阿里云 | 奥运会全球指定云服务商

C++20协程与应用

阿里云-程序语言与编译器

WWW.ALIYUN.COM

目录

1、协程

阿里云 | 奥运会全球指定云服务商

什么是协程？

- 协程是一种程序组件，是由子例程（过程、函数、例程、方法、子程序）的概念泛化而来的，子例程只有一个入口点且只返回一次，而协程允许多个入口点，可以在指定位置挂起和恢复执行。
  - ✓ 协程在控制离开时暂停执行，当控制再次进入时只能从离开的位置继续执行。
  - ✓ 协程的本地数据在后续调用中持久化。

为什么要用协程？

- VS 多线程
  - ✓ uthread：极高的执行&切换效率；
- VS 异步回调；
  - ✓ 统一的同步编程模式， 避免callback hell

协程分类：

libeasy, libco, boost.coroutine, go coroutine, C# await…

- 控制传递（Control-transfer）机制 ： 对称/非对称 （return-to-caller）
- 是否作为语言的第一类（First-class）对象提供 ：语言特性&编译器支持
- 栈式（Stackful）/无栈（Stackless）构造：是否在内部的嵌套调用中挂起（性能&挂起位置）

2、C++20协程

阿里云 | 奥运会全球指定云服务商

- C++20语言的重要的特性（Big Four - concept,ranges,module,coroutine）；
- C++20协程：对称、语言第一类支持、无栈
- 三个关键字：co_await、co_yield、co_return

一个例子：

```
Resumable func () {
  std::cout << "Hello" << std::endl;
  co_await std::experimental::suspend_always();
  std::cout << "Coroutine" << std::endl;
}
```

一个例子：

```
Resumable func () {
  std::cout << "Hello" << std::endl;
  co_await std::experimental::suspend_always();
  std::cout << "Coroutine" << std::endl;
}
```

```
$clang++ -std=c++17 -O0 -c x.cc -fcoroutines-ts -stdlib=libc++
x.cc:10:11: error: this function cannot be a coroutine: 'std::experimental::coroutines_v1::coroutine_traits<Resumeble>' has no
      member named 'promise_type'
Resumeble func() {
          ^
1 error generated.
```

如何实现Resumable?

- 协程句柄（coroutine handle）& 协程对象；
- promise_type结构；

编译器第一次改造：

```
Resumable func(args...){
  Frame *frame_ = operator new(std::size_t size);
  Promise_type promise;
  coroutine_handle *handle_ = coro_handle::from_promise(&promise);
  Resumable ret = promise.get_return_object();

  co_await promise.initial_suspend();

  try {
    // co-routine body
  } catch(...) {
    promise.unhandled_exception();
  }

final_suspend:
  co_await promise.final_suspend();

  return ret;
}
```

1、编译器选择promise_type的两个方法：

- Resumable里定义promise_type类型
- 特例化coroutine_traits<Resumable, Args...>类型

2、Coroutine_handle:和协程对象一一对应，可以恢复/销毁协程。在experimental/coroutine中实现。

3、返回Resumable对象

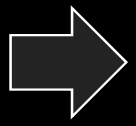
程序员实现： promise_type 、 Resumable

实现Resumable:

```
class Resumable {
public:
  struct promise_type;
  bool resume() {
    if (!handle_.done())
      handle_.resume();
    return !handle_.done();
  }
  coro_handle handle_; // 协程句柄
};

struct Resumable::promise_type {
  Resumable get_return_object() {/**/}
  auto initial_suspend() {/**/}
  auto final_suspend() {/**/}
  void return_void() {}
  void unhandled_exception();
};
```

编译器第一次改造：

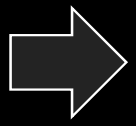
```
Resumable func () {
  std::cout << "Hello" << std::endl;
  co_await std::experimental::suspend_always();
  std::cout << "Coroutine" << std::endl;
}
```

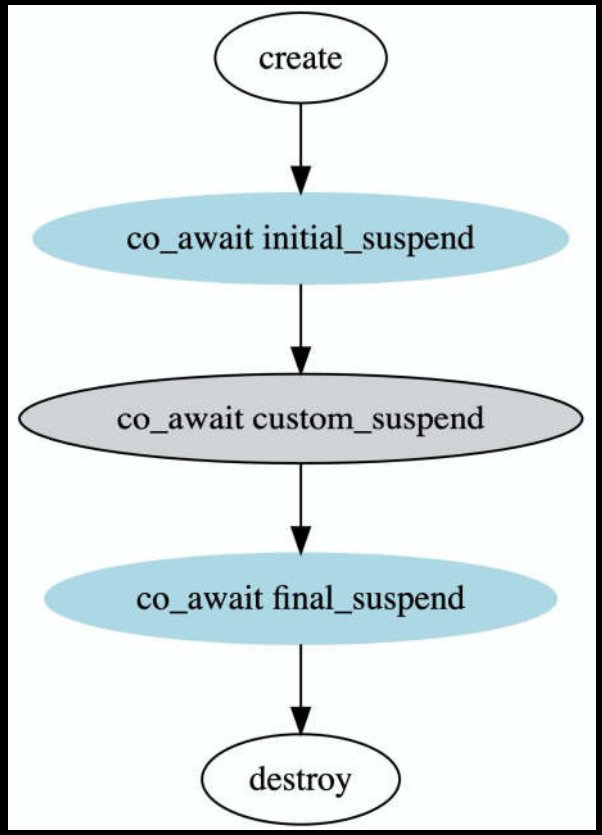


```
Resumable func(args...) {
  Frame *frame_ = operator new(std::size_t size);
  Promise_type promise;
  coroutine_handle *handle_ = coro_handle::from_promise(&promise);
  Resumable ret = promise.get_return_object();

  co_await promise.initial_suspend();

  try {

    // co-routine body
    std::cout << "Hello" << std::endl;
    co_await std::experimental::suspend_always();
    std::cout << "Coroutine" << std::endl;

  } catch(...) {
    promise.unhandled_exception();
  }

final_suspend:
  co_await promise.final_suspend();

  return ret;
}
```

编译器第一次改造：

```
Resumable func () {
  std::cout << "Hello" << std::endl;
  co_await std::experimental::suspend_always();
  std::cout << "Coroutine" << std::endl;
}
```



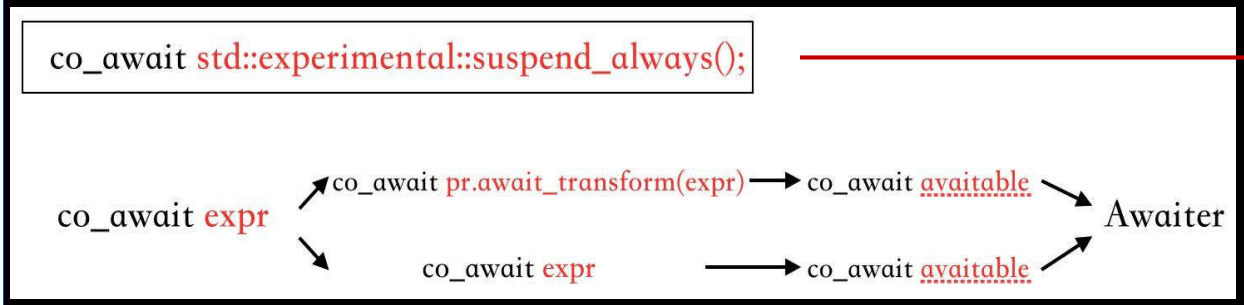


Awaiter

```
co_await std::experimental::suspend_always();
```


```
struct Awaiter {
  bool await_ready() {...}
  auto await_suspend(coroutine_handle<>) {...}
  auto await_resume() {...}
};
```

Awaiter




```
struct Awaiter {
  bool await_ready() {...}
  auto await_suspend(coroutine_handle<>) {...}
  auto await_resume() {...}
};
```

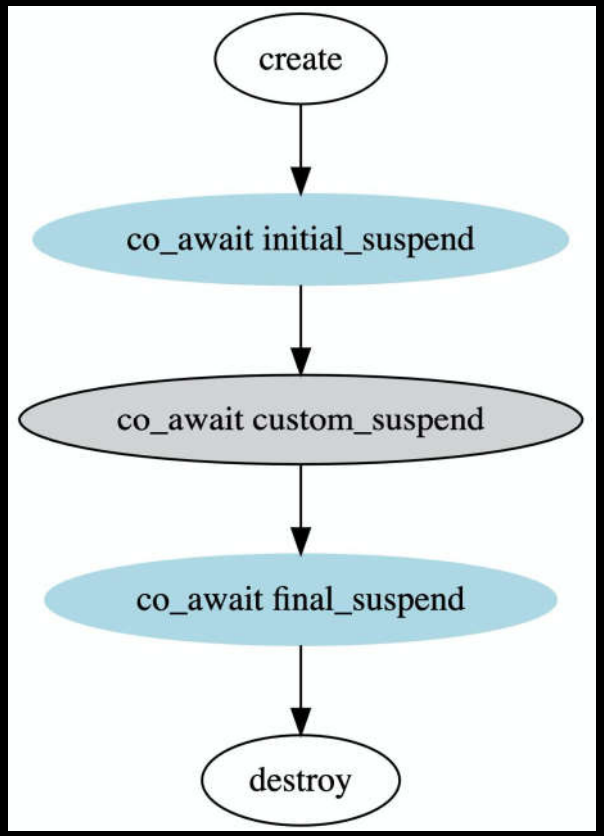
```
//1: if await_suspend returns void
if (not a.await_ready()) {
  try {
    a.await_suspend(coroutine_handle);
    return_to_the_caller();
  } catch (...) {
    exception = std::current_exception();
    goto resume_point;
  }
  //endif
}

resume_point:
  if(exception)
    std::rethrow_exception(exception);
  return a.await_resume();
```

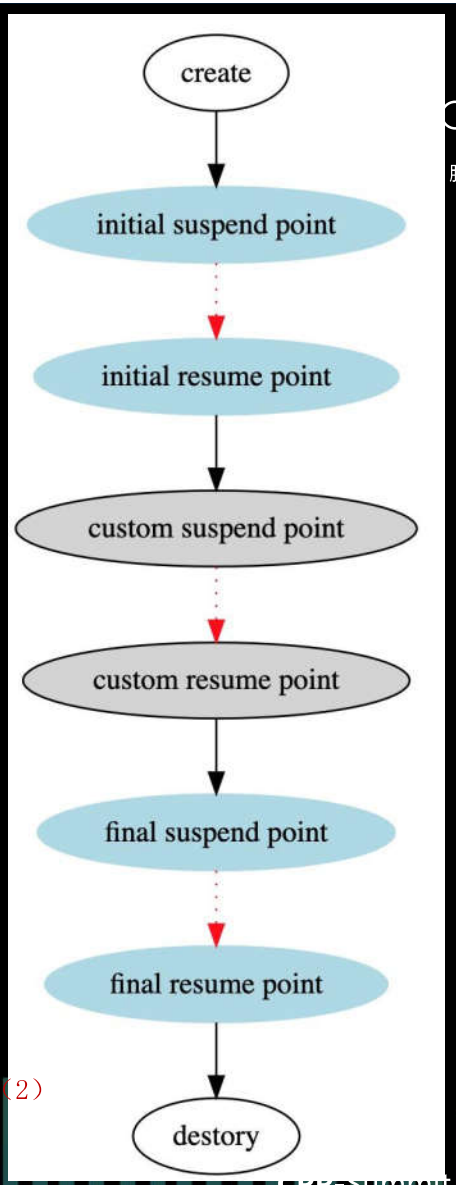
```
//2: if await_suspend returns bool
if (not a.await_ready()) {
  bool await_suspend_result;
  try {
    await_suspend_result = a.await_suspend(coroutine_handle);
  } catch (...) {
    exception = std::current_exception();
    goto resume_point;
  }
  if (not await_suspend_result)
    goto resume_point;
  return_to_the_caller();
  //endif
}
resume_point:
  if(exception)
    std::rethrow_exception(exception);
  return a.await_resume();
```

```
//3: if await_suspend returns another coroutine_handle
if (not a.await_ready()) {
  decltype(a.await_suspend(std::declval<coro_handle_t>())) another_coro_handle;
  try {
    another_coro_handle = a.await_suspend(coroutine_handle);
  } catch (...) {
    exception = std::current_exception();
    goto resume_point;
  }
}
another_coro_handle.resume();
return_to_the_caller();
  //endif

resume_point:
  if(exception)
    std::rethrow_exception(exception);
  return a.await_resume();
```

编译器第二次改造：

```
Resumable func () {
  std::cout << "Hello" << std::endl;
  co_await std::experimental::suspend_always();
  std::cout << "Coroutine" << std::endl;
}
```



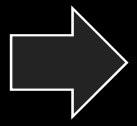


编译器改造（1）

编译器改造（2）

编译器第三次改造：

```
Resumable func () {
  std::cout << "Hello" << std::endl;
  co_await std::experimental::suspend_always();
  std::cout << "Coroutine" << std::endl;
}
```



```
// ramp function
Resumable func.ramp(args...){
  Frame *frame_ = operator new(std::size_t size);
  {
    // construct coroutine frame, there are fields such as:
    promise_type obj
    args...
    local variables
    // also there are following fields which relate to coroutine suspend/resume
    frame_.index = 0
    frame_.resume = func.resume;
    frame_.destroy = func.destroy
  }
  coroutine_handle r = frame_->promise.get_return_object();
  r.resume()  // eventually call func.resume()
  return Resumable(r);
}
```

```
// resume function
void func.resume(Frame *frame_) {
  switch(frame_.index) {
    case 0:
      // ...
      frame_.index = 1;
      initial suspend point;

    case 1:
      // initial resume point
      // ...
      frame_.index = 2;
      custom suspend point;

    case 2:
      // custom resume point
      // ...
      frame_.index = 3;
      final suspend point;

    // case ...:

    case N:
      // final resume point
      frame_.destroy() // func.destroy()
  }
}
```

```
// destroy function
void func.destroy(Frame *frame_) {
   // release coroutine's resources and destroy the coroutine object
}
```

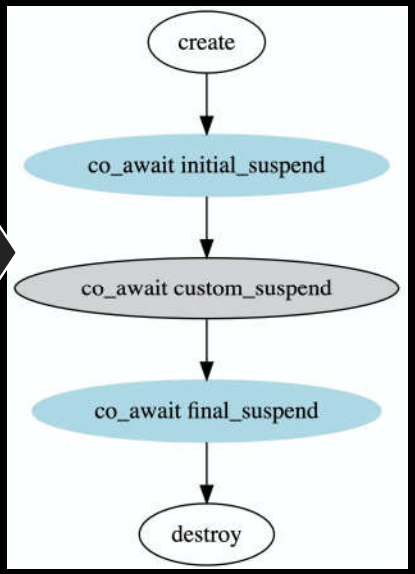
1、func.ramp:负责申请协程帧，保存协程参数、局部变量、promise obj、index、resume函数指针、destroy函数指针等

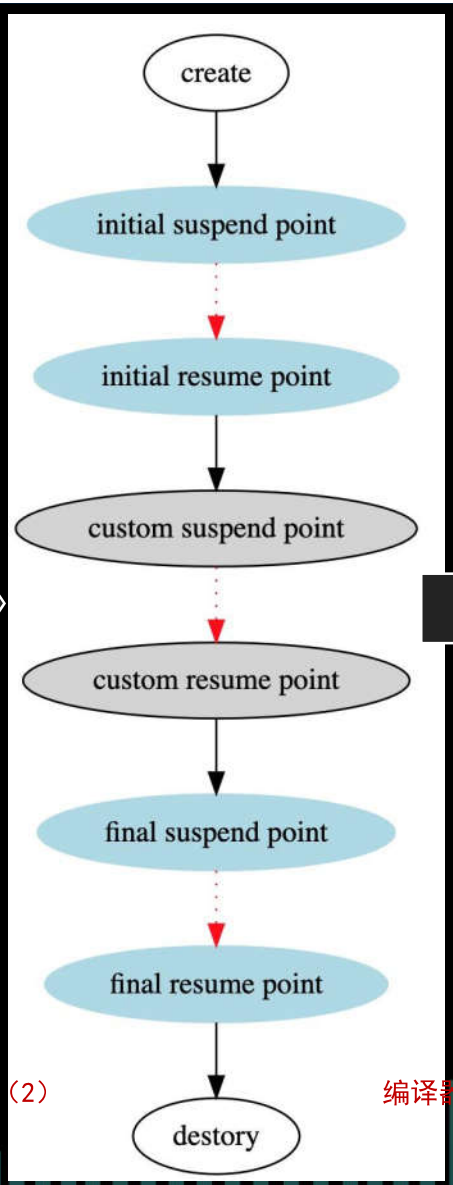
2、func.resume:简单状态机，根据frame_.index，选择恢复点。

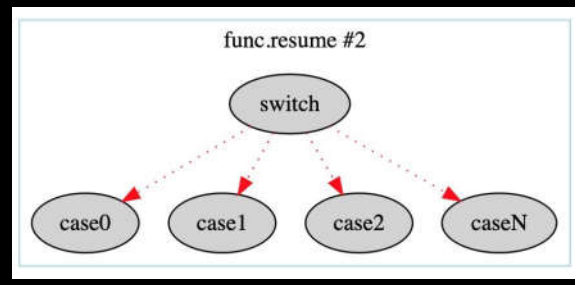
3、func.destroy：回收资源

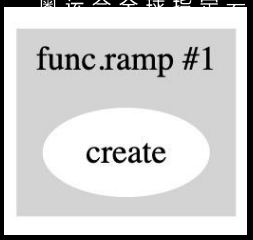
编译器的三次改造：

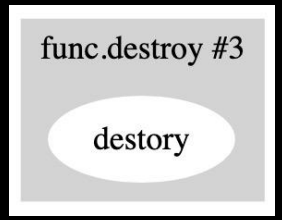
```
Resumable func () {
  std::cout << "Hello" << std::endl;
  co_await std::experimental::suspend_always();
  std::cout << "Coroutine" << std::endl;
}
```











编译器改造（1）

编译器改造（2）

编译器改造（3）

实例执行：

```
#include "resumable.h"

Resumable func () {
  std::cout << "Hello" << std::endl;
  co_await std::experimental::suspend_always();
  std::cout << "Coroutine" << std::endl;
}

int main() {
  Resumable res = func();
  while(res.resume())
    ;
  return 0;
}
/*
Hello
Coroutine
*/
```

3、编译器相关工作

GCC VS LLVM实现策略

- 协程函数展开
- 协程函数拆分
- Inline vs Split
- GCC在AST生成IR时就将协程函数拆开，优化策略为对已经生成好的协程函数进行分析和优化（Inline）
- LLVM则是基于IR做协程函数的拆分，利用已有优化对协程函数优化后，再拆分（Split）并优化

阿里云 | 奥运会全球指定云服务商

阿里巴巴从2019年10月开始C++协程相关工作

- GCC
  - ✓ 与社区合作进行协程的支持。
  - ✓ GCC-10将是第一个支持C++协程特性的GCC编译器。
  - ✓ 仅支持，无优化。

- LLVM
  - ✓ 与Clang/LLVM社区合作完善C++协程。
  - ✓ Clang/LLVM-11版本将具有成熟的协程特性支持，集团内Clang/LLVM-8稳定支持协程。
  - ✓ 改善&优化：协程逃逸分析和CoroElide优化，协程帧优化（Frame reduction），改进协程调试信息、use-after-free、尾调用优化等。

4、future_lite协程库

阿里云 | 奥运会全球指定云服务商

现状

- 缺少STL标准库支持，C++23可能会支持；

- 典型第三方协程库
  - ✓ 开源的cppcoro库
  - ✓ 阿里巴巴集团future_lite库（借鉴了facebook/folly库）

future_lite库

- Lazy<T>组件
- future/promise组件
- 协程执行器-Executor
- 批量执行组件
- Generator/AsyncGenerator组件
- 异步Mutex/ConditionVariable 组件
- Barrier、Latch、Semaphore、SharedMutex、SpinLock、SharedLock等异步组件

future_lite库应用实例：

```
// 协程函数getValue
template<typename T>
Lazy<T> getValue(T x) {

  struct ValueAwaiter {
  T value;
  ValueAwaiter(T v) : value(v) {}
  bool await_ready() { return false; }
  void await_suspend(std::experimental::coroutine_handle<> continuation) noexcept {
    std::thread([c = continuation]() mutable { c.resume(); }).detach();
  }
  T await_resume() noexcept { return value; }
  };

  co_return co_await ValueAwaiter(x);
}

// 协程函数task2, 默认T = void
Lazy<> task2() {
    auto t = getValue(10);
    auto x = co_await t; //恢复协程getValue/t的执行
    assert(x == 10);
}

// 普通函数func
void func() {
    SimpleExecutor exec;
    auto t1 = task1(10)
    auto x = syncAwait(t1.via(&exec)); // 指定执行器exec执行协程task1/t
    assert(x == 10);
}
```

```
Generator<std::uint64_t> fibonacci() {
    std::uint64_t a = 0, b = 1;
    while (true) {
        co_yield b; // yield a value
        auto tmp = a;
        a = b;
        b += tmp;
    }
};

for (auto i : fibonacci()) {
    if (i > 100'000) break;
    std::cout << i << std::endl;
}
```

future_lite库性能：

- 有栈协程：对比C++20协程和libeasy协程

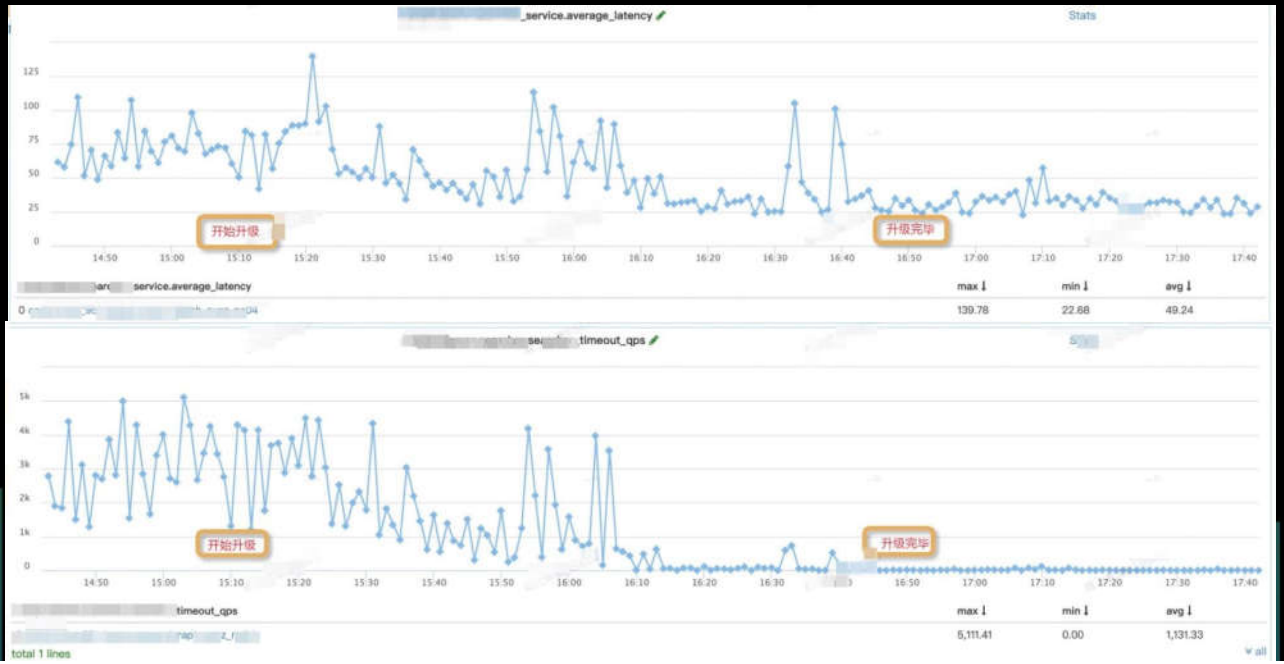
  注：libeasy阿里巴巴集团内部协程库
- 无栈协程：future_lite库和cppcoro库的性能

	future_lite	cppcoro	libeasy
场景1：单协程重复执行	181 ns	271 ns	< 100 us
场景2：线程池+海量协程任务	217 ms	458.8 ms	272 ms
场景3：协程调用链	252 ns	280 ns	—
场景4：批量执行接口	219.3 ms	302.6 ms	—

5、业务落地

阿里集团搜索、交互式分析等业务：

- 全链路改造
  - ✓ 查询、计算、存储
  - ✓ 方案：C++20协程 + 自研调度器 + 异步io改造。



	上线前	上线后
Rt	50 ~ 100 ms	30ms
Timeout qps	2k ~ 4k	0

6、未来

- LLVM
  - ✓ Add missing debugInfo
  - ✓ IPO CoroElide
  - ✓ Safe stack&frame check
- future_Lite
  - ✓ 开源
  - ✓ 有栈&无栈、同步&异步混合模式
  - ✓ 支持更多IO/Network接口

奥运会全球指定云服务商

WWW.ALIYUN.COM