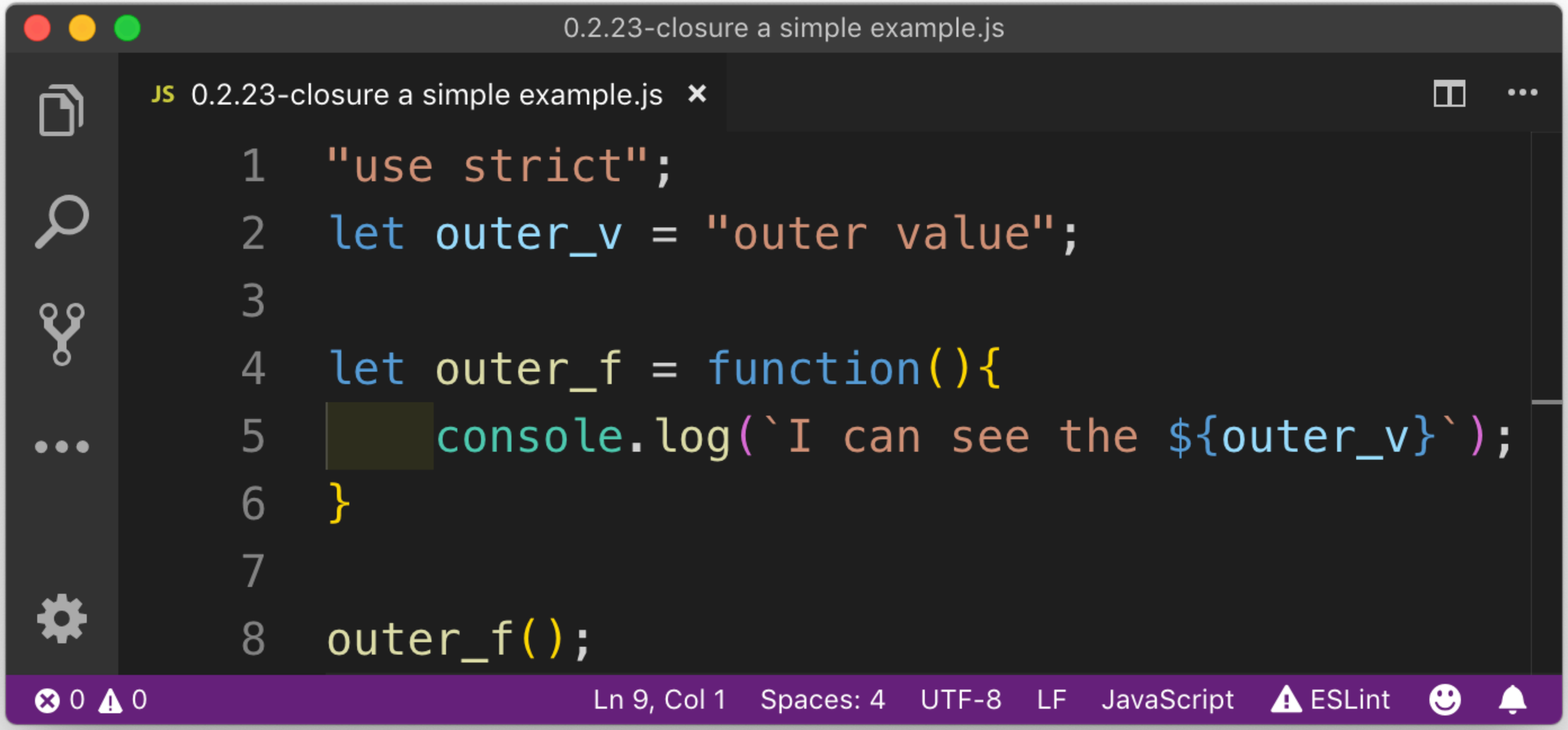
```
1    const add1 = (x, y) => { return x + y };
2
3    console.log(add1(1, 2));
4
5    const add2 = (x, y) => x + y;
6
7    console.log(add2(1, 2));
```

佛系排序法

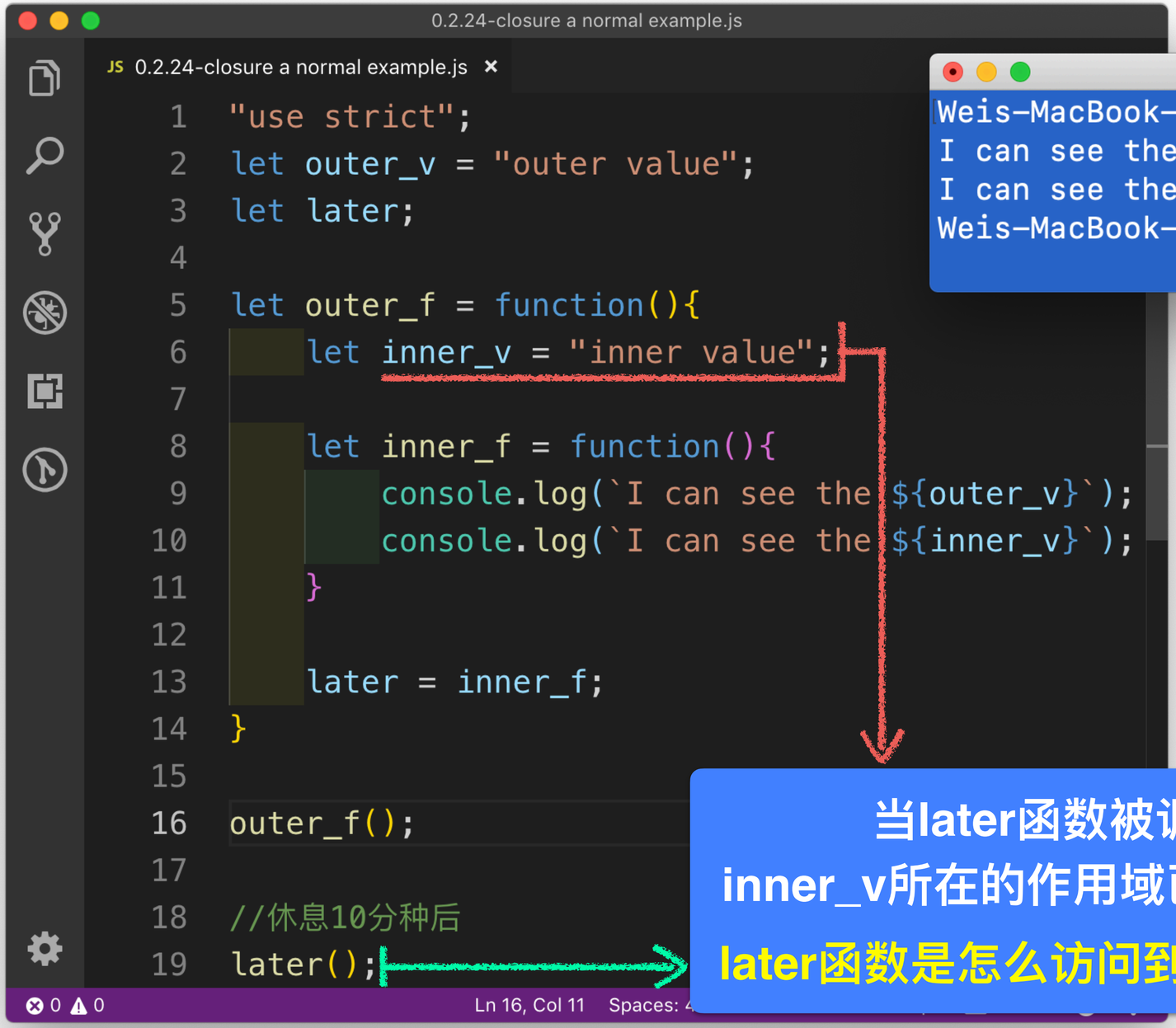
```
1   function lazySort(list, callback){
2       let rst = [];
3
4       list.forEach(i => {
5           setTimeout( () => {
6               rst.push(i);
7               if(rst.length == list.length) callback(rst);
8           }, i);
9       });
10  }
11
12  lazySort([4,5,6,7,1,2,4,5], console.log);
```

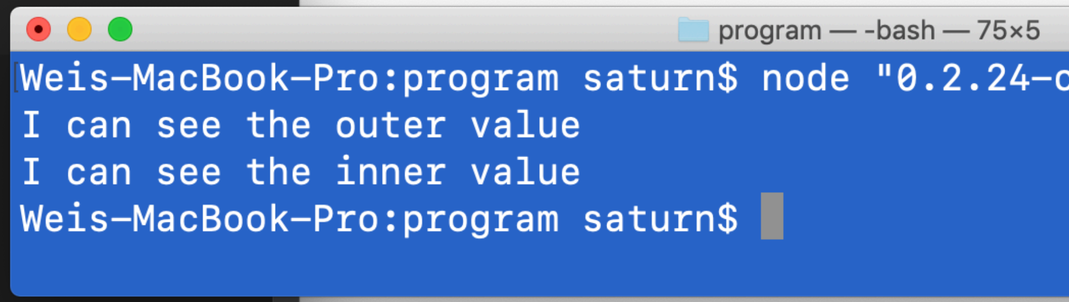
JS中的闭包（closure）

闭包：一个简单的例子


0.2.23-closure a simple example.js
JS 0.2.23-closure a simple example.js
1 "use strict";
2 let outer_v = "outer value";
3
4 let outer_f = function(){
5 console.log(`I can see the ${outer_v}`);
6 }
7
8 outer_f();
0 0
Ln 9, Col 1
Spaces: 4
UTF-8
LF
JavaScript
ESLint

闭包：一个更加明显的例子



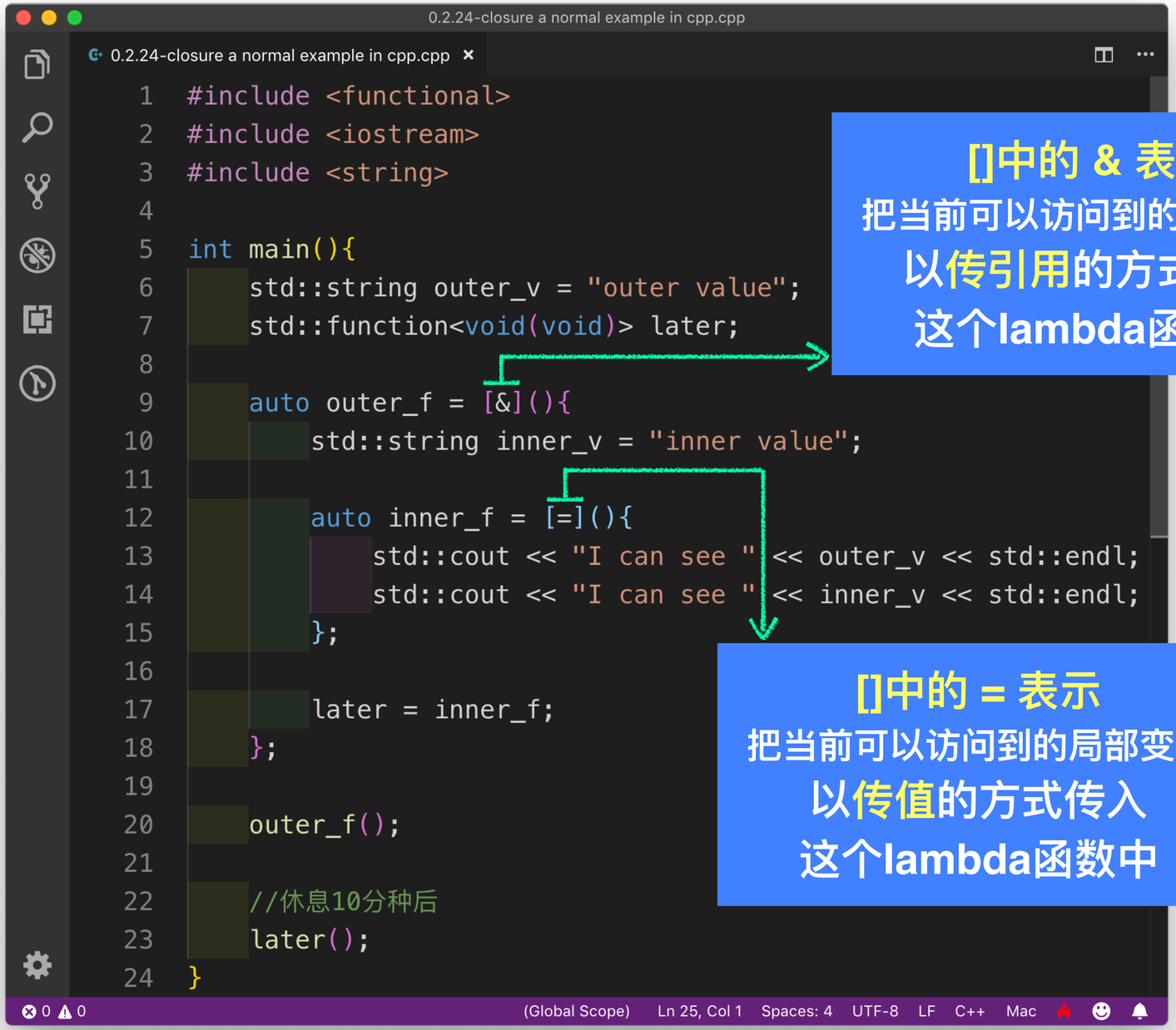


闭包

当later函数被调用时
inner_v所在的作用域已经不知所踪

later函数是怎么访问到inner_v的呢

上面的例子在C++中的近似等价程序



[]中的 & 表示
把当前可以访问到的局部变量
以传引用的方式传入
这个lambda函数中

[]中的 = 表示
把当前可以访问到的局部变量
以传值的方式传入
这个lambda函数中

JS中的原型链

你需要先知道：如何创建一个object

```
const zhs = {
    name: "张三",
    age : 18
}

zhs.height = "175cm";

console.log(zhs.name, zhs.age, zhs.height); // 张三 18 175cm
```

```
const dog = {
    type : "汪星人",
    say_hello(){ console.log("汪汪 " + this.type) }
}

const cat = {
    type : "喵星人",
    say_hello(){ console.log("喵喵, " + this.type) }
}

const whoAmI = { type : "unknown" }

Object.setPrototypeOf(whoAmI, dog);
whoAmI.say_hello(); // 汪汪, unknown

Object.setPrototypeOf(whoAmI, cat);
whoAmI.say_hello(); // 喵喵, unknown
```

在JS中实现带记忆的函数

无记忆、效率极低的一段程序

```
1    const fib = n => {
2        if (n == 0) return 0;
3        if (n == 1) return 1;
4
5        return fib(n - 2) + fib (n - 1);
6    }
7
8    console.log(fib(30));
```

带记忆的
斐波那契数列

```
const fib = n => {
    fib.memory = fib.memory || {};

    let rst = fib.memory[n];

    if(rst !== undefined) return rst;

        if(n == 0) rst = 0;
    else if(n == 1) rst = 1;
    else            rst = fib(n - 1) + fib(n - 2);

    fib.memory[n] = rst;

    return rst;
}

console.log(fib(100));
```

使用generator函数实现iota

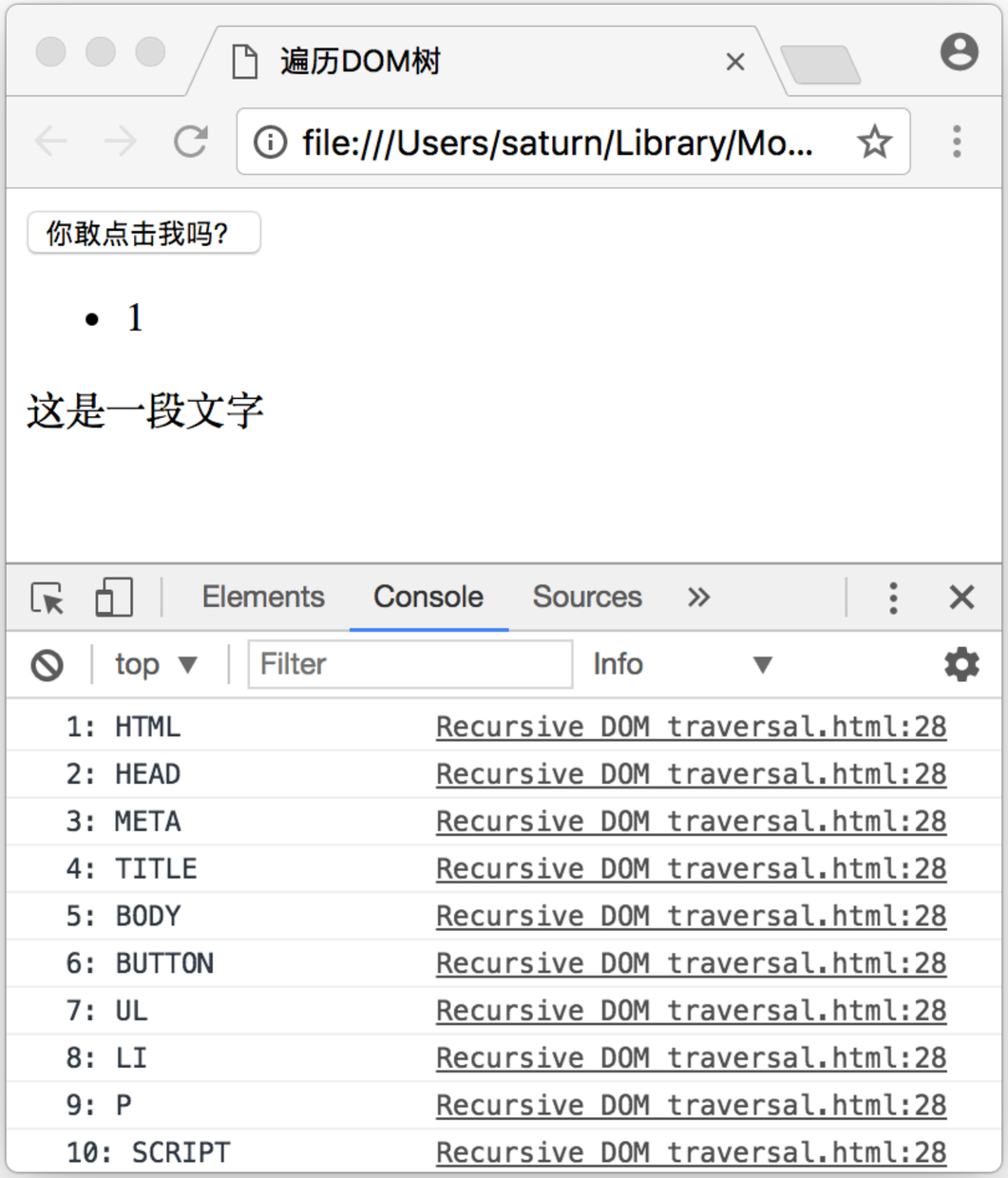
```
const iota = function * (beg, end){
    while(end === undefined || beg < end){
        yield beg;
        beg++;
    }
}

for(let v of iota(0, 5)) console.log(v); // 0 1 2 3 4

const gen = iota(0);

console.log(gen.next().value); // 0
console.log(gen.next().value); // 1
console.log(gen.next().value); // 2
console.log(gen.next().value); // 3
console.log(gen.next().value); // 4
```

使用generator函数
实现无回调函数的DOM树遍历

遍历DOM树：传统方式

```html
<html>
  <head>
    <meta charset="utf-8"/>
    <title>遍历DOM树</title>
  </head>
  <body>
    <button id="btn">你敢点击我吗? </button>
    <ul>
      <li>1</li>
    </ul>
    <p>这是一段文字</p>
    <script type="text/javascript">

      function traversalDOM(element, callback){
        callback(element);
        element = element.firstElementChild;
        while(element){
          traversalDOM(element, callback);
          element = element.nextElementSibling;
        }
      }

      const html = document.querySelector("html");
      let i = 0;

      document.addEventListener("DOMContentLoaded", () =>
        traversalDOM(html, element => {
          console.log(++i + ":", element.nodeName);
        }));
    </script>
  </body>
```

深度优先
中序遍历

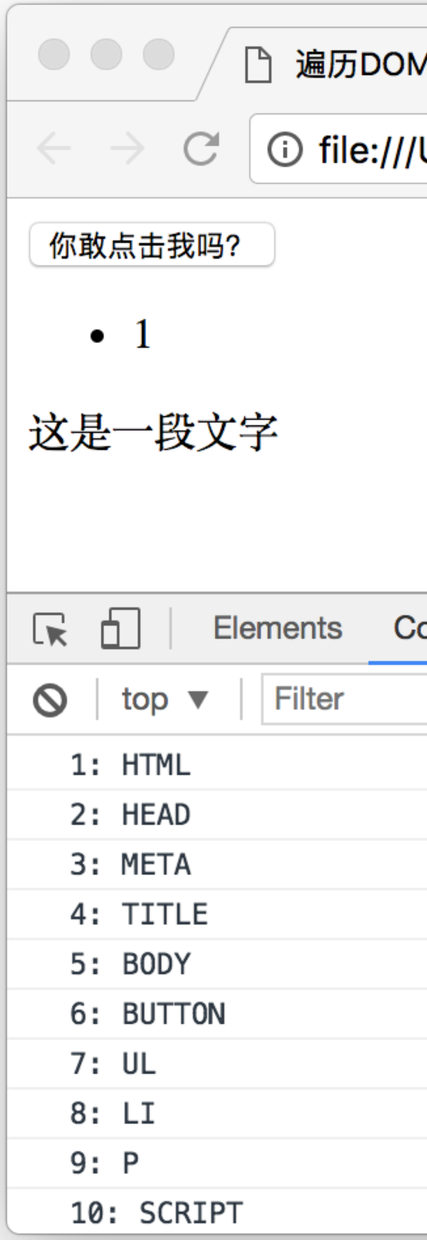


遍历DOM树： generator方式

```
<html>
  <head>
    <meta charset="utf-8"/>
    <title>遍历DOM树</title>
  </head>
  <body>
    <button id="btn">你敢点击我吗? </button>
    <ul>
      <li>1</li>
    </ul>
    <p>这是一段文字</p>
    <script type="text/javascript">

      function * DOMTraversal(element){
        yield element;
        element = element.firstElementChild;
        while(element){
          yield * DOMTraversal(element);
          element = element.nextElementSibling;
        }
      }

      const html = document.querySelector("html");
      let i = 0;

      document.addEventListener("DOMContentLoaded", () => {
        for(let element of DOMTraversal(html))
          console.log(++i + ":", element.nodeName);
        });
    </script>
  </body>
```

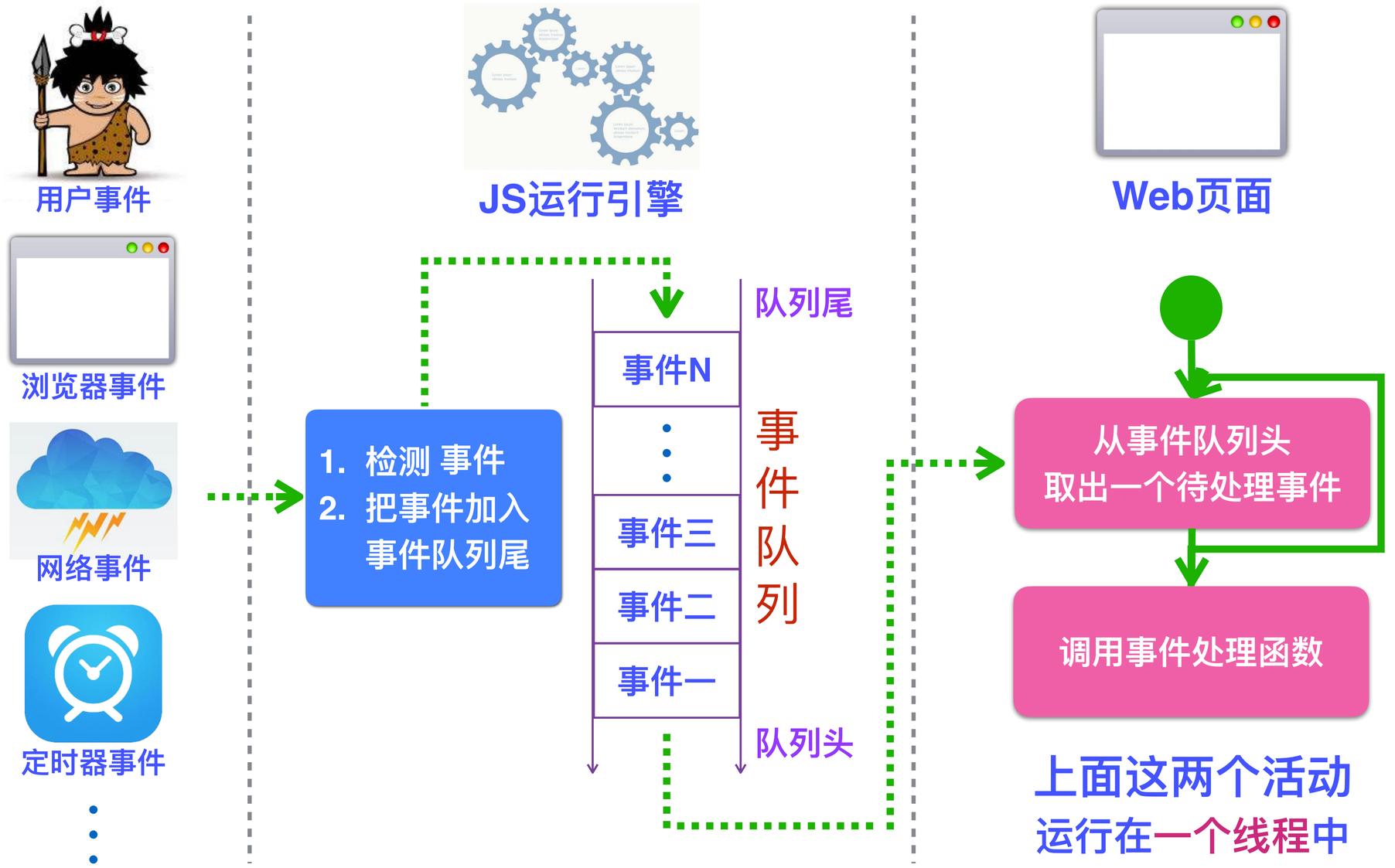
这个函数更纯粹
不涉及回调函数

深度优先
中序遍历
使用generator函数



使用Promise
实现优雅的异步IO编程

Web页面中JS程序的运行环境



异步IO

在Web页面中，经常需要经过从远程服务器读取一些数据	
Web页面中的JS程序通常运行在单个线程中	
为防止线程被数据读取活动阻塞，通常会采用异步IO的方式读取数据	
异步IO	1. 发出数据读取请求时，会传入一个回调函数
	2. 然后线程就去忙别的事情
	3. 当数据读取完毕后，回调函数会被放入到事件队列，等候处理

一个
异步IO
仿真函数

```
/*
 * url : 所要获取数据的url
 * callback: 一个回调函数，具有两个参数 err, data.
 *           err: 当发生错误时，记录错误信息；当没有发生错误时，该参数为假（或类似假的值）
 *           data: 当没有发生错误时，记录获取到的数据；当发生错误时，该参数无效。
 */
const getData = function(url, callback){
    const timePeriod = Math.random() * 5000;
    const isSuccess = Math.random() < 0.5 ? true : false;

    setTimeout( () => {
        if(isSuccess){
            callback(false, "这是你要的数据: " + Math.random());
        } else {
            callback("抱歉，出错了! ");
        }
    }, timePeriod);

};

//测试一下
getData("http://www.pku.edu.cn", (err, data) =>{
    err && console.log(err);
    !err && console.log(data);
});

console.log("我就在这里静静地看你表演...");
```

回调函数的缺点之一： 错误处理

假设要编写一段程序
要求：从一个url读取数据

同步情况下的编程方式

```
1  try {
2      const data1 = getDataSync(url1);
3  } catch(e){
4      console.log(e);
5  }
```

在异步环境里
代码出现顺序与执行顺序无关

异步情况下
用类似的方式
就可以了啊

```
3  try{
4      getData(url1, (err1, data1) => {
5          if(err1) {
6              console.log("出错了，我抛出了错误，你能接到吗？");
7              throw(err1);
8          }
9          console.log(data1);
10     });
11 } catch(e){
12     console.log("错误信息：" + e);
13 }
14
15 console.log("我就在这里静静地看你表演...");
```

回调函数是异步执行的
当执行到第7行时，第15行已经执行完了

孩子，你还太年轻

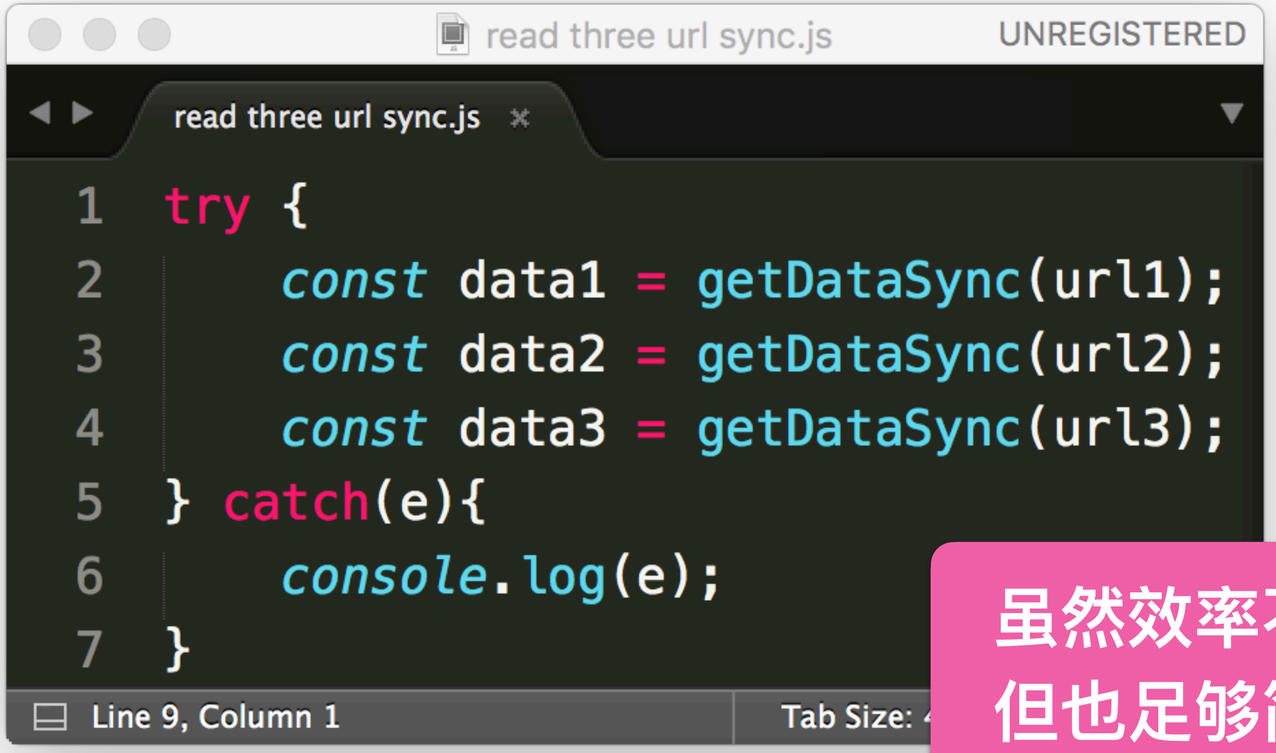
理想形式：像同步情况那样
错误处理与正常处理相互分离

回调函数的缺点之二： 异步顺序任务处理

假设要编写一段程序

要求：按照次序，从三个url读取数据

同步情况下的编程方式



虽然效率不高
但也足够简洁

回调函数的缺点之二： 异步顺序任务处理

假设要编写一段程序

要求：按照次序，从三个url读取数据

异步回调函数情况下的编程方式

```
23  getData(url1, (err1, data1) => {
24      if(err1){
25          console.log("url1:", err1); return;
26      }
27      getData(url2, (err2, data2) => {
28          if(err2){
29              console.log("url2:", err2); return;
30          }
31          getData(url3, (err3, data3) =>{
32              if(err3){
33                  console.log("url3:", err3); return;
34              }
35              console.log("终于成功的获取了所有数据");
36          })
37      })
38  });
```

理想形式：用同步的形式编写异步程序

行为上是异步的，形式上却像同步一样简洁

Promise的使用：示例

```
1  const promise = new Promise((resolve, reject) => {
2      getData("http://www.pku.edu.cn", (err, data) => {
3          if(err){
4              reject(err);
5          } else{
6              resolve(data);
7          }
8      });
9  });
10
11 promise.then(data => {
12     console.log("成功了! ", data);
13 }, err => {
14     console.log("失败了! ", err);
15 });
```

同步调用

- Promise构造函数接收一个函数
- 这个函数在promise对象构造时立即被调用

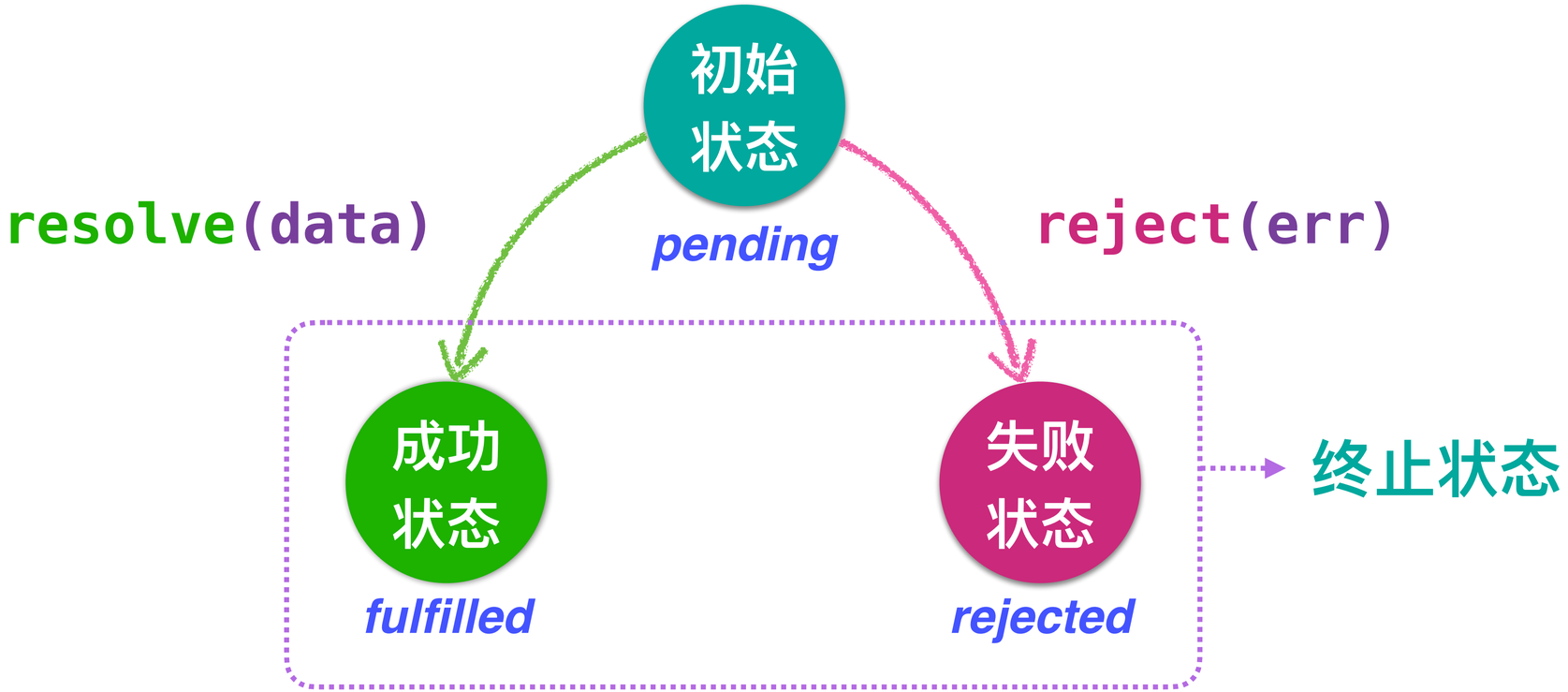
- 这个函数 接收 两个回调函数
- **resolve**: 当任务成功完成后，需调用该函数，相关的数据作为参数传入该函数
- **reject**: 当任务失败后，需调用该函数，相关的数据作为参数传入该函数

当任务执行过程中发生了异常
reject函数会自动被调用

then方法 接收 两个回调函数
分别对应：任务成功处理函数；任务失败处理函数

当需要回调某个函数时
该函数调用会被放入微任务队列中
等待任务处理机制的调度

promise对象：一个状态机



传入失败处理函数：另一种方式

```
const promise = new Promise((resolve, reject) => {
    getData("http://www.pku.edu.cn", (err, data) => {
        if(err){
            reject(err);
        } else{
            resolve(data);
        }
    });
});

promise.then(data => {
    console.log("成功了！", data);
}, err => {
    console.log("失败了！", err);
});
```

```
promise.then(data => {
    console.log("成功了！", data);
}).catch(err => {
    console.log("失败了！", err);
});
```

使用promise进行异步IO操作： 函数封装

```
1  function getDataPromise(url){
2      return new Promise((resolve, reject) => {
3          getData(url, (err, data) => {
4              if(err){
5                  reject(err);
6              } else{
7                  resolve(data);
8              }
9          });
10     });
11 }
12
13 getDataPromise("http://www.pku.edu.cn")
14 .then(data => {
15     console.log("成功了! ", data);
16 })
17 .catch(err => {
18     console.log("失败了! ", err);
19 });
```

这个函数后面还会用到

这个理想
我们已经达到了

理想形式： 像同步情况那样
错误处理与正常处理相互分离

这种情况 promise怎么处理

假设要编写一段程序

要求：按照次序，从三个url读取数据

异步回调函数情况下的编程方式

```
23  getData(url1, (err1, data1) => {
24      if(err1){
25          console.log("url1:", err1); return;
26      }
27      getData(url2, (err2, data2) => {
28          if(err2){
29              console.log("url2:", err2); return;
30          }
31          getData(url3, (err3, data3) =>{
32              if(err3){
33                  console.log("url3:", err3); return;
34              }
35              console.log("终于成功的获取了所有数据");
36          })
37      })
38  });
```

理想形式：用同步的形式编写异步程序

行为上是异步的，形式上却像同步一样简洁

这种情况 promise怎么处理

假设要编写一段程序

要求：按照次序，从三个url读取数据

```
1  const url1 = "", url2 = "", url3 = "";
2
3  getDataPromise(url1)
4  .then(data1 => getDataPromise(url2))
5  .then(data2 => getDataPromise(url3))
6  .then(data3 => console.log("顺序异步读取三个数据，成功！"))
7  .catch(error => console.log("发生错误："+error));
8
9  console.log("我就在这里静静地看你表演...");
```

这个理想
我们已经基本达到了

```
js — -bash — 67×7
Weis-MacBook-Pro:js saturn$ node "read three url async promise.js"
我就在这里静静地看你表演...
发生错误： 网络连接出现异常
Weis-MacBook-Pro:js saturn$ noc
我就在这里静静地看你表演...
顺序异步读取三个数据，成功！
Weis-MacBook-Pro:js saturn$
```

理想形式：用同步的形式编写异步程序
行为上是异步的，形式上却像同步一样简洁

1. JS是一种非常灵活、且学习成本不那么高的语言（相比C++）
2. FP中的很多思想在JS都能找到恰当的实现方式
3. 已经存在很多成熟的第三方库，可以让你在JS中方便地进行函数式编程

三个比较知名的第三方库

lodash / lodash

Code Issues 85 Pull requests 70 ...

FP Guide

Jump to bottom

Izaak Schroeder edited this page on May 4, 2020 · 5 revisions

lodash/fp

The `lodash/fp` module promotes a more functional programming (FP) friendly style by exporting an instance of `lodash` with its methods wrapped to produce immutable auto-curried iteratee-first data-last methods.

UNDERSCORE.JS

Underscore is a JavaScript library that provides a whole mess of useful functional programming helpers without extending any built-in objects. It's the answer to the question: "If I sit down in front of a blank HTML page, and want to start being productive immediately, what do I need?" … and the tie to go along with jQuery's tux and Backbone's suspenders.

Ramda

A practical functional library for JavaScript programmers.

build passing | npm package 0.27.1 | dependencies none | gitter join chat

Why Ramda?

There are already several excellent libraries with a functional flavor. Typically, they are meant to be general-purpose toolkits, suitable for working in multiple paradigms. Ramda has a more focused goal. We wanted a library designed specifically for a functional programming style, one that makes it easy to create functional pipelines, one that never mutates user data.

What's Different?

The primary distinguishing features of Ramda are:

- Ramda emphasizes a purer functional style. Immutability and side-effect free functions are at the heart of its design philosophy. This can help you get the job done with simple, elegant code.

2. FP in JavaScript

暂且结束了吧

第三部分

函数式编程思想

在其他编程语言中的应用

0	Introduction
1	FP in C++ 2020
2	FP in JavaScript

以上内容

不会出现在

本课程的任何考试/作业中