

一些和现代C++关联不大的C++知识（只是突然想回忆一下）

C++程序的构建过程通常分为四个主要阶段：**预处理、编译、汇编和链接**：

- 预处理：预处理器（cpp）接收C++源代码作为输入，**处理源代码文件中以#开头的预处理指令**，这包括宏定义的展开（#define）、条件编译指令（如#ifndef、#ifndef、#endif）以及包含头文件的指令（#include）。g++ -E
- 编译：由编译器（如g++或clang++）处理。在这个阶段，编译器将**预处理后的源代码转换成汇编语言代码**。g++ -S
- 汇编：由汇编器（如as）处理。汇编器将**编译阶段生成的汇编语言代码转换成机器语言指令**，通常是**目标代码（object code）**。这些目标代码是以二进制形式表示的，但在这个阶段，代码还不能直接运行，因为它可能还依赖于其他的目标文件或库。g++ -c
- 链接：由链接器（如ld）处理。在这个阶段，链接器将**一个或多个目标文件以及必要的库文件合并成一个单一的可执行文件**。g++

g++和gcc都是GNU编译器套件（GNU Compiler Collection，GCC）的一部分：

- g++ 是专门用于编译C++代码的编译器前端。它自动链接C++程序所需的C++标准库，比如libstdc++。当你使用g++编译C++代码时，不需要显式指定链接这些库。
- gcc 最初是作为GNU C编译器开发的，用于编译C语言代码。随着时间的推移，它成为了一个多语言编译器前端，可以通过不同的前端来支持多种编程语言，包括C、C++、Objective-C、Fortran、Ada等。

gcc命令也可以用来编译C++文件，需要显式地添加-lstdc++标志来链接C++标准库，否则在链接时可能会出现未定义的引用错误。但通常推荐使用g++来编译C++程序，因为g++会自动处理一些C++特定的编译和链接步骤，使得编译过程更加简洁和直接。

```
gcc example.cpp -lstdc++ -o example
```

迈向现代C++

C++不是C的超集。

在编写C++时，也应该尽可能的避免使用诸如void*之类的程序风格。而在不得不使用C时，应该注意使用extern "C"这种特性（确保当这个头文件被C++代码包含时，以C语言的链接方式进行编译），将C语言的代码与C++代码进行分离编译，再统一链接。

```
// foo.h
#ifndef __cplusplus
extern "C" {
#endif

int add(int x, int y);

#ifndef __cplusplus
}
#endif
```

用两个条件编译指令是因为只有在C++编译过程中才需要extern "C"

```
gcc -c foo.c
# 再和C++文件一起链接
```

语言可用性的强化

语言可用性是指程序运行之前的行为，包括编写代码和编译器编译过程中产生的行为。

变量

隐式转换: 编译器自动进行的类型转换，无需程序员干预。这种转换通常发生在当表达式中涉及多种不同的数据类型时，编译器会自动将其中某些类型转换为其他类型，以满足操作的需求。

```
int i = 5;
double d = 6.7;
auto result = i + d; // i 被隐式转换为 double
```

显式转换: 程序员明确指定的类型转换。在C++中，显式转换可以通过四种C++风格的转换操作符实现：static_cast、dynamic_cast、const_cast和reinterpret_cast，以及传统的C风格类型转换。

```
double d = 9.5;
int i = static_cast<int>(d); // 显式转换
```

nullptr

nullptr出现的目的是为了替代NULL。传统C++会把NULL、0视为同一种东西，这取决于编译器如何定义NULL，有些编译器会将NULL定义为((void*)0)，有些则会直接将其定义为0。

C++不允许直接将void *隐式转换到其他类型。但如果编译器尝试把NULL定义为((void*)0)，那么在下面这句代码中：

```
char *ch = NULL;
```

没有了void *隐式转换的C++只好将NULL定义为0。而这依然会产生新的问题，将NULL定义成0将导致C++中重载特性发生混乱。考虑下面这两个foo函数：

```
void foo(char*);
void foo(int);
```

那么foo(NULL);这个语句将会去调用foo(int)，从而导致代码违反直觉（**实际过程中这个语句不能编译**）。为了解决这个问题，C++11引入了nullptr关键字，专门用来区分空指针、0。而nullptr的类型为nullptr_t，能够隐式的转换为任何指针或成员指针的类型，也能和他们进行相等或者不等的比较。

constexpr

C++11提供了constexpr让用户显式的声明函数或对象构造函数在编译期会成为常量表达式，从C++14开始，constexpr函数可以在内部使用局部变量、循环和分支等简单语句。这将能增加程序的性能，因为调用函数会使用堆栈。

```
constexpr int len_foo_constexpr() {
    return 5;
}

int nums[len_foo_constexpr()] // 在编译期间这里会直接变成int nums[5]，而不是在运行期间
```

变量及其初始化

if/switch变量声明强化

c++17中可以在if/switch中声明临时变量：

```
// 将临时变量放到 if 语句内
if (const std::vector<int>::iterator itr = std::find(vec.begin(), vec.end(), 3);
    itr != vec.end()) {
    *itr = 4;
}
```

这和go很像

```
if v, found := someMap["key"]; found {
    fmt.Println("Found:", v)
} else {
    fmt.Println("Not found")
}
```

初始化列表

在C++11之前，初始化容器（如std::vector, std::map等）时，通常需要逐个插入元素。使用初始化列表（std::initializer_list）后，可以直接在容器创建时初始化所有元素，大大简化了代码。

```
std::vector<int> v = {1, 2, 3, 4, 5};
std::map<int, std::string> m = {{1, "one"}, {2, "two"}, {3, "three"}};
```

初始化对象：

```
class MagicFoo {
public:
    std::vector<int> vec;
    MagicFoo(std::initializer_list<int> list) {
        for (std::initializer_list<int>::iterator it = list.begin();
            it != list.end(); ++it)
            vec.push_back(*it);
    }
    void foo(std::initializer_list<int> list) {
        for (std::initializer_list<int>::iterator it = list.begin();
            it != list.end(); ++it) vec.push_back(*it);
    }
};
MagicFoo magicFoo = {1, 2, 3, 4, 5};
```

作为形参：

```
magicFoo.foo({6, 7, 8, 9});
```

结构化绑定

在go中函数可以返回多个相同/不同类型的值，而c++通常只能返回一个值（普遍的做法是传引用），c++11之前则可以通过std::pair返回两个值，c++11后新增了std::tuple容器用于构造一个元组，进而囊括多个返回值。但缺陷是，C++11/14并没有提供一种简单的方法直接从元组中拿到并定义元组中的元素，尽管我们可以使用std::tie对元组进行拆包，但我们依然必须非常清楚这个元组包含多少个对象，各个对象是什么类型，非常麻烦。

C++17完善了这一设定，给出的结构化绑定可以让我们写出这样的代码：

```
#include <iostream>
#include <tuple>

std::tuple<int, double, std::string> f() {
    return std::make_tuple(1, 2.3, "456");
}

int main() {
    auto [x, y, z] = f();
    std::cout << x << ", " << y << ", " << z << std::endl;
    return 0;
}
```

类型推导

auto

auto就不说了，迭代器里面经常用，很好用。

decltype

decltype关键字是为了解决auto关键字只能对变量进行类型推导的缺陷而出现的。它的用法和typeof很相似：

```
decltype(表达式)
```

尾返回类型

c++14开始可以直接让普通函数具备返回值推导：

```
template<typename T, typename U>
auto add3(T x, U y) {
    return x + y;
}
```

也可以用decltype(auto)， decltype(auto)的优势在于能够正确处理更复杂的类型推导场景，比如当函数返回一个变量的引用或者带有cv限定符的类型时。 decltype(auto)会保留返回表达式的完整类型，包括其引用性和cv限定符，而auto则不会。

控制流

if constexpr

c++17后constexpr可以用在if语句中来优化性能。

区间for迭代

这个就用的很多了，注意需要用引用来可写：

```
for (auto element : vec)
    std::cout << element << std::endl; // 只是拷贝, read only
for (auto &element : vec) {
    element += 1;                      // writeable
}
```

模板

实际开发中模板其实用的不多（除非造轮子写基础库

外部模板

传统C++中，模板只有在使用时才会被编译器实例化。换句话说，只要在每个编译单元（文件）中编译的代码中遇到了被完整定义的模板，都会实例化。这就产生了重复实例化而导致的编译时间的增加。并且，我们没有办法通知编译器不要触发模板的实例化。

为此，C++11引入了外部模板，扩充了原来的强制编译器在特定位置实例化模板的语法，使我们能够显式的通知编译器何时进行模板的实例化：

```
template class std::vector;      // 强行实例化，通常是在定义模板或专门实例化模板的文件中
extern template class std::vector; // 不在该当前编译文件中实例化模板，其他使用模板的文件中都应该声明
```

类型别名模板

C++11使用using支持模板别名：

```
template<typename T>
using TrueDarkMagic = MagicType<std::vector<T>, std::string>;

int main() {
    TrueDarkMagic<bool> you;
}
```

变长参数模板

在C++11之前，无论是类模板还是函数模板，都只能按其指定的样子，接受一组固定数量的模板参数；而C++11加入了新的表示方法，允许任意个数、任意类别的模板参数，同时也不需要在定义时将参数的个数固定。

```
template<typename... Ts> class Magic;
```

模板类 Magic 的对象，能够接受不受限制个数的 typename 作为模板的形式参数，例如下面的定义：

```
class Magic<int, std::vector<int>, std::map<std::string, std::vector<int>>>
darkMagic;
```

既然是任意形式，所以个数为0的模板参数也是可以的：class Magic<> nothing;。

如果不希望产生的模板参数个数为 0，可以手动的定义至少一个模板参数：

```
template<typename Require, typename... Args> class Magic;
```

变长参数模板也能被直接调整到到模板函数上。除了在模板参数中能使用...表示不定长模板参数外，函数参数也使用同样的表示法代表不定长参数，这也就为我们简单编写变长参数函数提供了便捷的手段，例如：

```
template<typename... Args> void printf(const std::string &str, Args... args);

// 计算参数个数可用以下办法：
template<typename... Ts>
void magic(Ts... args) {
    std::cout << sizeof...(args) << std::endl;
}

// 展开
template<typename T0, typename... T>
void printf2(T0 t0, T... t) {
    std::cout << t0 << std::endl;
    if constexpr (sizeof...(t) > 0) printf2(t...);
}
```

折叠表达式

```
#include <iostream>
template<typename ... T>
auto sum(T ... t) {
    return (t + ...);
}
int main() {
    std::cout << sum(1, 2, 3, 4, 5, 6, 7, 8, 9, 10) << std::endl;
}
```

非类型模板参数

类型模板参数：

```
template <typename T, typename U>
auto add(T t, U u) {
    return t+u;
}
```

非类型模板参数：

```

template <typename T, int BufSize>
class buffer_t {
public:
    T& alloc();
    void free(T& item);
private:
    T data[Bufsize];
}

buffer_t<int, 100> buf; // 100 作为模板参数

```

C++17可以使用auto关键字，让编译器辅助完成具体类型的推导：

```

template <auto value> void foo() {
    std::cout << value << std::endl;
    return;
}

int main() {
    foo<10>(); // value 被推导为 int 类型
}

```

面向对象

委托构造

C++11引入了委托构造的概念，这使得构造函数可以在同一个类中一个构造函数调用另一个构造函数：

```

#include <iostream>
class Base {
public:
    int value1;
    int value2;
    Base() {
        value1 = 1;
    }
    Base(int value) : Base() { // 委托 Base() 构造函数
        value2 = value;
    }
};

```

继承构造

```

#include <iostream>
class Base {
public:
    int value1;
    int value2;
    Base() {
        value1 = 1;
    }
    Base(int value) : Base() { // 委托 Base() 构造函数
        value2 = value;
    }
};

```

```
    }
};

class SubClass : public Base {
public:
    using Base::Base; // 继承构造
};
```

显示虚函数构造

override

当重载虚函数时，引入override关键字将显式的告知编译器进行重载，编译器将检查基函数是否存在这样的其函数签名一致的虚函数，否则将无法通过编译：

```
struct Base {
    virtual void foo(int);
};

struct SubClass: Base {
    virtual void foo(int) override; // 合法
    virtual void foo(float) override; // 非法，父类没有此虚函数
};
```

final

final则是为了防止类被继续继承以及终止虚函数继续重载引入的。

```
struct Base {
    virtual void foo() final;
};

struct SubClass1 final: Base {
}; // 合法

struct SubClass2 : SubClass1 {
}; // 非法，SubClass1 已 final

struct SubClass3: Base {
    void foo(); // 非法，foo 已 final
};
```

显式禁用默认函数

c++11允许显式的声明采用或拒绝编译器自带的函数（构造函数、析构函数、复制构造和赋值运算符）。

```
class Magic {
public:
    Magic() = default; // 显式声明使用编译器生成的构造
    Magic& operator=(const Magic&) = delete; // 显式声明拒绝编译器生成构造
    Magic(int magic_number);
}
```

强类型枚举

C++11 入了枚举类 (enumeration class) , 并使用enum class的语法进行声明:

```
enum class new_enum : unsigned int {
    value1,
    value2,
    value3 = 100,
    value4 = 100
};
```

这样定义的枚举实现了类型安全, 首先他不能够被隐式的转换为整数, 同时也不能够将其与整数数字进行比较, 更不可能对不同的枚举类型的枚举值进行比较。但相同枚举值之间如果指定的值相同, 那么可以进行比较。想获得枚举值的值时, 将必须显式的进行类型转换, 不过我们可以通过重载<<来进行输出。

语言运行期的强化

Lambda表达式

Lambda表达式是现代C++中最重要的特性之一，Lambda表达式，实际上就是提供了一个类似匿名函数的特性，而匿名函数则是在需要一个函数，但是又不想费力去命名一个函数的情况下使用的。

基础

Lambda表达式的基础语法如下：

```
[捕获列表] (参数列表) mutable(可选) 异常属性 -> 返回类型 {  
// 函数体  
}
```

捕获列表起到一个传递参数的作用，默认情况下函数体内部不能使用函数体外部的变量，这时就可以通过捕获列表来传递外部变量。

捕获列表可分为以下几种：

值捕获

与参数传值类似，值捕获的前提是变量可以拷贝（如果一个类的拷贝构造函数被删除（使用= delete标记）或者未定义，那么这个类的对象就不能被拷贝。例如，标准库中的std::unique_ptr就是不可拷贝的），不同之处则在于，被捕获的变量在Lambda表达式被创建时拷贝，而非调用时才拷贝：

```
void lambda_value_capture() {  
    int value = 1;  
    auto copy_value = [value] {  
        return value;  
    };  
    value = 100;  
    auto stored_value = copy_value();  
    std::cout << "stored_value = " << stored_value << std::endl;  
    // 这时，stored_value == 1, 而 value == 100.  
    // 因为 copy_value 在创建时就保存了一份 value 的拷贝  
}
```

引用捕获

与引用传参类似，引用捕获保存的是引用，值会发生变化。

```
void lambda_reference_capture() {  
    int value = 1;  
    auto copy_value = [&value] {  
        return value;  
    };  
    value = 100;  
    auto stored_value = copy_value();  
    std::cout << "stored_value = " << stored_value << std::endl;  
    // 这时，stored_value == 100, value == 100.  
    // 因为 copy_value 保存的是引用  
}
```

隐式捕获

可以在捕获列表中写一个&或=向编译器声明采用引用捕获或者值捕获。捕获列表最常用的四种形式是：

- []空捕获列表
- [name1, name2]捕获一系列变量
- [&]引用捕获，让编译器自己推导捕获列表
- [=]值捕获，让编译器自己推导捕获列表

表达式捕获

C++14允许捕获的成员用任意的表达式进行初始化，这就允许了右值的捕获，被声明的捕获变量类型会根据表达式进行判断。

泛型Lambda

C++14开始，Lambda函数的形式参数可以使用auto关键字来产生意义上的泛型（auto不能写在普通函数的参数，会与模板的功能产生冲突）：

```
auto add = [](auto x, auto y) {
    return x+y;
};

add(1, 2);
add(1.1, 2.2);
```

函数对象包装器

std::function

Lambda表达式的本质是一个和函数对象类型相似的类类型（称为闭包类型）的对象（称为闭包对象），当Lambda表达式的捕获列表为空时，闭包对象还能够转换为函数指针值进行传递：

```
#include <iostream>

using foo = void(int); // 定义函数类型
void functional(foo f) { // 参数列表中定义的函数类型 foo 被视为退化后的函数指针类型 foo*
    f(1); // 通过函数指针调用函数
}

int main() {
    auto f = [] (int value) {
        std::cout << value << std::endl;
    };
    functional(f); // 传递闭包对象，隐式转换为 foo* 类型的函数指针值
    f(1); // Lambda 表达式调用
    return 0;
}
```

上面的代码给出了两种不同的调用形式，一种是将Lambda作为函数类型传递进行调用，而另一种则是直接调用Lambda表达式，在C++11中，统一了这些概念，将能够被调用的对象的类型，统一称之为可调用类型。而这种类型，便是通过std::function引入的。C++11中的std::function是一种通用、多态的函数封装，它的实例可以对任何可以调用的目标实体进行存储、复制和调用操作，**它也是对C++中现有的**

可调用实体的一种类型安全的包裹（相对来说，函数指针的调用不是类型安全的），换句话说，就是函数的容器。

通过函数指针调用函数在某些情况下被认为不安全，主要原因包括：

- 类型安全性：函数指针可能会指向任何类型的函数，如果函数签名（参数类型和返回类型）不匹配，编译器可能无法捕获这种错误。这可能导致运行时错误，如访问违规或未定义行为。
- 内存安全性：如果函数指针指向的函数已经从内存中卸载（例如，在动态链接库被卸载后），再次调用该指针将导致程序崩溃。
- 代码可读性和维护性：使用函数指针可能会使代码难以阅读和维护，特别是在指针被频繁传递和修改的情况下。
- 安全漏洞：恶意代码可能会尝试修改函数指针的值，以指向恶意函数，从而执行任意代码。这是常见的安全攻击之一，如缓冲区溢出攻击。

```
#include <functional>
#include <iostream>

int foo(int para) {
    return para;
}

int main() {
    // std::function 包装了一个返回值为 int，参数为 int 的函数
    std::function<int(int)> func = foo;

    int important = 10;
    std::function<int(int)> func2 = [&](int value) -> int {
        return 1+value+important;
    };
    std::cout << func(10) << std::endl;
    std::cout << func2(10) << std::endl;
}
```

std::bind和std::placeholder

std::bind和std::placeholder可以将部分调用参数提前绑定到函数身上成为一个新的对象，然后在参数齐全后，完成调用。

```
int foo(int a, int b, int c) {
    ;
}

int main() {
    // 将参数1,2绑定到函数 foo 上,
    // 但使用 std::placeholders::_1 来对第一个参数进行占位
    auto bindFoo = std::bind(foo, std::placeholders::_1, 1, 2);
    // 这时调用 bindFoo 时，只需要提供第一个参数即可
    bindFoo(1);
}
```

右值引用

浅拷贝和深拷贝：

浅拷贝

浅拷贝仅复制对象的顶层结构，不复制对象内部引用的其他对象。如果原对象和副本中的某个成员是指向同一个对象的指针，那么修改这个内部对象的状态会影响到原对象和副本。

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

class ShallowCopyExample {
public:
    int* data;
    ShallowCopyExample(int d) {
        data = new int(d);
    }
    // 浅拷贝构造函数
    ShallowCopyExample(const ShallowCopyExample& other) {
        data = other.data;
    }
    ~ShallowCopyExample() {
        delete data;
    }
};
```

在这个例子中，复制ShallowCopyExample对象会导致新对象的data成员指向与原对象相同的内存地址。如果一个对象被销毁（调用析构函数），另一个对象的data指针就会悬挂，指向已释放的内存。

深拷贝

深拷贝不仅复制对象的顶层结构，还递归地复制对象内部引用的所有对象。这意味着原对象和副本是完全独立的；一个对象的修改不会影响另一个对象。

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

class DeepCopyExample {
public:
    int* data;
    DeepCopyExample(int d) {
        data = new int(d);
    }
    // 深拷贝构造函数
    DeepCopyExample(const DeepCopyExample& other) {
        data = new int(*other.data);
    }
    ~DeepCopyExample() {
        delete data;
    }
};
```

在这个例子中，复制DeepCopyExample对象时，会创建一个新的int对象，并将其地址赋给新对象的data成员。这样，即使原对象被销毁，副本对象也不会受到影响，因为它们指向不同的内存地址。

总结：

- 浅拷贝：复制对象的顶层结构，不复制内部引用的对象。原对象和副本可能共享内部状态。
- 深拷贝：复制对象的顶层结构及其引用的所有对象，确保原对象和副本完全独立。

左值、右值的纯右值、将亡值、右值

左值：表达式后仍然存在的持久对象

右值：表达式之后不再存在的临时对象

- 纯右值：纯粹的右值，要么是纯粹的字面量，例如10, true；要么是求值结果相当于字面量或匿名临时对象，例如 $1+2$ 。非引用返回的临时变量、运算表达式产生的临时变量、原始字面量、Lambda表达式都属于纯右值。

- 将亡值：即将被销毁、却能够被移动的值。

将亡值：

```
std::vector<int> foo() {
    std::vector<int> temp = {1, 2, 3, 4};
    return temp;
}
std::vector<int> v = foo();
```

PS：教科书说这样会引起拷贝开销，尤其是当temp特别大的时候（是我没看完，作者最后提了）。而GPT说C++11引入的返回值优化（Return Value Optimization, RVO）以及后来的拷贝省略（Copy Elision, C++17）规则，不会引起额外的拷贝开销。

下面三个知识点可以看看[一文读懂C++右值引用和std::move-腾讯技术工程的文章](#)

右值引用和左值引用

拿到一个将亡值，就需要用到右值引用：T &&，其中T是类型。右值引用的声明让这个临时值的生命周期得以延长、只要变量还活着，那么将亡值将继续存活。C++11提供了std::move将左值参数转换为右值（实际上右值的本质也是通过move来实现）：

```
int &&ref_r = 5

// 等价于下面
int tmp = 5
int &&ref_r = std::move(tmp)

const int &b = std::move(1); // 合法，常量左引用允许引用右值
```

需要注意的是ref_r虽然是右值引用，但是个左值。

移动语义

传统的C++没有区分『移动』和『拷贝』的概念，造成了大量的数据拷贝，浪费时间和空间。右值引用的出现恰好就解决了这两个概念的混淆问题：

```
#include <iostream>
class A {
public:
    int *pointer;
    A(): pointer(new int(1)) {
```

```

        std::cout << "构造" << pointer << std::endl;
    }
A(A& a):pointer(new int(*a.pointer)) {
    std::cout << "拷贝" << pointer << std::endl;
} // 无意义的对象拷贝
A(A&& a):pointer(a.pointer) {
    a.pointer = nullptr;
    std::cout << "移动" << pointer << std::endl;
}
~A() {
    std::cout << "析构" << pointer << std::endl;
    delete pointer;
}
};

// 防止编译器优化
A return_rvalue(bool test) {
    A a,b;
    if(test) return a; // 等价于 static_cast<A&&>(a);
    else return b;     // 等价于 static_cast<A&&>(b);
}
int main() {
    A obj = return_rvalue(false);
    std::cout << "obj:" << std::endl;
    std::cout << obj.pointer << std::endl;
    std::cout << *obj.pointer << std::endl;
    return 0;
}

```

a.pointer = nullptr;的目的是为了确保源对象a在资源被"移动"到新对象之后，不再拥有这块资源。这是必要的，因为当源对象a最终被销毁时，其析构函数会被调用，如果pointer没有被设置为nullptr，那么析构函数会尝试删除同一块内存两次（一次是在源对象a的析构函数中，另一次是在新对象的析构函数中），这会导致未定义行为，通常是程序崩溃。

标准库中移动语义的例子：

```

#include <iostream> // std::cout
#include <utility> // std::move
#include <vector> // std::vector
#include <string> // std::string

int main() {
    std::string str = "Hello world.";
    std::vector<std::string> v;

    // 将使用 push_back(const T&)，即产生拷贝行为
    v.push_back(str);
    // 将输出 "str: Hello world."
    std::cout << "str: " << str << std::endl;

    // 将使用 push_back(const T&&)，不会出现拷贝行为
    // 而整个字符串会被移动到 vector 中，所以有时候 std::move 会用来减少拷贝出现的开销
    // 这步操作后，str 中的值会变为空
    v.push_back(std::move(str));
    // 将输出 "str: "
    std::cout << "str: " << str << std::endl;
}

```

```
    return 0;
}
```

完美转发

一个声明的右值引用其实是一个左值。这就为我们进行参数转发（传递）造成了问题：

```
void reference(int& v) {
    std::cout << "左值" << std::endl;
}
void reference(int&& v) {
    std::cout << "右值" << std::endl;
}
template <typename T>
void pass(T&& v) {
    std::cout << "普通传参:" ;
    reference(v); // 始终调用 reference(int&)
}
int main() {
    std::cout << "传递右值:" << std::endl;
    pass(1); // 1是右值，但输出是左值

    std::cout << "传递左值:" << std::endl;
    int l = 1;
    pass(l); // l 是左值，输出左值

    return 0;
}
```

无论模板参数是什么类型的引用，当且仅当实参类型为右引用时，模板参数才能被推导为右引用类型。这是因为引用塌缩的出现。

函数形参类型	实参参数类型	推导后函数形参类型
T&	左引用	T&
T&	右引用	T&
T&&	左引用	T&
T&&	右引用	T&&

完美转发就是基于上述规律产生的。所谓完美转发，就是为了让我们在传递参数的时候，保持原来的参数类型（左引用保持左引用，右引用保持右引用）。为了解决这个问题，我们应该使用 `std::forward` 来进行参数的转发（传递）：

```
#include <iostream>
#include <utility>
```

```

void reference(int& v) {
    std::cout << "左值引用" << std::endl;
}
void reference(int&& v) {
    std::cout << "右值引用" << std::endl;
}
template <typename T>
void pass(T&& v) {
    std::cout << "普通传参: ";
    reference(v);
    std::cout << "      std::move 传参: ";
    reference(std::move(v));
    std::cout << "      std::forward 传参: ";
    reference(std::forward<T>(v));
    std::cout << " static_cast<T&&> 传参: ";
    reference(static_cast<T&&>(v));
}
int main() {
    std::cout << "传递右值:" << std::endl;
    pass(1);

    std::cout << "传递左值:" << std::endl;
    int v = 1;
    pass(v);

    return 0;
}

/*
传递右值:
    普通传参: 左值引用
    std::move 传参: 右值引用
    std::forward 传参: 右值引用
static_cast<T&&> 传参: 右值引用
传递左值:
    普通传参: 左值引用
    std::move 传参: 右值引用
    std::forward 传参: 左值引用
static_cast<T&&> 传参: 左值引用
*/

```

在使用循环语句的过程中，`auto&&`是最安全的方式：因为当`auto`被推导为不同的左右引用时，与`&&`的坍缩组合是完美转发。

在腾讯技术工程中看到的一些问题，遂记录：

- 空指针到底能不能访问? (`int *p = nullptr; p = 5;`)：不可以。尝试通过空指针 (`nullptr`) 访问或修改数据会导致未定义行为 (Undefined Behavior, UB)，通常会导致程序崩溃。例如，`int *p = nullptr; *p = 5;` 这段代码尝试通过空指针 `p` 写入值 5，这是不合法的操作。
- 给一个变量取地址，取到的是不是物理地址？(`int a; std::cout << &a;`)：取到的是虚拟地址，而非物理地址。在现代操作系统中，程序看到的地址是虚拟内存地址，操作系统和硬件协同工作，通过内存管理单元 (MMU) 将虚拟地址映射到物理地址上。因此，`std::cout << &a;` 输出的是变量 `a` 的虚拟地址。
- 操作一个常数地址是否合法？(`((int *)0xa0000 = 0x41;)`)：通常不合法且危险。尽管技术上可以通过强制类型转换将一个地址常量转换为指针并尝试写入 (如`*((int *)0xa0000) = 0x41;`)，但这种操作依赖于特定的硬件和操作系统环境，且很容易导致程序崩溃或者未定义行为。在受保护的操作系统中，随意访问某个物理地址是被禁止的。
- 全局变量、静态局部变量、字符串字面量等在内存中是如何布局的？
 - 全局变量：存储在程序的数据段 (data segment) 或BSS段 (未初始化的全局变量)。
 - 静态局部变量：存储在数据段，与全局变量类似，但只在声明它的函数内可见。
 - 字符串字面量：存储在只读数据段 (rodata segment)，尝试修改字符串字面量的内容会导致未定义行为。
- C/C++程序如何编译为内核代码，运行在内核态程序上？：C/C++代码通常编译为用户态程序。要运行在内核态，代码需要以内核模块的形式编写并加载到内核中，或者直接编写为操作系统的一部分。这涉及到使用特定于操作系统的API和遵守内核编程的规则 (如内存管理、同步机制等)。编译内核代码通常需要使用特定的编译器选项和链接到内核提供的服务。
- gdb过程中，看到的寄存器是否是真实的？：是真实的。在使用GDB调试程序时，看到的寄存器值反映的是当前CPU寄存器的状态或者特定时刻的快照。这些信息对于理解程序的执行流程和调试非常有用。

容器

线性容器

`std::array`

1.`std::array` 和 `std::vector`: `std::array` 对象的大小是固定的，如果容器大小是固定的，那么可以优先考虑使用 `std::array` 容器。另外由于 `std::vector` 是自动扩容的，当存入大量的数据后，并且对容器进行了删除操作，容器并不会自动归还被删除元素相应的内存，这时候就需要手动运行 `shrink_to_fit()` 释放这部分内存。

```
std::vector<int> v;
std::cout << "size:" << v.size() << std::endl;           // 输出 0
std::cout << "capacity:" << v.capacity() << std::endl; // 输出 0

// 如下可看出 std::vector 的存储是自动管理的，按需自动扩张
// 但是如果空间不足，需要重新分配更多内存，而重分配内存通常是性能上有开销的操作
v.push_back(1);
v.push_back(2);
v.push_back(3);
std::cout << "size:" << v.size() << std::endl;           // 输出 3
std::cout << "capacity:" << v.capacity() << std::endl; // 输出 4

// 这里的自动扩张逻辑与 Golang 的 slice 很像
```

```

v.push_back(4);
v.push_back(5);
std::cout << "size:" << v.size() << std::endl;           // 输出 5
std::cout << "capacity:" << v.capacity() << std::endl; // 输出 8

// 如下可看出容器虽然清空了元素，但是被清空元素的内存并没有归还
v.clear();
std::cout << "size:" << v.size() << std::endl;           // 输出 0
std::cout << "capacity:" << v.capacity() << std::endl; // 输出 8

// 额外内存可通过 shrink_to_fit() 调用返回给系统
v.shrink_to_fit();
std::cout << "size:" << v.size() << std::endl;           // 输出 0
std::cout << "capacity:" << v.capacity() << std::endl; // 输出 0

```

2.std::array和普通数组：std::array能够让代码变得更加“现代化”，而且封装了一些操作函数，比如获取数组大小以及检查是否非空，同时还能够友好的使用标准库中的容器算法，比如std::sort。

```

std::array<int, 4> arr = {1, 2, 3, 4};

arr.empty(); // 检查容器是否为空
arr.size(); // 返回容纳的元素数

// 迭代器支持
for (auto &i : arr)
{
    // ...
}

// 用 lambda 表达式排序
std::sort(arr.begin(), arr.end(), [] (int a, int b) {
    return b < a;
});

// 数组大小参数必须是常量表达式
constexpr int len = 4;
std::array<int, len> arr = {1, 2, 3, 4};

// 非法，不同于 C 风格数组，std::array 不会自动退化成 T*
// int *arr_p = arr;

```

如果需要兼容c风格的接口，可采用以下做法：

```

void foo(int *p, int len) {
    return;
}

std::array<int, 4> arr = {1, 2, 3, 4};

// C 风格接口传参
// foo(arr, arr.size()); // 非法，无法隐式转换
foo(&arr[0], arr.size());
foo(arr.data(), arr.size());

```

std::forward_list

和std::list的双向链表的实现不同，std::forward_list使用单向链表进行实现，提供了O(1)复杂度的元素插入，不支持快速随机访问（这也是链表的特点），也是标准库容器中唯一一个不提供size()方法的容器。当不需要双向迭代时，具有比std::list更高的空间利用率。

需要使用list的场景：需要频繁在序列的中间插入或删除元素时，list是一个很好的选择。例如，在实现一个撤销功能时，你可能需要在操作的历史记录中频繁地插入和删除记录。

无序容器

传统C++中的有序容器std::map/std::set，**这些元素内部通过红黑树进行实现，插入和搜索的平均复杂度均为O(log(size))（二分查找）**。在插入元素时候，会根据<操作符比较元素大小并判断元素是否相同，并选择合适的位置插入到容器中。当对这个容器中的元素进行遍历时，输出结果会按照<操作符的顺序来逐个遍历。

而无序容器中的元素是不进行排序的，内部通过Hash表实现，插入和搜索元素的平均复杂度为O(constant)，在不关心容器内部元素顺序时，能够获得显著的性能提升。

C++11引入了的两组无序容器分别是：std::unordered_map/std::unordered_multimap 和 std::unordered_set/std::unordered_multiset。它们的用法和原有的 std::map/std::multimap/std::set/std::multiset 基本类似。

元组

元组基本操作

关于元组的使用有三个核心的函数：

- std::make_tuple: 构造元组
- std::get: 获得元组某个位置的值
- std::tie: 元组拆包

```
#include <tuple>
#include <iostream>

auto get_student(int id)
{
    // 这个时候用auto就很方便，返回类型被推断为 std::tuple<double, char, std::string>

    if (id == 0)
        return std::make_tuple(3.8, 'A', "张三");
    if (id == 1)
        return std::make_tuple(2.9, 'C', "李四");
    if (id == 2)
        return std::make_tuple(1.7, 'D', "王五");
    return std::make_tuple(0.0, 'D', "null");
    // 如果只写 0 会出现推断错误，编译失败
}

int main()
{
    auto student = get_student(0);
    std::cout << "ID: 0, "
    << "GPA: " << std::get<0>(student) << ", "
    << std::get<1>(student) << ", "
    << std::get<2>(student);
}
```

```
<< "成绩: " << std::get<1>(student) << ", "
<< "姓名: " << std::get<2>(student) << '\n';

double gpa;
char grade;
std::string name;

// 元组进行拆包
std::tie(gpa, grade, name) = get_student(1);
std::cout << "ID: 1, "
<< "GPA: " << gpa << ", "
<< "成绩: " << grade << ", "
<< "姓名: " << name << '\n';
}
```

std::get除了使用常量获取元组对象外，C++14增加了使用类型来获取元组中的对象：

```
std::tuple<std::string, double, double, int> t("123", 4.5, 6.7, 8);
std::cout << std::get<std::string>(t) << std::endl;
```

ps：后面还有运行期索引和元组合并和遍历，感觉比较鸡肋，一般还是自己定义结构体然后返回

智能指针与内存管理

RAII与引用计数

引用计数这种计数是为了防止内存泄露而产生的。基本想法是对于动态分配的对象，进行引用计数，每当增加一次对同一个对象的引用，那么引用对象的引用计数就会增加一次，每删除一次引用，引用计数就会减一，当一个对象的引用计数减为零时，就自动删除指向的堆内存。

在传统 C++ 中，『记得』手动释放资源，总不是最佳实践。因为我们很有可能就忘记了去释放资源而导致泄露。所以通常的做法是对于一个对象而言，我们在构造函数的时候申请空间，而在析构函数（在离开作用域时调用）的时候释放空间，也就是我们常说的RAII资源获取即初始化技术。

凡事都有例外，我们总会有需要将对象在自由存储上分配的需求，在传统C++里我们只好使用new和delete去『记得』对资源进行释放。而C++11引入了智能指针的概念，使用了引用计数的想法，让程序员不再需要关心手动释放内存。这些智能指针包括std::shared_ptr/std::unique_ptr/std::weak_ptr，使用它们需要包含头文件。

引用计数和垃圾回收的大概区别

引用计数：引用计数是一种内存管理技术，通过在对象上维护一个引用计数器来跟踪对象的引用数量。当引用计数为0时，表示没有任何指针指向该对象，可以安全地释放对象占用的内存。

优点：

- 立即回收：对象在成为垃圾时会立即被回收，不需要等待垃圾回收器的运行。简单高效：引用计数的实现相对简单，不需要进行停顿性的垃圾回收操作。

缺点：

- 循环引用问题：引用计数难以处理循环引用，即两个对象互相引用导致引用计数永远不会为0，从而导致内存泄漏。
- 开销大：每次引用计数发生变化时都需要更新计数器，可能会带来额外的性能开销（更新计数器本身其实是很小的开销，但是所有的引用传递都去更新，累积起来也很大）。

垃圾回收：垃圾回收是一种自动内存管理技术，通过检测和回收不再被程序使用的内存来避免内存泄漏。常见的垃圾回收算法包括标记-清除、引用计数、复制算法等。

优点：

- 处理循环引用：垃圾回收器可以处理循环引用，通过识别不可达对象并将其回收来解决内存泄漏问题。
- 自动化：程序员无需手动管理内存，减少了出错的可能性。

缺点：

- 停顿：某些垃圾回收算法可能导致程序在进行垃圾回收时出现停顿，影响程序性能。
- 实现复杂：一些高效的垃圾回收算法实现较为复杂，可能会消耗额外的计算资源。

std::shared_ptr

std::shared_ptr能够记录多少个shared_ptr共同指向一个对象，从而消除显式的调用delete，当引用计数变为零的时候就会将对象自动删除。因为使用 std::shared_ptr 仍然需要使用new来调用，这使得代码出现了某种程度上的不对称。std::make_shared就能够用来消除显式的使用new，所以 std::make_shared会分配创建传入参数中的对象，并返回这个对象类型的std::shared_ptr指针。

std::shared_ptr可以通过get()方法来获取原始指针，通过reset()来减少一个引用计数，并通过use_count()来查看一个对象的引用计数。

```
#include <iostream>
#include <memory>
void foo(std::shared_ptr<int> i) {
    (*i)++;
}
int main() {
    // auto pointer = new int(10); // illegal, no direct assignment
    // Constructed a std::shared_ptr
    auto pointer = std::make_shared<int>(10);
    foo(pointer);
    std::cout << *pointer << std::endl; // 11
    // The shared_ptr will be destructed before leaving the scope
    return 0;
}
auto pointer = std::make_shared<int>(10);
auto pointer2 = pointer; // 引用计数+1
auto pointer3 = pointer; // 引用计数+1
int *p = pointer.get(); // 这样不会增加引用计数
std::cout << "pointer.use_count() = " << pointer.use_count() << std::endl; // 3
std::cout << "pointer2.use_count() = " << pointer2.use_count() << std::endl; // 3
std::cout << "pointer3.use_count() = " << pointer3.use_count() << std::endl; // 3

pointer2.reset();
std::cout << "reset pointer2:" << std::endl;
std::cout << "pointer.use_count() = " << pointer.use_count() << std::endl; // 2
std::cout << "pointer2.use_count() = "
    << pointer2.use_count() << std::endl; // pointer2 已 reset; 0
std::cout << "pointer3.use_count() = " << pointer3.use_count() << std::endl; // 2
pointer3.reset();
std::cout << "reset pointer3:" << std::endl;
std::cout << "pointer.use_count() = " << pointer.use_count() << std::endl; // 1
std::cout << "pointer2.use_count() = " << pointer2.use_count() << std::endl; // 0
std::cout << "pointer3.use_count() = " << pointer3.use_count() << std::endl; // pointer3 已 reset; 0
```

std::unique_ptr

std::unique_ptr是一种独占的智能指针，它禁止其他智能指针与其共享同一个对象，从而保证代码的安全。虽然不可复制，但是可以使用std::move（可以将左值转换为右值）转移给其他的std::unique_ptr。

```
std::unique_ptr<int> pointer = std::make_unique<int>(10); // make_unique 从 C++14
引入, C++11没有是单纯被忘记了:(  
std::unique_ptr<int> pointer2 = pointer; // 非法
```

std::weak_ptr

前面提到引用计数无法处理循环引用，从而导致资源的引用永远不会变为0，从而导致内存泄漏。

```
struct A;  
struct B;
```

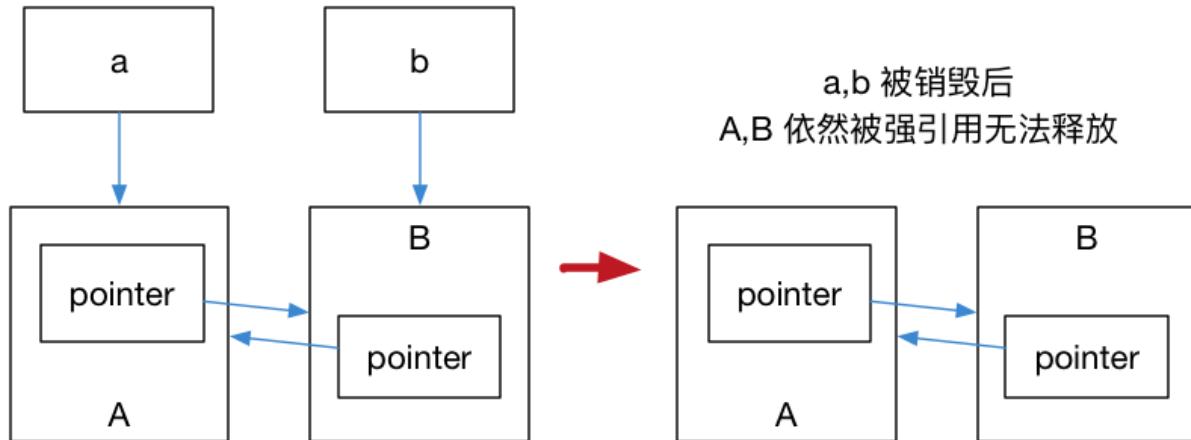
```

struct A {
    std::shared_ptr<B> pointer;
    ~A() {
        std::cout << "A 被销毁" << std::endl;
    }
};

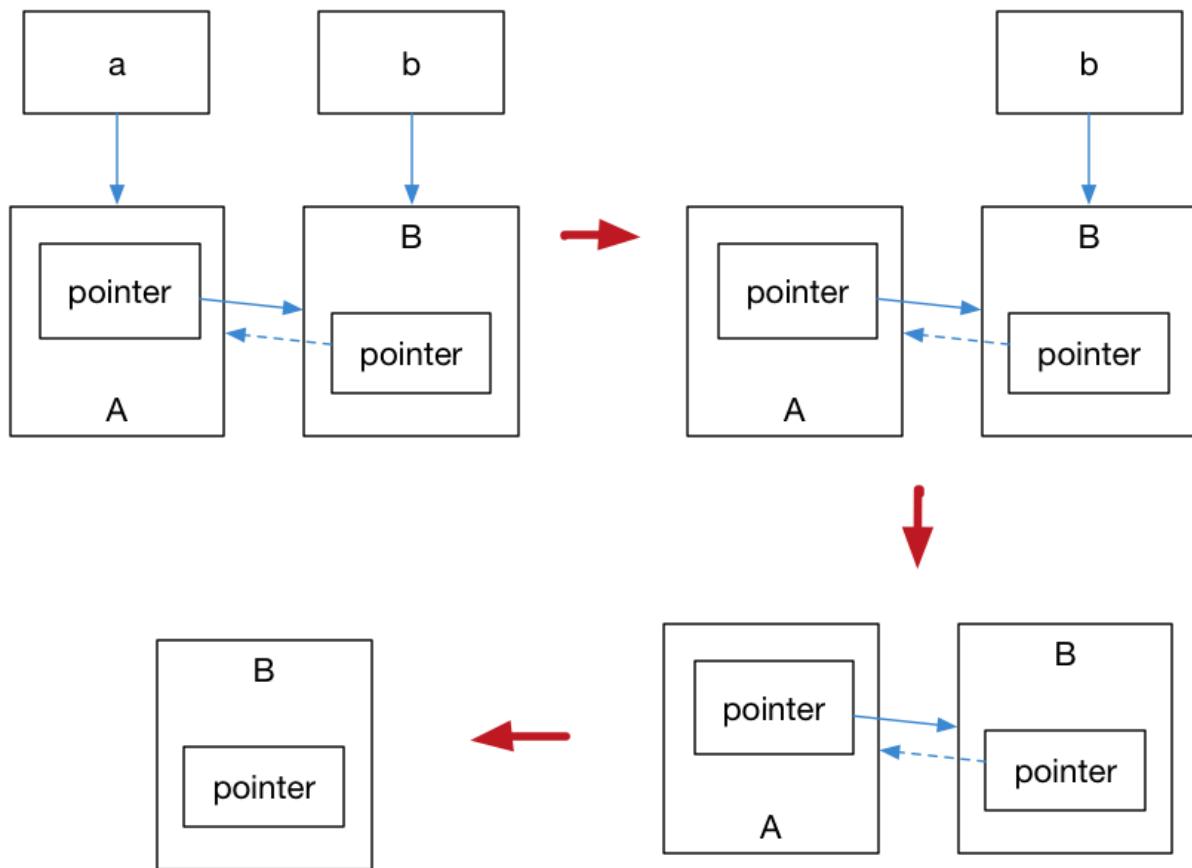
struct B {
    std::shared_ptr<A> pointer;
    ~B() {
        std::cout << "B 被销毁" << std::endl;
    }
};

int main() {
    auto a = std::make_shared<A>();
    auto b = std::make_shared<B>();
    a->pointer = b;
    b->pointer = a;
}

```



`std::weak_ptr`, `std::weak_ptr` 是一种弱引用 (相比较而言 `std::shared_ptr` 就是一种强引用) , 弱引用不会引起引用计数增加。



上图中，由于`b->pointer`为弱引用，因此`a->pointer`只有`a`一个引用，当`a`销毁后，`a->pointer`也随之销毁，`b`被销毁后，`b->pointer`没有引用，内存资源被释放。

`std::weak_ptr`没有 * 运算符和 > 运算符，所以不能够对资源进行操作，它主要用于检查`std::shared_ptr`是否存在，其`expired()`方法能在资源未被释放时，会返回false，否则返回true；除此之外，它也可以用于获取指向原始对象的`std::shared_ptr`指针，其`lock()`方法在原始对象未被释放时，返回一个指向原始对象的`std::shared_ptr`指针，进而访问原始对象的资源，否则返回`nullptr`。

正则表达式

一般使用正则表达式主要是实现下面三个需求：

- 检查一个串是否包含某种形式的子串；
- 将匹配的子串替换；
- 从某个串中取出符合条件的子串。

正则表达式是由普通字符（例如a到z）以及特殊字符组成文字模式。模式描述在搜索文本时要匹配的一个或多个字符串。正则表达式作为一个模板，将某个字符模式与所搜索的字符串进行匹配。

特殊字符

特殊字符是正则表达式里有特殊含义的字符，也是正则表达式的核心匹配语法。参见下表：

特别字符	描述
\$	匹配输入字符串的结尾位置。
(,)	标记一个子表达式的开始和结束位置。子表达式可以获取供以后使用。
*	匹配前面的子表达式零次或多次。
+	匹配前面的子表达式一次或多次。
.	匹配除换行符 \n 之外的任何单字符。
[标记一个中括号表达式的开始。
?	匹配前面的子表达式零次或一次，或指明一个非贪婪限定符。
\	将下一个字符标记为或特殊字符、或原义字符、或向后引用、或八进制转义符。例如，\n 匹配字符 n。\\n 匹配换行符。序列 \\ 匹配 '\\' 字符，而 \\ 则匹配 '\"' 字符。
^	匹配输入字符串的开始位置，除非在方括号表达式中使用，此时它表示不接受该字符集合。
{	标记限定符表达式的开始。
	指明两项之间的一个选择。

限定符

限定符用来指定正则表达式的一个给定的组件必须要出现多少次才能满足匹配。见下表：

字符	描述
*	匹配前面的子表达式零次或多次。例如，foo* 能匹配 fo 以及 foooo。* 等价于 {0,}。
+	匹配前面的子表达式一次或多次。例如，foo+ 能匹配 foo 以及 foooo，但不能匹配 fo。+ 等价于 {1,}。

字符	描述
?	匹配前面的子表达式零次或一次。例如， <code>Your(s)?</code> 可以匹配 <code>Your</code> 或 <code>Yours</code> 中的 <code>Your</code> 。 <code>?</code> 等价于 <code>{0,1}</code> 。
{n}	<code>n</code> 是一个非负整数。匹配确定的 <code>n</code> 次。例如， <code>o{2}</code> 不能匹配 <code>for</code> 中的 <code>o</code> ，但是能匹配 <code>foo</code> 中的两个 <code>o</code> 。
{n,}	<code>n</code> 是一个非负整数。至少匹配 <code>n</code> 次。例如， <code>o{2,}</code> 不能匹配 <code>for</code> 中的 <code>o</code> ，但能匹配 <code>fooooooo</code> 中的所有 <code>o</code> 。 <code>o{1,}</code> 等价于 <code>o+</code> 。 <code>o{0,}</code> 则等价于 <code>o*</code> 。
{n,m}	<code>m</code> 和 <code>n</code> 均为非负整数，其中 <code>n</code> 小于等于 <code>m</code> 。最少匹配 <code>n</code> 次且最多匹配 <code>m</code> 次。例如， <code>o{1,3}</code> 将匹配 <code>foooooo</code> 中的前三个 <code>o</code> 。 <code>o{0,1}</code> 等价于 <code>o?</code> 。注意，在逗号和两个数之间不能有空格。

std::regex及其相关

C++作为一门高性能语言，在后台服务的开发中，对URL资源链接进行判断时，使用正则表达式也是工业界最为成熟的普遍做法。一般的解决方案就是使用boost的正则表达式库。而C++11正式将正则表达式的处理方法纳入标准库的行列，从语言级上提供了标准的支持，不再依赖第三方。C++11提供的正则表达式库操作std::string对象，模式std::regex（本质是 std::basic_regex）进行初始化，通过std::regex_match进行匹配，从而产生std::smatch（本质是std::match_results对象）。

例如：[a-z]+\.\txt: 在这个正则表达式中，[a-z] 表示匹配一个小写字母，+ 可以使前面的表达式匹配多次，因此 [a-z]+ 能够匹配一个小写字母组成的字符串。在正则表达式中一个 . 表示匹配任意字符，而 \. 则表示匹配字符 .，最后的 txt 表示严格匹配 txt 则三个字母。因此这个正则表达式的所要匹配的内容是由纯小写字母组成的文本文件。

std::regex_match用于匹配字符串和正则表达式，有很多不同的重载形式。最简单的一个形式就是传入 std::string 以及一个 std::regex 进行匹配，当匹配成功时，会返回 true，否则返回 false。

另一种常用的形式就是依次传入 std::string/std::smatch/std::regex 三个参数，其中 std::smatch 的本质其实是 std::match_results。故而在标准库的实现中，std::smatch 被定义为了 std::match_results std::string::const_iterator，也就是一个子串迭代器类型的 match_results。使用 std::smatch 可以方便的对匹配的结果进行获取。

```
#include <iostream>
#include <string>
#include <regex>

int main() {
    std::string fnames[] = {"foo.txt", "bar.txt", "test", "a0.txt", "AAA.txt"};
    // 在 C++ 中 \ 会被作为字符串内的转义符,
    // 为使 \. 作为正则表达式传递进去生效，需要对 \ 进行二次转义，从而有 \\.
    std::regex txt_regex("[a-z]+\.\txt");
    for (const auto &fname: fnames)
        std::cout << fname << ":" << std::regex_match(fname, txt_regex) <<
    std::endl;
}

std::regex base_regex("([a-z]+)\.\txt");
std::smatch base_match;
for(const auto &fname: fnames) {
    if (std::regex_match(fname, base_match, base_regex)) {
```

```
// std::smatch 的第一个元素匹配整个字符串
// std::smatch 的第二个元素匹配了第一个括号表达式
if (base_match.size() == 2) {
    std::string base = base_match[1].str();
    std::cout << "sub-match[0]: " << base_match[0].str() << std::endl;
    std::cout << fname << " sub-match[1]: " << base << std::endl;
}
}
```

进程和线程

进程是操作系统分配资源的基本单位。每个进程都有自己独立的地址空间、内存、数据栈以及其他记录其运行轨迹的辅助数据。进程之间相互独立，一个进程无法直接访问另一个进程的资源和数据，因此进程是资源隔离和保护的基本边界。不同进程间的通信（IPC，Inter-Process Communication）需要通过操作系统提供的机制进行，如管道、消息队列、共享内存等。在C++中创建新进程通常依赖于操作系统提供的API。在Unix/Linux系统中，可以使用fork()系统调用来创建新进程。

线程是进程内的执行单元，一个进程可以包含一个或多个线程。线程是操作系统调度的实体和基本单位，操作系统的调度器（Scheduler）会在可运行的线程之间进行切换，决定哪个线程将获得CPU时间进行执行。线程不拥有资源，只拥有一点在运行中必不可少的资源（如执行栈、寄存器状态等），但能访问隶属进程的资源。同一进程内的线程共享该进程的地址空间和资源，这使得线程间的通信和数据共享变得更加容易，但也需要注意同步和数据一致性的问题。在C++中，创建和管理线程可以通过C++11引入的线程库来实现。

主要区别：

- 资源独立性：进程拥有独立的资源，而线程共享进程资源。
- 通信方式：进程间通信需要特殊的IPC机制，线程间通信更加方便，因为它们自然地共享了进程的资源。
- 开销：创建和销毁进程的开销远大于线程，同理，进程间的切换开销也大于线程间的切换。
- 数据共享、同步：由于线程共享进程资源，因此线程间的数据共享和同步非常重要，以避免数据不一致等问题。

并行与并发

并行和并发的概念很容易混淆，都是用于描述系统如何处理多个任务的方式。

- [之前写的博客：Concurrency and Parallelism](#)
- [Talk: Rob Pike: Concurrency and Parallelism](#)

并发是指系统能够处理多个任务的能力。在并发模型中，一个处理器在同一时间点或时间段内处理多个任务。这不意味着这些任务实际上是在同一时刻执行的；而是通过任务之间的快速切换，给用户一种多个任务同时进行的错觉。**并发更多地关注的是结构上的分离，即如何组织程序或系统以同时处理多个任务。**并发的一个常见场景是在单核CPU上运行的多任务操作系统。尽管CPU在任意时刻只能执行一个任务，但操作系统通过在不同任务之间迅速切换，使得多个程序似乎是“同时”运行。在C++中可以通过std::thread来实现并发（不使用则程序主要运行在单个线程上，即主线程，除非使用的某些外部库内部创建了线程）。

并行是指多个处理器或多核处理器同时执行多个任务或同一个任务的不同部分的能力。并行计算的目的是通过同时使用多个计算资源来加速处理过程。**并行更多地关注的是性能上的提升，即如何利用多个处理器或多核心同时执行任务以提高效率和速度。**一个典型的并行计算例子是在多核CPU上运行的程序，其中每个核心被分配了不同的任务或同一个任务的不同部分，它们真正意义上是在同一时刻并行执行的。在C++中可以通过std::thread和std::async来实现并行。

可以说并行是并发的一个子集或特例。

并行基础

std::thread用于创建一个执行的线程实例，使用时需要包含头文件，它提供了很多基本的线程操作，例如get_id()来获取所创建线程的线程ID，使用join()来等待一个线程结束（与该线程汇合）等等：

```
#include <iostream>
#include <thread>

int main() {
    std::thread t([](){
        std::cout << "hello world." << std::endl;
    });
    t.join();
    return 0;
}
```

互斥量与临界区

互斥量与临界区都是用于同步多个线程对共享资源访问的机制，避免发生竞态条件和保证数据的一致性。

互斥量是一种同步原语，用于保护共享资源，确保在任何时刻只有一个线程可以访问该资源。互斥量可以是跨进程的（即可以在不同进程的线程之间同步）。适用于需要跨进程同步或者需要更细粒度控制的场景。例如，C++中的std::mutex就提供了一个跨线程的互斥锁实现。互斥量提供了锁定（lock）和解锁（unlock）操作。当一个线程锁定了互斥量后，其他试图锁定该互斥量的线程将会阻塞，直到互斥量被解锁。可以跨不同的代码区域使用。

临界区通常指代代码中访问共享资源的部分，用于控制同一进程中多个线程对共享资源的访问。主要用于同一进程内的线程同步，由于临界区对象通常存储在用户进程的内存空间中，因此它们不能用于进程间同步。临界区相对于互斥量有更轻量级的开销，因为它们是为了同一进程内的线程同步而设计的。

std::mutex是C++11中最基本的mutex类，通过实例化std::mutex可以创建互斥量，而通过其成员函数lock()可以进行上锁，unlock()可以进行解锁。但是在实际编写代码的过程中，最好不去直接调用成员函数，因为调用成员函数就需要在每个临界区的出口处调用unlock()，当然，还包括异常。这时候C++11还为互斥量提供了一个RAII语法的模板类std::lock_guard。RAII在不失代码简洁性的同时，很好的保证了代码的异常安全性。在RAII用法下，对于临界区的互斥量的创建只需要在作用域的开始部分。

```
#include <iostream>
#include <mutex>
#include <thread>

int v = 1;

void critical_section(int change_v) {
    static std::mutex mtx;
    std::lock_guard<std::mutex> lock(mtx);

    // 执行竞争操作
    v = change_v;

    // 离开此作用域后 mtx 会被释放
}

int main() {
    std::thread t1(critical_section, 2), t2(critical_section, 3);
    t1.join();
    t2.join();

    std::cout << v << std::endl;
```

```
    return 0;  
}
```

由于C++保证了所有栈对象在生命周期结束时会被销毁，所以上述代码也是异常安全的。无论critical_section()正常返回、还是在中途抛出异常，都会引发堆栈回退，也就自动调用了unlock()。

而std::unique_lock则是相对于 std::lock_guard 出现的，std::unique_lock更加灵活，std::unique_lock的对象会以独占所有权（没有其他的unique_lock对象同时拥有某个mutex对象的所有权）的方式管理mutex对象上的上锁和解锁的操作。所以在并发编程中，推荐使用std::unique_lock。**std::lock_guard不能显式的调用lock和unlock，而std::unique_lock可以在声明后的任意位置调用，可以缩小锁的作用范围，提供更高的并发度**。如果用到了条件变量std::condition_variable::wait则必须使用std::unique_lock作为参数。

```
#include <iostream>  
#include <mutex>  
#include <thread>  
  
int v = 1;  
  
void critical_section(int change_v) {  
    static std::mutex mtx;  
    std::unique_lock<std::mutex> lock(mtx);  
    // 执行竞争操作  
    v = change_v;  
    std::cout << v << std::endl;  
    // 将锁进行释放  
    lock.unlock();  
  
    // 在此期间，任何人都可以抢夺 v 的持有权  
  
    // 开始另一组竞争操作，再次加锁  
    lock.lock();  
    v += 1;  
    std::cout << v << std::endl;  
}  
  
int main() {  
    std::thread t1(critical_section, 2), t2(critical_section, 3);  
    t1.join();  
    t2.join();  
    return 0;  
}
```

期物

期物（Future）表现为std::future，它提供了一个访问异步操作结果的途径，这句话很不好理解。试想当主线程A希望新开辟一个线程B去执行某个我们预期的任务，并返回我一个结果。而这时候，线程A可能正在忙其他的事情，无暇顾及B的结果，所以我们会很自然的希望能够在某个特定的时间获得线程B的结果。在C++11的std::future被引入之前，通常的做法是：创建一个线程A，在线程A里启动任务B，当准备完毕后发送一个事件，并将结果保存在全局变量中。而主函数线程A里正在做其他的事情，当需要结果的时候，调用一个线程等待函数来获得执行的结果。例如下面的例子，条件变量后面会介绍。

```
#include <iostream>
```

```

#include <thread>
#include <mutex>
#include <condition_variable>

// 全局变量用于保存任务结果
int result;
// 互斥锁和条件变量用于同步
std::mutex mtx;
std::condition_variable cv;
bool ready = false; // 用于指示任务是否完成

// 任务函数
void square(int x) {
    std::this_thread::sleep_for(std::chrono::seconds(1)); // 模拟耗时操作
    int temp_result = x * x;

    // 保存结果到全局变量，并通知等待的线程
    {
        std::lock_guard<std::mutex> lk(mtx);
        result = temp_result;
        ready = true;
    }
    cv.notify_one();
}

int main() {
    // 创建并启动线程
    std::thread t(square, 4);

    // 在主线程中等待任务完成
    {
        std::unique_lock<std::mutex> lk(mtx);
        cv.wait(lk, []{return ready;});
    }

    // 使用结果
    std::cout << "Result is: " << result << std::endl;

    // 确保线程结束
    t.join();
    return 0;
}

```

而C++11提供的std::future简化了这个流程，可以用来获取异步任务的结果。自然地，我们很容易能够想象到把它作为一种简单的线程同步手段，即屏障（barrier）。为了看一个例子，这里额外使用std::packaged_task，它可以用来封装任何可以调用的目标，从而用于实现异步的调用。

```

#include <iostream>
#include <future>
#include <thread>
#include <chrono>

// 定义一个简单的计算函数
int square(int x) {
    // 假设这个计算需要一些时间

```

```

    std::this_thread::sleep_for(std::chrono::seconds(1));
    return x * x;
}

int main() {
    // 使用std::packaged_task封装square函数, int(int)是square的签名
    std::packaged_task<int(int)> task(square);
    // 获取与packaged_task相关联的future
    std::future<int> result = task.get_future();

    // 在一个新线程上执行任务
    std::thread t(std::move(task), 4); // 计算4的平方

    // 主线程可以继续做其他事情...
    std::cout << "主线程正在做其他事情..." << std::endl;
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::cout << "主线程等待结果..." << std::endl;

    // 等待异步任务的结果
    int value = result.get(); // 这里会阻塞, 直到异步任务完成并返回结果

    std::cout << "计算结果: " << value << std::endl;

    // 确保线程执行完成
    t.join();

    return 0;
}

```

std::packaged_task对象是不可复制的，但可以被移动。这意味着你不能通过简单地复制一个std::packaged_task对象来将其从一个作用域传递到另一个作用域（例如，从主线程传递到子线程）。这是因为复制一个std::packaged_task对象会引起所有权和状态的不明确，特别是它关联的std::future对象的状态。**使用std::move可以有效地将task对象的所有权从主线程转移到新创建的线程t中。在std::move后，原始的task对象在主线程中变为处于“已移动”状态，这意味着它不再持有之前封装的任务或任何资源。而在新线程中，移动后的task对象则完整地拥有了原始任务和资源的所有权，可以安全地执行封装的任务。**

条件变量

互斥锁 (std::mutex) 是实现线程同步的基本工具之一，它可以保护共享数据，防止多个线程同时访问，从而避免数据竞争和不一致性。然而，仅仅使用互斥锁并不能解决所有的线程同步问题。特别是，当某些操作需要在特定条件满足时才能进行时，仅使用互斥锁可能会导致问题。例如，如果一个线程需要等待某个条件变为真才能继续执行，它可能会进入一个忙等待 (busy-wait) 状态，不断检查条件是否满足。这种忙等待不仅浪费CPU资源，两个或多个进程或线程在等待对方持有的资源，从而导致它们之间的循环等待，就会导致死锁。更具体地说，如果线程A持有互斥锁并且在等待某个条件（这个条件需要线程B的操作才能成立），但是线程B为了执行使条件成立的操作，也需要获取之前被线程A持有的互斥锁，这时就发生了死锁。线程A等待条件成立才释放锁，而线程B需要锁才能使条件成立。

为了解决这个问题，C++标准库提供了条件变量 (std::condition_variable)。条件变量允许一个或多个线程在某个条件变为真之前挂起（等待），而不占用任何CPU资源。当条件可能已经满足时，其他线程可以通过条件变量通知等待的线程重新检查条件，并根据需要继续执行。

std::condition_variable主要通过以下三个成员函数实现其功能：

- `wait(lock, predicate)`: 该函数使调用线程等待（挂起），直到被通知（通过`notify_one`或`notify_all`），同时释放传入的lock（通常是基于`std::unique_lock<std::mutex>`的锁）。在重新获得锁后，`wait`函数会再次检查`predicate`（一个返回布尔值的函数或lambda表达式），如果`predicate`结果为`false`，线程会再次等待。这防止了虚假唤醒（spurious wakeups）。
- `notify_one()`: 唤醒一个等待（挂起）在这个条件变量上的线程。如果有多个线程在等待，选择哪个线程被唤醒是不确定的。
- `notify_all()`: 唤醒所有等待（挂起）在这个条件变量上的线程。

```
#include <queue>
#include <chrono>
#include <mutex>
#include <thread>
#include <iostream>
#include <condition_variable>

int main() {
    std::queue<int> produced_nums;
    std::mutex mtx;
    std::condition_variable cv;
    bool notified = false; // 通知信号

    // 生产者
    auto producer = [&]() {
        for (int i = 0; ; i++) {
            std::this_thread::sleep_for(std::chrono::milliseconds(900));
            std::unique_lock<std::mutex> lock(mtx);
            std::cout << "producing " << i << std::endl;
            produced_nums.push(i);
            notified = true;
            cv.notify_all(); // 此处也可以使用 notify_one
        }
    };
    // 消费者
    auto consumer = [&]() {
        while (true) {
            std::unique_lock<std::mutex> lock(mtx);
            while (!notified) { // 避免虚假唤醒
                cv.wait(lock);
            }
            // 短暂取消锁，使得生产者有机会在消费者消费空前继续生产
            lock.unlock();
            // 消费者慢于生产者
            std::this_thread::sleep_for(std::chrono::milliseconds(1000));
            lock.lock();
            while (!produced_nums.empty()) {
                std::cout << "consuming " << produced_nums.front() << std::endl;
                produced_nums.pop();
            }
            notified = false;
        }
    };
}

// 分别在不同的线程中运行
```

```

    std::thread p(producer);
    std::thread cs[2];
    for (int i = 0; i < 2; ++i) {
        cs[i] = std::thread(consumer);
    }
    p.join();
    for (int i = 0; i < 2; ++i) {
        cs[i].join();
    }
    return 0;
}

```

上述代码存在一些潜在的问题：在消费者线程中，lock.unlock()和随后的lock.lock()之间存在一个时间窗口，其他线程（例如生产者线程）可能会在这个时间窗口内修改produced_nums和notified，这可能导致一些不一致的行为。

一部分输出结果如下：

```

producing 0
producing 1
consuming 0
consuming 1
producing 2
producing 3
consuming 2
consuming 3

```

生产者线程的工作是无限循环生成整数，并将其放入队列中。对于每个生成的整数，生产者都会：

1. 暂停900毫秒，模拟生产过程。
2. 获取互斥锁mtx。
3. 打印生产的整数。
4. 将整数推入队列produced_nums。
5. 设置notified为true，表示有数据可供消费。
6. 通过调用cv.notify_all()通知等待的消费者线程（在这个场景中，使用notify_one()也可以，但notify_all()确保所有消费者线程都被唤醒，这在多消费者场景下更有用），如果我们打印出当前消费者的编号，你会发现是随机的。

消费者线程的工作也是无限循环，但它等待生产者生成数据后才开始消费。对于每次消费操作，消费者都会：

1. 获取互斥锁mtx。
2. 检查notified是否为false，如果是，则等待条件变量cv的通知。这里使用while循环是为了防止虚假唤醒（即线程被唤醒但条件并未真正满足）。
3. 暂时释放互斥锁mtx，允许生产者有机会生产更多数据。
4. 暂停1000毫秒，模拟消费过程，这里消费者故意设置为比生产者慢，以便观察生产和消费的过程。
5. 再次获取互斥锁mtx。
6. 消费队列中的所有数据，打印每个被消费的整数。
7. 设置notified为false，表示所有数据已被消费完毕。

原子操作和内存模型

上节中的生产者消费者模型的例子可能存在编译器优化导致程序出错的情况。例如，布尔值notified没有被volatile修饰，编译器可能对此变量存在优化，例如将其作为一个寄存器的值，从而导致消费者线程永远无法观察到此值的变化。这是一个好问题，为了解释清楚这个问题，我们需要进一步讨论从C++ 11起引入的内存模型这一概念。

volatile：用于告诉编译器，一个变量的值可能会在没有明显的程序代码修改的情况下发生改变。这通常用于访问硬件设备、中断处理、多线程应用中共享变量的读写等场景。**当一个变量被声明为volatile时，编译器会避免对这个变量的读写操作进行优化，确保每次访问都直接从内存中读取数据，而不是从缓存或寄存器中。**

```
#include <thread>
#include <iostream>

int main() {
    int a = 0;
    int flag = 0;

    std::thread t1([&] () {
        while (flag != 1);

        int b = a;
        std::cout << "b = " << b << std::endl;
    });

    std::thread t2([&] () {
        a = 5;
        flag = 1;
    });

    t1.join();
    t2.join();
    return 0;
}
```

从直观上看，t2中a = 5; 这一条语句似乎总在flag = 1; 之前得到执行，而t1中while (flag != 1) 似乎保证了std::cout << "b = " << b << std::endl; 不会再标记被改变前执行。从逻辑上看，似乎b的值应该等于5。但实际情况远比此复杂得多，或者说这段代码本身属于未定义的行为，因为对于a和flag而言，他们在两个并行的线程中被读写，出现了竞争。除此之外，即便我们忽略竞争读写，仍然可能受CPU的乱序执行，编译器对指令的重排的影响，导致a = 5发生在flag = 1之后。从而b可能输出0。

笔者跑了几遍，都是b=5，总体而言这个示例还是太简单太弱了，但理论上确实是可能的

原子操作

为了使上述代码能够正确且可靠地在多线程环境下工作，可以采用以下任一解决方案：

- 使用std::atomic来声明a和flag变量，确保它们的读写操作是原子的，从而避免数据竞争和确保内存可见性。
- 使用互斥锁（std::mutex）和条件变量（std::condition_variable）来同步线程之间的操作。

std::mutex可以解决上面出现的并发读写的问题，但互斥锁是操作系统级的功能，这是因为一个互斥锁的实现通常包含两条基本原理：1.提供线程间自动的状态转换，即『锁住』这个状态；2.保障在互斥锁操作期间，所操作变量的内存与临界区外进行隔离。这是一组非常强的同步条件，换句话说当最终编译为CPU指令时会表现为非常多的指令。这对于一个仅需原子级操作（没有中间态）的变量，似乎太苛刻了。现代CPU体系结构提供了CPU指令级的原子操作，因此在C++11中多线程下共享变量的读写这一问题上，还引入了std::atomic模板，使得我们实例化一个原子类型，将一个原子类型读写操作从一组指令，最小化到单个CPU指令。

```
std::atomic<int> counter;
```

并为整数或浮点数的原子类型提供了基本的数值成员函数，举例来说，包括fetch_add, fetch_sub等，同时通过重载方便的提供了对应的+，-版本。

```
#include <atomic>
#include <thread>
#include <iostream>

std::atomic<int> count = {0};

int main() {
    std::thread t1([](){
        count.fetch_add(1);
    });
    std::thread t2([](){
        count++;           // 等价于 fetch_add
        count += 1;        // 等价于 fetch_add
    });
    t1.join();
    t2.join();
    std::cout << count << std::endl;
    return 0;
}
```

当然，并非所有的类型都能提供原子操作，这是因为原子操作的可行性取决于具体的CPU架构，以及所实例化的类型结构是否能够满足该CPU架构对内存对齐条件的要求，因而我们总是可以通过std::atomic::is_lock_free来检查该原子类型是否支持原子操作

```
#include <atomic>
#include <iostream>

struct A {
    float x;
    int y;
    long long z;
};

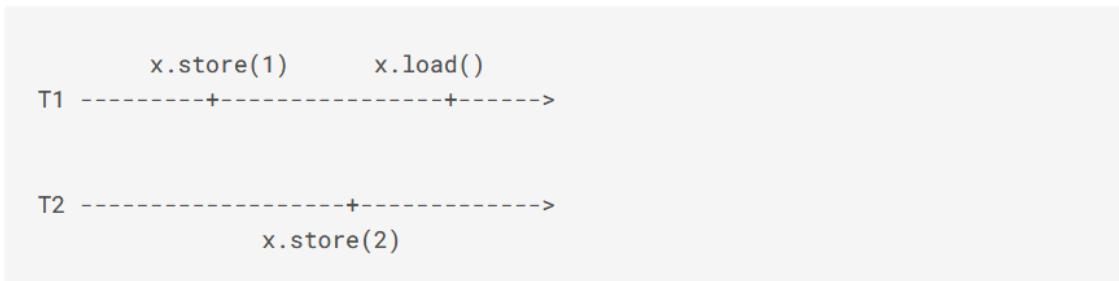
int main() {
    std::atomic<A> a;
    std::cout << std::boolalpha << a.is_lock_free() << std::endl;
    return 0;
}
```

报编译错误了

一致性模型

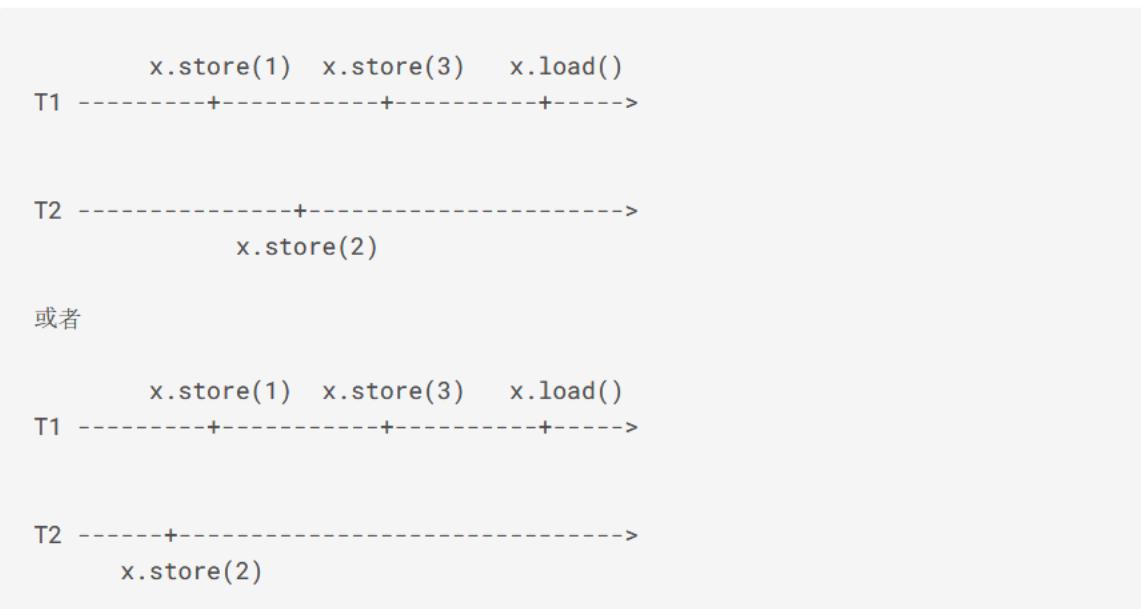
一致性模型是在并行计算中，定义了对共享数据的访问在多个处理器或线程间如何被看到的规则。并行执行的多个线程，从某种宏观层面上讨论，可以粗略的视为一种分布式系统。在分布式系统中，任何通信乃至本地操作都需要消耗一定时间，甚至出现不可靠的通信。每个线程可以对应为一个集群节点，而线程间的通信也几乎等价于集群节点间的通信。削弱进程间的同步条件从而提升效率，通常我们会考虑四种不同的一致性模型：

1. 线性一致性：又称强一致性或原子一致性。它要求任何一次读操作都能读到某个数据的最近一次写的数据，并且所有线程的操作顺序与全局时钟下的顺序是一致的。



在这种情况下线程T1, T2对x的两次写操作是原子的，且x.store(1)是严格的发生在x.store(2)之前，x.store(2)严格的发生在x.load()之前。值得一提的是，**线性一致性对全局时钟的要求是难以实现的**，这也是人们不断研究比这个一致性更弱条件下其他一致性的算法的原因。

2. 顺序一致性：同样要求任何一次读操作都能读到数据最近一次写入的数据，但未要求与全局时钟的顺序一致。



在顺序一致性的要求下，x.load()必须读到最近一次写入的数据，因此x.store(2)与x.store(1)并无任何先后保障，即只要T2的x.store(2)发生在x.store(3)之前即可。

3. 因果一致性：它的要求进一步降低，只需要有因果关系的操作顺序得到保障，而非因果关系的操作顺序则不做要求。

```
a = 1      b = 2  
T1 -----+-----+----->
```

```
T2 -----+-----+-----+----->  
x.store(3)      c = a + b      y.load()
```

或者

```
a = 1      b = 2  
T1 -----+-----+----->
```

```
T2 -----+-----+-----+----->  
x.store(3)      y.load()      c = a + b
```

亦或者

```
b = 2      a = 1  
T1 -----+-----+----->
```

```
T2 -----+-----+-----+----->  
y.load()      c = a + b      x.store(3)
```

上面给出的三种例子都是属于因果一致的，因为整个过程中，只有c对a和b产生依赖，而x和y在此例子中表现为没有关系（但实际情况中我们需要更详细的信息才能确定x与y确实无关）。这是分布式系统中常见的一致性模型，允许系统在保证因果关系的前提下优化性能。

4. 最终一致性：是最弱的一致性要求，它只保障某个操作在未来的某个时间节点上会被观察到，但并未要求被观察到的时间。因此我们甚至可以对此条件稍作加强，例如规定某个操作被观察到的时间总是有界的。

内存一致性

为了追求极致的性能，实现各种强度要求的一致性，C++11为原子操作定义了六种不同的内存顺序 std::memory_order 的选项，表达了四种多线程间的同步模型（PS：太高级了x，得对体系结构比较熟练了x）：

1. 宽松模型：在此模型下，单个线程内的原子操作都是顺序执行的，不允许指令重排，但不同线程间原子操作的顺序是任意的。类型通过 std::memory_order_relaxed 指定。
2. 释放/消费模型：在此模型中，我们开始限制进程间的操作顺序，如果某个线程需要修改某个值，但另一个线程会对该值的某次操作产生依赖，即后者依赖前者。具体而言，线程A完成了三次对x的写操作，线程B仅依赖其中第三次x的写操作，与x的前两次写行为无关，则当A主动 x.release() 时候（即使用 std::memory_order_release），选项 std::memory_order_consume 能够确保B在调用 x.load() 时候观察到A中第三次对x的写操作。
3. 释放/获取模型：在此模型下，我们可以进一步加紧对不同线程间原子操作的顺序的限制，在释放 std::memory_order_release 和获取 std::memory_order_acquire 之间规定时序，即发生在释放（release）操作之前的所有写操作，对其他线程的任何获取（acquire）操作都是可见的，亦即发生顺序（happens-before）。可以看到，std::memory_order_release 确保了它之前的写操作不会发生在释放操作之后，是一个向后的屏障（backward），而 std::memory_order_acquire 确保了它之前的写行为不会发生在该获取操作之后，是一个向前的屏障（forward）。对于选项

`std::memory_order_acq_rel` 而言，则结合了这两者的特点，唯一确定了一个内存屏障，使得当前线程对内存的读写不会被重排并越过此操作的前后。

4. 顺序一致模型：在此模型下，原子操作满足顺序一致性，进而可能对性能产生损耗。可显式的通过 `std::memory_order_seq_cst` 进行指定。

文件系统

相对来说比较简单，因为用过go的file库

文件系统库（`std::filesystem`）在C++17中被引入，提供了一套用于操作文件系统的高级接口。它允许开发人员以跨平台的方式执行文件和目录的创建、查询和管理等操作。这个库最初是基于Boost.Filesystem库开发的，后来经过标准化过程被纳入C++标准库中。

8.1 文档与链接

- **C++官方文档**: cppreference.com 提供了关于 `std::filesystem` 库的详细文档，包括各种功能的使用方法、示例代码和接口说明。
- **Boost.Filesystem文档**: 虽然 `std::filesystem` 已经是C++标准的一部分，但了解其来源的 [Boost.Filesystem库](#) 仍然非常有用，特别是对于使用旧版本C++标准的项目。

8.2 std::filesystem

`std::filesystem` 库提供了一系列操作文件系统的功能，包括文件和目录的创建、复制、删除，文件大小查询，路径操作等。下面是一些主要组件和功能的概述：

- **路径 (`std::filesystem::path`)**：表示文件系统中的路径，可以是文件或目录。`path` 类提供了丰富的接口来操作和查询路径。
- **文件状态和属性**：`std::filesystem` 提供了函数来查询文件的状态（如是否存在、是否为目录）和属性（如文件大小、最后修改时间）。
- **文件和目录的操作**：包括创建、复制、移动、删除文件和目录的函数。这些操作对于文件管理和维护非常重要。
- **目录遍历**：`std::filesystem` 支持递归和非递归地遍历目录，让开发者可以轻松地处理目录中的所有文件。
- **文件系统错误处理**：通过 `std::filesystem::filesystem_error` 异常和错误代码，库提供了一套错误处理机制。

示例代码

下面是一个简单的示例，展示了如何使用 `std::filesystem` 库来查询一个文件的大小并打印出来：

```
#include <iostream>
#include <filesystem>
namespace fs = std::filesystem;

int main() {
    fs::path filePath = "/path/to/your/file.txt";
    if (fs::exists(filePath) && fs::is_regular_file(filePath)) {
        auto fileSize = fs::file_size(filePath);
        std::cout << "File size: " << fileSize << " bytes" << std::endl;
    } else {
        std::cout << "File does not exist or is not a regular file." <<
    std::endl;
    }
    return 0;
}
```

这段代码首先检查指定路径的文件是否存在且为普通文件，如果是，则查询并打印其大小。

在C++中，基本数据类型的大小（以字节为单位）并不是固定的，而是依赖于编译器实现和运行平台（操作系统和硬件架构）。

- char: 1字节
- int: 4字节（在32位和64位系统上普遍如此）
- long:
 - 在32位系统上通常是4字节
 - 在64位系统上可能是4字节（如Windows）或8字节（如Linux和MacOS）
- float: 4字节
- long long: 8字节

其他杂项

新类型

long long int

long long int并不是C++11最先引入的，其实早在C99，long long int就已经被纳入C标准中（long long），所以大部分的编译器早已支持。C++11的工作则是正式把它纳入标准库，规定了一个long long int类型至少具备64位的比特数。

noexcept的修饰和操作

C++11将异常的声明简化为以下两种情况，并使用noexcept对这两种行为进行限制：

1. 函数可能抛出任何异常
2. 函数不能抛出任何异常

```
void may_throw(); // 可能抛出异常
void no_throw() noexcept; // 不可能抛出异常
```

使用noexcept修饰过的函数如果抛出异常，编译器会使用std::terminate()来立即终止程序运行。

noexcept还能够做操作符，用于操作一个表达式，当表达式无异常时，返回true，否则返回false。

```
#include <iostream>
void may_throw() {
    throw true;
}
auto non_block_throw = []{
    may_throw();
};

void no_throw() noexcept {
    return;
}

auto block_throw = []() noexcept {
    no_throw();
};

int main()
{
    std::cout << std::boolalpha
        << "may_throw() noexcept? " << noexcept(may_throw()) << std::endl
```

```

        << "no_throw() noexcept? " << noexcept(no_throw()) << std::endl
        << "lmay_throw() noexcept? " << noexcept(non_block_throw()) << std::endl
        << "lno_throw() noexcept? " << noexcept(block_throw()) << std::endl;
    return 0;
}

```

noexcept修饰完一个函数之后能够起到封锁异常扩散的功效，如果内部产生异常，外部也不会触发。

```

try {
    may_throw();
} catch (...) {
    std::cout << "捕获异常，来自 may_throw()" << std::endl;
}
try {
    non_block_throw();
} catch (...) {
    std::cout << "捕获异常，来自 non_block_throw()" << std::endl;
}
try {
    block_throw();
} catch (...) {
    std::cout << "捕获异常，来自 block_throw()" << std::endl;
}

/*
捕获异常，来自 may_throw()
捕获异常，来自 non_block_throw()
*/

```

字面量

传统C++里面要编写一个充满特殊字符的字符串其实是非常痛苦的一件事情，比如一个包含HTML本体的字符串需要添加大量的转义符，例如一个Windows上的文件路径经常会：`C:\\File\\To\\Path`。
C++11提供了原始字符串字面量的写法，可以在一个字符串前方使用R来修饰这个字符串，同时，将原始字符串使用括号包裹。

```

#include <iostream>
#include <string>

int main() {
    std::string str = R"(C:\\File\\To\\Path)";
    std::cout << str << std::endl;
    return 0;
}

```

自定义字面量

(感觉没太大用)

C++11引进了自定义字面量的能力，通过重载双引号后缀运算符实现。自定义字面量支持四种字面量：

```

// 字符串字面量自定义必须设置如下的参数列表
std::string operator"" _wow1(const char *wow1, size_t len) {
    return std::string(wow1)+"ooooooooooooow, amazing";
}

```

```

}

std::string operator"" _wow2 (unsigned long long i) {
    return std::to_string(i)+"woooooooooooooow, amazing";
}

int main() {
    auto str = "abc"_wow1;
    auto num = 1_wow2;
    std::cout << str << std::endl;
    std::cout << num << std::endl;
    return 0;
}

```

- 整型字面量：重载时必须使用unsigned long long、const char *、模板字面量算符参数，在上面的代码中使用的是前者；
- 浮点型字面量：重载时必须使用long double、const char *、模板字面量算符；
- 字符串字面量：必须使用 (const char *, size_t) 形式的参数表；
- 字符字面量：参数只能是 char, wchar_t, char16_t, char32_t 这几种类型。

内存对齐

C++ 11引入了两个新的关键字alignof和alignas来支持对内存对齐进行控制。alignof关键字能够获得一个与平台相关的std::size_t类型的值，用于查询该平台的对齐方式。当然我们有时候并不满足于此，甚至希望自定义结构的对齐方式，同样，C++ 11还引入了alignas来重新修饰某个结构的对齐方式。其中std::max_align_t要求每个标量类型的对齐方式严格一样，因此它几乎是最大标量没有差异，进而大部分平台上得到的结果为long double，因此我们这里得到的AlignasStorage的对齐要求是8或16。

```

#include <iostream>

struct Storage {
    char     a;
    int      b;
    double   c;
    long long d;
};

struct alignas(std::max_align_t) AlignasStorage {
    char     a;
    int      b;
    double   c;
    long long d;
};

int main() {
    std::cout << alignof(Storage) << std::endl;
    std::cout << alignof(AlignasStorage) << std::endl;
    return 0;
}
/*
8
16
*/

```


在main执行之前和之后执行的代码可能是什么

main之前主要是初始化系统相关资源：

- 栈指针
- 静态变量和全局变量 (.data段内容)
- 将未初始化的全局变量赋初值，比如int为0，bool为false，指针为nil (.bss段内容)
- 全局对象初始化，构造函数
- 将main函数的参数，argc, argv传递给main函数，正式执行main函数
- _attribute_((constructor))

main函数执行之后：

- 全局对象的析构函数
- atexit注册一个函数，会在main之后执行
- _attribute_((destructor))

结构体内存对齐

- 结构体成员按照生命顺序存储，第一个成员地址就是结构体地址
- 未特殊指明，结构体按size最大的成员对齐（整个结构体的大小需要是其整数倍）

c++11后引入alignas关键字指定结构体的对齐方式，alignof关键字计算类型的对齐方式。alignas可能会失效，比如小于最小自然对齐单位。

自然对齐：

- `uint8_t` 类型通常只需要1字节对齐，因为它只占用1字节。
- `uint16_t` 类型通常需要2字节对齐，这意味着它的地址应该是2的倍数。
- `uint32_t` 和更大的类型，如 `uint64_t`，则可能需要更大的对齐，通常是4字节或8字节。

```
// alignas 生效的情况

struct Info {
    uint8_t a;
    uint16_t b;
    uint8_t c;
};

std::cout << sizeof(Info) << std::endl; // 6 2 + 2 + 2
std::cout << alignof(Info) << std::endl; // 2

struct alignas(4) Info2 {
    uint8_t a; // 自然对齐是1字节，但是b的开始地址需要满足是2字节的倍数，因此a后面补充1字节
    uint16_t b; // 按照2字节自然对齐
    uint8_t c; // 自然对齐是1字节，但是为了满足结构体按4字节对齐，c后面补充3字节
};

std::cout << sizeof(Info2) << std::endl; // 8 2 + 2 + 2 + 2
std::cout << alignof(Info2) << std::endl; // 4
```

指针和引用的区别

- 指针是一个变量，存储的是地址；引用和原变量实质是一个东西，是原变量的别名
- 指针可以有多级，引用只有一级
- 指针可以为空，引用不能为空并且定义时必须初始化
- sizeof指针是指针大小，sizeof引用是引用所指变量大小

传递参数时，什么时候使用指针，什么时候使用引用

- 使用类对象作为参数传递的时候使用引用，这是C++类对象传递的标准
- 参数是可选的，即允许不传递任何对象，使用指针更好
- 动态分配的对象，通常使用指针

堆和栈的区别

- 栈由系统自动分配，速度快，不会有碎片
- 堆由程序员手动申请和释放，速度慢，会有碎片
- 堆向上，向高地址方向增长
- 栈向下，向低地址方向增长
- 堆是不连续的内存区域，大小受限于计算机系统中有效的虚拟内存；栈是连续的，严格按照函数调用的顺序进行。每当一个函数调用发生，一个新的栈帧就会被压入栈顶；函数返回时，栈帧被弹出。大小是操作系统预定好的

栈更快还是堆更快

- 栈更快，有操作系统底层的支持，有专门的寄存器存放，有专门的指令入栈出栈
- 堆的操作有C++函数库提供，并且需要算法计算合适大小的内存，获取堆的内容需要两次访问，第一次访问指针，第二次根据指针保存的地址访问内存。

区别以下指针类型

```
int *p[10]
int (*p)[10]
int *p(int)
int (*p)(int)
```

- 第一个表示一个int*类型的指针数组，大小为10，每个元素都是int*
- 第二个表示一个指向int[10]的指针p
- 第三个表示函数名为p，参数是int，返回值是int*
- 第四个表示函数指针，该指针指向的函数参数是int，返回值是int

new/delete与malloc/free的异同

- 两者都可以用于内存的动态申请和释放
- 前者是C++运算符，后者是C/C++语言标准函数
- new是类型安全的，会返回正确的类型指针，不允许隐式类型转换，比如 int *p = new float[2]; // 编译错误
- malloc返回的是void*，需要强制转换为适当类型

- new会自动调用类型的构造函数， delete会调用析构函数； malloc和free则不会
- new在无法分配内存会抛出异常； malloc会返回nullptr

new和delete如何实现

- new： 调用operator new的标准库函数， 分配足够大内存， 运行该类型的构造函数并初始化， 返回分配构造好的对象的指针
- delete： 调用指向对象的析构函数， 调用operator delete的标准库函数释放内存

为什么还需要有new/delete， 直接malloc/free不好吗

malloc和delete不能指向构造函数和析构函数， 非基本数据类型通常需要

被free的内存是立即返还给操作系统吗

不是， 会使用双链表保存起来， 下一次申请内存时会从这些内存中寻找合适的返回， 避免频繁的系统调用， 同时对小内存进行合并， 避免过多的内存碎片。

为什么用双链表：

- 双向遍历
- 方便合并
- 快速插入和删除

宏定义和函数的区别

- 宏在预处理阶段完成替换， 被替换的文本参与编译， 等于直接插入代码。 无需函数调用， 执行起来更快
- 函数调用需要跳转到具体函数， 需要更多开销
- 宏定义不需要在最后加分号； typedef需要

宏定义和typedef

- 宏主要定义复杂书写内容和常量
- typedef主要定义类型别名
- 宏不需要最后加分号； typedef需要

变量声明和定义区别

- 声明只是把变量的声明位置和类型提供给编译器， 不分配内存空间
- 定义需要在定义的地方为其分配存储空间
- 相同变量可以在多处声明（外部变量extern）， 定义只能在一处

strlen和sizeof

- sizeof是运算符， strlen是库函数
- sizeof函数可以是任何数据的类型或数据； strlen只能说字符指针，并且以'\0'结尾

```
int main(int argc, char const *argv[]){
    const char* str = "name";
    sizeof(str); // 取的是指针str的长度，是8
    strlen(str); // 取的是这个字符串的长度，不包含结尾的 \0。大小是4
    return 0;
}
```

一个指针多少字节

看机器位数，64位/8字节，32位/4字节

常量指针和指针常量

- 常量指针：这个指针是常量，也就是地址不能改变，必须初始化，const在*后面，比如int * const p
- 指针常量：指针指向的值是常量，const在*前面，比如int const *p或const int *p

a和&a有什么区别

```
int a[10];
int (*p)[10] = &a;
```

a是数组名，也是数组首元素地址，+1表示地址加上一个int类型大小，*(a + 1)为a[1]

&a是数组的指针，类型为int (*)[10]，+1表示数组首地址+加上整个数组的便宜

C++和python区别

- C++是编译型语言，需要对特定平台进行编译，Python是脚本语言，解释执行，更方便跨平台，C++效率更高

C++和C的区别

C++面向对象

C++和Java的区别

- java语法更简洁，通过jvm可以实现一次编码，到处运行；而C++则需要针对不同的平台系统进行编译
- java没有指针的概念
- java有gc
- c++底层

struct和class的区别

- struct和class都有成员函数，共有和私有部分，任何用class完成的工作，都可以同struct完成
- 不对成员指定公私有，struct默认是公有，class默认是私有；类似的，struct默认是共有继承，class是私有继承

C++和C中struct的区别：

- c++中的struct可以作为类的特例，可以有访问权限，可以有成员函数，可以继承，可以实现多态

define宏定义和const的区别

- define在预处理的时候起作用，只进行替换，不做检查，最好加上一个大括号包住全部内容，不然容易出错
- const有数据类型，编译器可以进行类型安全检查
- 宏定义的数据由于是替换，不分配内存空间；const定义的是值不能改变，要分配内存空间
- 最好是用const替换define，出错了方便debug

C++中const和static的作用

const声明变量为常量，初始化后就不能修改：

- 修饰变量
- 修饰指针：常量指针（指针是常量，const在*后面），指针常量（指针指向的是常量，const在*前面）
- 修饰成员函数：表示这个函数不会修改类成员变量

static：

- 修饰局部变量，使变量值可以在函数调用之间持久化
- 修饰全局变量，使得作用域只能在声明它的文件内
- 修饰类的成员，使成员属于类本身，也就是所有类对象共享一个成员
- 修饰类的成员函数，使得函数可以在没有类实例的情况下被调用，只可以访问类的静态成员
- 单例模式，通过局部静态变量实现线程安全的单例模式（C++11之后，规定局部静态变量的线程安全初始化）

```
class Singleton {  
private:  
    // 私有构造函数，防止外部构造  
    Singleton() {};  
    // 禁止拷贝构造  
    Singleton(const Singleton& other) = delete;  
    // 禁止拷贝赋值运算符
```

```

    singleton& operator=(const singleton& other) = delete;
public:
    // 全局访问点
    static singleton& getInstance() {
        static singleton instance; // 局部静态变量
        return instance;
    }
}

int main() {
    singleton& s = singleton::getInstance();
}

```

C++的顶层const和底层const

常量指针：顶层const

指针常量：底层const

数组名和指针（指向数组首元素的地址）区别

- 都可以通过增减偏移量来访问数组中的元素
- 数组名不是真正意义上的指针，可以看作数组首元素的地址
- 数组名不可以赋值，指针可以赋值更加灵活
- 数组名作为形参进行传递，退化成了一般指针，可以自增、自减，sizeof大小也变了

final和override关键字

C++11中被引入

final:

- 修饰类，表示该类不能被继承，防止类的进一步派生
- 修饰虚函数表示，该虚函数在当前类是最终实现，不能在任何派生类中被重写

override:

- 只用于虚函数，明确指出需要重写基类中的一个虚函数，有助于编译器检查函数签名是否匹配基类中的一个虚函数

拷贝初始化和直接初始化

- 直接初始化，通常是()，可以调用explicit构造函数，更高效，允许直接的对象构造
- 拷贝初始化，通常是=，只能调用非explicit构造函数，会分配一个临时对象，再使用拷贝构造函数

```

string str1("I am a string");//语句1 直接初始化
string str2(str1);//语句2 直接初始化，str1是已经存在的对象，直接调用拷贝构造函数对str2进行
//初始化
string str3 = "I am a string";//语句3 拷贝初始化，先为字符串"I am a string"创建临时对
//象，再把临时对象作为参数，使用拷贝构造函数构造str3
string str4 = str1;//语句4 拷贝初始化，这里相当于隐式调用拷贝构造函数，而不是调用赋值运算符函
//数

```

初始化和赋值的区别

- 初始化发生在对象的创建，为成员分配初始值，比如构造函数完成，也可以初始化列表或直接在类定义中为成员指定默认值
- 赋值发生在对象存在之后，改变其内容，涉及赋值运算符=
- 对于简单数据类型，两者没什么区别
- 对于类和复杂数据类型，可能有区别，比如重载赋值运算符

extern “C”

加上后可以正确的调用C语言的代码，相当于告诉编译器这部分代码是C写的，按照C语言进行编译。

这个语句一般位于.h头文件或.cpp文件 (.c文件不支持)

extern的作用

- 在多个文件中使用同一个全局变量时，你可以在一个文件中定义该变量，并在其他文件中使用 `extern` 来声明它，以便使用。这样做可以避免多重定义错误，并确保所有文件中的变量都引用同一个内存位置。
- `extern “C”`

野指针和悬空指针

野指针：没有初始化过的指针，解决办法：使用指针最好都初始化或为nullptr

```
int main(void) {  
  
    int* p;      // 未初始化  
    std::cout << *p << std::endl; // 未初始化就被使用  
  
    return 0;  
}
```

悬空指针：

- 指针指向的对象被删除或释放，比如使用`delete`删除了一个动态分配的对象，但指向该对象的指针任然存在
- 函数返回局部变量的地址
- 指针未初始化

```
int main(void) {  
    int * p = nullptr;  
    int* p2 = new int;  
  
    p = p2;  
  
    delete p2;  
}  
  
// p和p2就是悬空指针，需要如下设置  
// p = p2 = nullptr
```

解决办法：

- 及时设置为nullptr
- 使用智能指针自动管理内存
- 避免返回局部变量的地址

C和C++的类型安全

什么是类型安全：编译时能够确保类型的正确使用，避免类型错误冲突的运行时错误

C语言相对较少类型安全，允许广泛的类型转换，容易出错

C++类型安全更多，比如：

- 强类型检查，编译时更严格的类型检查，函数重载参数必须明确
- 类型转换符：C++ 提供了 `static_cast`、`dynamic_cast`、`const_cast` 和 `reinterpret_cast`，使类型转换更明确和安全
- 智能指针
- 模板提供了编译时的类型检查

C++中的重载、重写（覆盖）和隐藏

重载：函数名相同，但是参数类型和数目不同，不能仅依靠返回值不同来区分，和是否是虚函数无关

重写：在派生类中重写基函数，函数名，参数类型和个数，返回值都得一样

重载和重写：

- 重载是不同函数之间的水平关系，重写是父类和子类的垂直关系
- 参数，返回值不同
- 重写根据调用的类型确定，重载根据形参和实参确定

隐藏：派生类的函数隐藏了基类的同名函数。比如基类函数不是虚函数；或者两个函数参数不同，不管是虚函数，都会被隐藏。

```
// 父类
class A {
public:
    virtual void fun(int a) { // 虚函数
        cout << "This is A fun " << a << endl;
    }
    void add(int a, int b) {
        cout << "This is A add " << a + b << endl;
    }
};

// 子类
class B: public A {
public:
    void fun(int a) override { // 覆盖
        cout << "this is B fun " << a << endl;
    }
    void add(int a) { // 隐藏
        cout << "This is B add " << a + a << endl;
    }
};
```

```

int main() {
    // 基类指针指向派生类对象时，基类指针可以直接调用到派生类的覆盖函数，也可以通过 :: 调用到基
    // 类被覆盖

    // 的虚函数；而基类指针只能调用基类的被隐藏函数，无法识别派生类中的隐藏函数。

    A *p = new B();
    p->fun(1);           // 调用子类 fun 覆盖函数
    p->A::fun(1);        // 调用父类 fun
    p->add(1, 2);
    // p->add(1);        // 错误，识别的是 A 类中的 add 函数，参数不匹配
    // p->B::add(1);      // 错误，无法识别子类 add 函数
    return 0;
}

```

C++有哪几种构造函数

- 默认构造函数：没有任何参数
- 参数化构造函数：接受参数
- 拷贝构造函数：ClassName(const ClassName& other)
- 移动构造函数：接受一个同类型对象的右值引用作为参数。用于支持将一个临时对象的资源“移动”到一个新对象中，而不是复制。形式通常为 `ClassName(ClassName&& other)`
- 委托构造函数：同一个类中的一个构造函数调用另一个构造函数
- 继承构造函数：C++11，允许从基类继承构造函数

深拷贝和浅拷贝

浅拷贝：只拷贝一个指针，没有新开辟一个地址，拷贝的指针的原来的指针指向同一个地址，如果原来指针指向的资源释放了，再释放浅拷贝指针的资源就会出现错误。

深拷贝：拷贝值，并且开辟一块新的空间来存放，即使原来的对象不存在，也不会对深拷贝的值有影响。

```

#include <iostream>
#include <string.h>
using namespace std;

class Student
{
private:
    int num;
    char *name;
public:
    Student(){
        name = new char(20);
        cout << "Student" << endl;
    };
    ~Student(){
        cout << "~Student" << &name << endl;
        delete name;
        name = NULL;
    };
    Student(const Student &s){//拷贝构造函数
        //浅拷贝，当对象的name和传入对象的name指向相同的地址
    }
}

```

```

        name = s.name;
        //深拷贝
        //name = new char(20);
        //memcpy(name, s.name, strlen(s.name));
        cout << "copy Student" << endl;
    };
};

int main()
{
    { // 花括号让s1和s2变成局部对象，方便测试
        Student s1;
        Student s2(s1); // 复制对象
    }
    system("pause");
    return 0;
}

//浅拷贝执行结果:
//Student
//copy Student
//~Student 0xfffffed0c3ec0
//~Student 0xfffffed0c3ed0
//*** Error in `/tmp/815453382/a.out': double free or corruption (fasttop):
0x0000000001c82c20 ***

//深拷贝执行结果:
//Student
//copy Student
//~Student 0x7fffebca9fb0
//~Student 0x7fffebca9fc0

```

内联函数 (inline) 和宏定义

宏在预处理阶段做简单替换，内联函数可以在编译时进行类型检查。

内联函数在编译时直接将函数代码嵌入，省去函数调用开销。

宏不方便debug，并且需要用{}括起来，否则容易用歧义，内联函数则不会，使用宏的地方都可以使用inline。

```

#include <iostream>

// 使用inline关键字定义内联函数
inline int max(int x, int y) {
    return (x > y) ? x : y;
}

int main() {
    int a = 5;
    int b = 10;

    // 调用内联函数
    std::cout << "The maximum of " << a << " and " << b << " is " << max(a, b) <<
    std::endl;

    return 0;
}

```

```
}
```

public, private和protected

public: 类的内部外部都可以访问

private: 只能类内部访问、友元函数或友元类访问

protected: 类的内部和其派生类中访问

继承:

- public: 如果一个类以 public 方式继承另一个类, 基类的 public 成员在派生类中仍然是 public, protected 成员在派生类中仍然是 protected, 而 private 成员仍然是不可访问的。
- private: 如果一个类以 private 方式继承另一个类, 基类的 public 和 protected 成员都会成为派生类的 private 成员, 而基类的 private 成员在派生类中仍然是不可访问的。
- protected: 如果一个类以 protected 方式继承另一个类, 基类的 public 和 protected 成员都会成为派生类的 protected 成员, 而基类的 private 成员仍然是不可访问的。

如何用代码判断大小端存储

大端存储: 数据的高字节存储在低地址

小端存储: 数据的低字节存储在低地址

x86架构一般是小端存储, arm和mips大小端都支持。

而网络传输使用大端存储, 即网络字节顺序, 这是一种约定, 因此使用socket编程需要转换, c和c++提供了一些函数。

- htonl(): 将主机字节顺序的无符号长整型转换为网络字节顺序。
- htons(): 将主机字节顺序的短整型转换为网络字节顺序。
- ntohl(): 将网络字节顺序的无符号长整型转换为主机字节顺序。
- ntohs(): 将网络字节顺序的短整型转换为主机字节顺序。

可以使用强制类型转换判断:

```
#include <iostream>
using namespace std;
int main()
{
    int a = 0x1234;
    unsigned char c = (unsigned char)(a); // 使用 unsigned char 以避免符号问题
    if (c == 0x12)
        cout << "big endian" << endl;
    else if (c == 0x34)
        cout << "little endian" << endl;
}
```

volatile、mutable和explicit

volatile: 防止编译器对变量的访问进行优化, 确保每次读取或写入都对应原始内存地址

mutable: 放宽类中const函数的限制, const函数保证不会修改类成员变量, 有时候需要修改一些类成员, 可以加上mutable, 常用于需要修改的是类的内部状态, 不影响外部可观察行为。

explicit：防止C++隐式类型转换

```
class MyClass {  
public:  
    explicit MyClass(int a) {  
        // Constructor code  
    }  
};  
  
void func(MyClass m) {  
    // Function code  
}  
  
int main() {  
    MyClass obj = MyClass(10); // 正确：显式调用  
    // MyClass obj2 = 10;      // 错误：不能隐式转换  
    func(MyClass(20));       // 正确：显式调用  
    // func(20);             // 错误：不能隐式转换  
}
```

什么情况会调用拷贝构造函数

1. 用类的实例化对象去初始化另一个对象
2. 函数的参数是类的对象（非引用传递）

C++有几种类型的new

- plain new: 最普通的new, 也就是常用的new, 如果内存空间分配失败, 会panic, 需要catch捕捉异常
- nothrow new: 内存空间分配失败的情况下不抛出异常, 返回nullptr
- placement new: 允许在已经分配的内存上构造对象

```
// nothrow new
#include <new> // 必须包含这个头文件

void* place = ...; // 指向预先分配好的内存
MyClass* myObject = new(place) MyClass(arguments);

// placement new
#include <iostream>
#include <string>
using namespace std;

int main()
{
    char *p = new(nothrow) char[10e11];
    if (p == NULL)
    {
        cout << "alloc failed" << endl;
    }
    delete p;
    return 0;
}
//运行结果: alloc failed
```

C++异常处理的方法

常见的异常:

- 数组下标越界
- 除数为0
- 动态分配空间不足

异常处理: try、throw和catch

```
#include <iostream>
using namespace std;
int main()
{
    double m = 1, n = 0;
    try {
        cout << "before dividing." << endl;
        if (n == 0)
            throw - 1; //抛出int型异常
        else if (m == 0)
            throw - 1.0; //抛出 double 型异常
        else
            cout << m / n << endl;
    }
```

```

        cout << "after dividing." << endl;
    }
    catch (double d) {
        cout << "catch (double)" << d << endl;
    }
    catch (...) {
        cout << "catch (...)" << endl;
    }
    cout << "finished" << endl;
    return 0;
}
//运行结果
//before dividing.
//catch ...
//finished

```

先执行try，如果没有发生异常，不会进入catch，否则throw出异常，catch进行捕获。

catch(...)表示捕获任何异常，catch里面还可以throw通知上一层调用者。

还可以通过异常类

形参和实参

形参就是函数定义声明时那个变量，只有在调用时才为其分配内存单元，调用结束释放内存单元，也就是只在函数内部有效。

实参就是调用函数时，传递给形参的变量，做一次拷贝，需要与形参相匹配。如果不是指针参数，相当于两个变量。

值传递、指针传递和引用传递

值传递：对实参对象做一次拷贝，赋值给形参，大小根据参数类型确定，不会修改原始数据。

指针传递：只对地址做一次拷贝，大小为固定的地址大小，通过指针可以对原始数据进行修改。

引用传递：不涉及数据拷贝，是原始数据的一个直接别名。

指针传递和引用传递效率比值传递高，更推荐引用传递，高效并且语义更好。

静态变量什么时候初始化

初始化只有一次，赋值可以有多次。

- 全局静态变量，它会在程序开始时初始化。
- 局部静态变量：如果你在函数内部定义了一个静态变量，它会在该函数的第一次调用时在该变量首次使用的地方初始化。

const关键字的作用

1. 阻止一个变量被改变，使用const关键字定义变量通常需要对其初始化。
2. 常量指针，指针常量
3. 修饰函数的形参，避免这个参数被改变
4. 修饰类的成员函数，表面这个函数不会对成员变量进行修改
5. 修饰返回值，避免返回一个左值
6. const对象只能调用const成员函数

7. const变量可以通过const_cast转换为非const类型

什么是类的继承

首先类和类之间的关系：

1. has-a, 类的类成员是另外一个类
2. use-a, 通过成员函数
3. is-a, 继承, 具有传递性

继承就是一个类继承了另外一个类的属性和方法，子类/派生类，父类/基类，子类对象可以当作父类对象使用。

从汇编层面去解释一下引用

引用相对于指针更加安全，为什么更加安全，比如野指针和空指针。

引用就是原来变量的别名，地址是相同的。引用必须在定义时初始化，并且不能重新指向。

delete p、delete []p、allocator的作用

- delete p: 释放new分配的单个对象
- delete []p: 释放new分配的数组的内存

delete除了释放内存，还会调用对象的析构函数

allocator是stl容器的默认内存管理工具

new/delete的实现原理，delete如何知道释放内存的大小

new:

- 分配内存：调用malloc为对象分配足够的内存，除了对象本身需要的空间，还包括额外的元数据，比如对象的大小和类型
- 调用构造函数：在分配的内存上调用对象的构造函数

delete:

- 调用对象的析构函数
- 释放内存：调用free来释放对象所占用的内存
- new内存分配时会存储对象大小的元数据，也得看编译器的实现

malloc申请的内存能用delete释放吗

malloc需要使用free配套使用，需要明确的大小，不能用在动态类，并且不会执行析构函数。

malloc和free的原理

malloc:

- 小于128KB，使用brk()，移动堆顶指针
- 大于128KB，使用mmap系统调用，在文件映射区进行私有匿名映射，私有表示只能这个进程访问，匿名表示和磁盘文件无关
- 申请内存是申请虚拟内存，第一次访问会发生缺页中断，然后分配物理内存
- 使用brk申请的内存不会立刻还给操作系统，会存起来下次再用，但容易产生内存碎片，因为分配的内存是连续的

- 而mmap申请的内存则会立刻还给操作系统，因此也更容易发生缺页中断

malloc、realloc、calloc的区别

名字可以区分

malloc：需要手动计算，成功返回分配内存的指针，失败返回NULL，分配的内存内容没有初始化，可能包含任意数据

```
void* malloc(size_t size);
```

calloc：需要指定每个元素的大小和数量，更方便计算大小，成功和失败返回和malloc一致，会对分配的内存区域初始化为零

```
void* calloc(size_t num, size_t size);
```

realloc：重新调整分配的内存大小，成功和失败返回和malloc一致

- 如果 new_size 大于原始大小，新分配的部分不会被初始化。
- 如果 new_size 小于原始大小，可能会丢失超出新大小部分的数据。
- 如果 ptr 是 NULL，realloc 的行为类似于 malloc(new_size)。

```
void* realloc(void* ptr, size_t new_size);
```

类成员初始化的方式，构造函数执行顺序，为什么用成员初始化列表会更快

初始化：

- 成员列表初始化：更推荐，会在对象的内存分配后，构造函数体执行前初始化成员变量，可以直接调用成员的构造函数来初始化成员
- 构造函数体内赋值初始化：先默认构造函数后再赋值（拷贝构造）

为什么更快：成员列表初始化少一次调用类的默认构造函数，对于内置类型没有差别，但为了保持写法相同也采用成员列表初始化。

执行顺序：

1. 基类构造函数
2. 成员变量初始化（按照类声明顺序，而非成员初始化列表中顺序）
3. 构造函数体

哪些情况必须用到成员列表初始化，作用是什么

因为成员列表初始化是分配内存后就初始化成员变量，在构造函数体之前；而构造函数体赋值则是先构造再初始化赋值，因此考虑必须在定义时初始化的成员：

1. const
2. 引用
3. 调用基类的非默认构造函数，也就是这个构造函数有参数

```
class Base {
public:
    Base(int x) {}
};

class Derived : public Base {
public:
    Derived(int x) : Base(x) {}
};
```

1. 有个成员是另外一个类，需要调用他的非默认构造函数

```
class Member {
public:
    Member(int x) {}

class Container {
    Member m;
public:
    Container(int x) : m(x) {}

};
```

string和char *

string对char *进行了封装：char *，容量，长度；可以动态扩展（申请原空间两倍的空间，再拷贝过去，增加新的内容），更安全

什么是内存泄露，如何检测和避免

内存泄露一般指堆内存的泄露，对于没有GC的语言，比如C/C++，需要程序员去管理这部分内存的申请和释放，如果用完忘记释放，就会导致这块内存不可用，导致内存泄露。

如何避免：

- 使用new/malloc的时候，要注意使用成对的delete/free
- 使用智能指针
- 使用继承的时候，将基类的析构函数设置为虚函数，这样不管是使用基类或派生类的对象进行删除，都会触发正确的析构过程，先派生类，后基类。否则只会触发基类的析构函数。

```
#include <iostream>

class Base {
public:
    virtual ~Base() { std::cout << "Base destructor\n"; }
};

class Derived : public Base {
private:
    int* data;
public:
    Derived() : data(new int(42)) { std::cout << "Derived constructor\n"; }
    ~Derived() {
        std::cout << "Derived destructor\n";
        delete data;
    }
};

int main() {
    Base* b = new Derived();
    delete b; // 正确调用Derived的析构函数，然后是Base的析构函数
    return 0;
}
```

构造函数是先基类->派生类；析构函数是先派生类->基类；主要是派生类可能会基类的资源

检测工具：

- Valgrind

对象复用了解吗，零拷贝呢

对象复用就是提前创建一批资源，然后存储到资源池，实现重复利用，避免频繁的创建释放开销。比如线程池、sql连接池，并发互斥。

零拷贝：减少CPU搬运数据

- DMA之前：CPU需要将数据从设备缓冲区拷贝到内核缓冲区，再到用户缓冲区
- DMA之后：每个设备都有自己的DMA，会将数据拷贝到内核缓冲区
- 共享内存，sendfile（直接将源地址到目标地址，减少一次到用户缓冲区）

vector中的emplace_back()就用了零拷贝的思路，减少不必要的拷贝；会将插入的元素直接在容器空间内的原地构造，不需要触发拷贝构造和转移构造（push_back则需要）

```
#include <vector>
#include <string>
#include <iostream>
using namespace std;

struct Person
{
    string name;
    int age;
    //初始构造函数
    Person(string p_name, int p_age): name(std::move(p_name)), age(p_age)
    {
        cout << "I have been constructed" << endl;
    }
    //拷贝构造函数
    Person(const Person& other): name(std::move(other.name)), age(other.age)
    {
        cout << "I have been copy constructed" << endl;
    }
    //转移构造函数
    Person(Person&& other): name(std::move(other.name)), age(other.age)
    {
        cout << "I have been moved" << endl;
    }
};

int main()
{
    vector<Person> e;
    cout << "emplace_back:" << endl;
    e.emplace_back("Jane", 23); //不用构造类对象

    vector<Person> p;
    cout << "push_back:" << endl;
    p.push_back(Person("Mike", 36));
    return 0;
}
//输出结果:
//emplace_back:
//I have been constructed
//push_back:
//I have been constructed
//I am being moved.
```

面向对象的特性，并举例

1. 继承：子类继承父类的属性和方法，比如人作为一个基类，拥有基本的成员比如器官，基本的方法吃饭睡觉等；到更具体的一类人，就可以继承人，然后再加上独特的成员和方法
2. 封装：将客观事物抽象成具体的类，类可以通过private、public和protected做访问控制
3. 多态：同一个事物对不同的消息展现出不同的表现，或者说允许将子类的指针赋值给父类的指针

1. 重载 (overload) : 同样的函数名，但是参数类型和数量不同（不能只通过返回值不同，但返回值可以不同），编译时多态
2. 覆盖 (override) : 虚函数，在子类中重新定义

C++的四种强制转换

都会在编译时进行检查

1. static_cast: 最常用，执行非多态类型的转换，在相关类型之间进行转换，比如整型和浮点型

```
int a = 10;
double b = static_cast<double>(a); // 将 int 转换为 double
```

2. dynamic_cast: 处理多态类型，可以在类的继承体系中安全向下转换，比如基类向派生类转换，但要求基类至少有一个虚函数

```
class Base { virtual void dummy() {} };
class Derived : public Base { int a; };

Base* base = new Derived();
Derived* derived = dynamic_cast<Derived*>(base); // 合法的向下转换
```

3. const_cast: 修改类型的const或volatile属性（禁止编译器对变量进行优化，直接从内存地址读取）

```
const int a = 10;
int* pa = const_cast<int*>(&a); // 去除常量性
*pa = 20; // 实际上这样做是未定义行为，因为a是常量
```

4. reinterpret_cast: 用于低级别的、不安全的类型重新解释

```
int* p = new int(10);
char* pc = reinterpret_cast<char*>(p); // 将 int* 转换为 char*
```

C++函数调用的压栈过程

```
#include <iostream>
using namespace std;

int f(int n)
{
    cout << n << endl;
    return n;
}

void func(int param1, int param2)
{
    int var1 = param1;
    int var2 = param2;
    printf("var1=%d, var2=%d", f(var1), f(var2)); //如果将printf换为cout进行输出，输出结果则刚好相反
}
```

```
int main(int argc, char* argv[])
{
    func(1, 2);
    return 0;
}
//输出结果
//2
//1
//var1=1,var2=2
```

1. 程序从main函数开始执行，编译器将操作系统运行状态，main函数的返回地址，main函数的参数，main函数中的变量依次压栈。
2. main调用func()，编译器将main函数的状态、func函数的返回地址、func函数的参数从右到左、func函数定义的变量依次压栈。
3. main函数调用f()，编译器将func函数的状态，f函数的返回地址、f函数的参数从右到左、f定义的变量依次压栈。

从代码可以看出，f(var1)、f(var2)依次入栈，先执行f(var2)，再执行f(var1)，最后打印整个字符串，栈中的变量依次弹出，主函数返回。

函数调用为什么使用栈，队列不行吗

栈是后进先出的，这种结构特点十分适合函数调用和递归，确保了最后调用的函数最先返回，再压栈的时候对函数的返回地址、函数的参数、函数的变量依次压栈，函数执行完毕后，会依次出栈，回到函数被调用的地方。

队列是先进先出，与函数需求不符，并且队列管理更加复杂。

coredump，怎么调试

coredump是由异常退出或终止，比如段错误生成的core文件，这个core文件会记录程序运行时的内存、寄存器状态、内存指针和堆栈等信息，可以使用gdb进行定位调试。

```
#include<stdio.h>
int main(){
    int i;
    scanf("%d", i); //正确的应该是&i，这里使用i会导致segment fault
    printf("%d\n", i);
    return 0;
}
```

```
g++ coredumpTest.cpp -g -o coredumpTest
./coredumpTest
gdb [可执行文件名] [core文件名]
```

移动构造函数

比如用对象a初始化对象b，a的作用仅是初始化，后续不在使用b，使用拷贝构造函数会把a的内容拷贝到b中。但由于a后续不再使用，那么可以直接使用a的空间来初始化b，于是有了移动构造函数。

拷贝构造函数的参数是一个左值引用，移动构造函数的参数是右值或将亡值的引用，换句话说只有用右值或将亡值的初始化另一个对象时，才会调用移动构造函数，move语句用于将一个左值变成一个将亡值。

C++将临时变量作为返回值时的处理过程

函数退出时，临时变量出栈被销毁，返回值会存储到寄存器中，与临时变量的生命周期没有关系。通常步骤如下：

1. **临时变量的构造**：在函数内部，临时变量被构造。
2. **返回值的拷贝**：在函数返回时，临时变量需要被拷贝到返回值中。这可能涉及到调用拷贝构造函数或移动构造函数。
3. **销毁临时变量**：一旦临时变量的内容被拷贝（或移动）到返回位置，该临时变量就会被销毁，即其析构函数被调用。
4. **使用返回值**：调用者接收返回值。如果涉及到进一步的拷贝或移动，可能会有额外的构造和析构发生。

但编译器会进行优化，比如返回值优化：

1. 直接构造：在返回值的存储位置直接构造临时变量，而不是在函数内部构造然后拷贝或移动到返回位置。
2. 无需拷贝/移动：通过这种方式，可以避免调用拷贝构造函数或移动构造函数。
3. 直接使用：函数返回后，调用者可以直接使用这个已经构造好的对象。

如何获得结构成员相对于结构开头的字节偏移量

使用<stddef.h>中的offsetof宏

```
#include <iostream>
#include <stddef.h>
using namespace std;

struct S
{
    int x;
    char y;
    int z;
    double a;
};

int main()
{
    cout << offsetof(S, x) << endl; // 0
    cout << offsetof(S, y) << endl; // 4
    cout << offsetof(S, z) << endl; // 8
    cout << offsetof(S, a) << endl; // 16
    return 0;
}
```

静态类型和动态类型，静态绑定和动态绑定

静态类型：对象声明时采用的类型，编译期已确定

动态类型：一个指针或引用目前所指对象的类型，运行期确定

静态绑定：绑定的是静态类型，对应的函数或属性依赖于对象的静态类型，发生在编译器

动态绑定：绑定的是动态类型，对应的函数或属性依赖于对象的动态类型，发生在运行期

一般来说对象非虚函数都是静态绑定，虚函数是动态绑定

```

#include <iostream>
using namespace std;

class A
{
public:
    /*virtual*/ void func() { std::cout << "A::func()\n"; }
};

class B : public A
{
public:
    void func() { std::cout << "B::func()\n"; }
};

class C : public A
{
public:
    void func() { std::cout << "C::func()\n"; }
};

int main()
{
    C* pc = new C(); //pc的静态类型是它声明的类型C*, 动态类型也是C*;
    B* pb = new B(); //pb的静态类型和动态类型也都是B*;
    A* pa = pc;      //pa的静态类型是它声明的类型A*, 动态类型是pa所指向的对象pc的类型C*;
    pa = pb;          //pa的动态类型可以更改, 现在它的动态类型是B*, 但其静态类型仍是声明时候
                      的A*;
    C *pnull = NULL; //pnull的静态类型是它声明的类型C*, 没有动态类型, 因为它指向了NULL;

    pa->func();      //A::func() pa的静态类型永远都是A*, 不管其指向的是哪个子类, 都是直接
                      调用A::func();
    pc->func();      //C::func() pc的动、静态类型都是C*, 因此调用C::func();
    pnull->func();   //C::func() 不用奇怪为什么空指针也可以调用函数, 因为这在编译期就确定
                      了, 和指针空不空没关系;
    return 0;
}

```

引用能否实现动态绑定, 为什么

可以的, 虚函数。

```

#include <iostream>
using namespace std;

class Base
{
public:
    virtual void fun()
    {
        cout << "base :: fun()" << endl;
    }
};

class Son : public Base
{
public:
    virtual void fun()

```

```

{
    cout << "son :: fun()" << endl;
}
// 非虚函数, b中无法调用
void func()
{
    cout << "son :: not virtual function" << endl;
}
};

int main()
{
    Son s;
    Base& b = s; // 基类类型引用绑定已经存在的Son对象, 引用必须初始化
    s.fun(); //son::fun()
    b.fun(); //son :: fun()
    return 0;
}

```

全局变量和局部变量的区别

作用域不同，一个是整个程序都可以用，一个是在局部，比如{}，两者都是离开其作用域即销毁。

全局变量在main函数执行前就被初始化，在.data段，局部变量则是运行过程在被加载，在堆栈。

指针加减需要注意什么

不要让指针指向一块未知的内存地址，指针运算转换为10进制，加减是对应个数的指针类型的长度。

```

#include <iostream>
using namespace std;

int main()
{
    int *a, *b, c;
    a = (int*)0x500;
    b = (int*)0x520;
    c = b - a;
    /*
    在 C++ 中, 两个指针相减的结果是它们之间的元素数量。由于这里的指针都是 int*, 指向整数, 每个
    整数通常占用 4 个字节(这取决于平台)。因此, 地址差 0x520 - 0x500 等于 32(十六进制的 20),
    除以每个整数的大小 4 字节, 得到 8。这是指针之间的整数元素数量。
    */
    printf("%d\n", c); // 8
    a += 0x020;
    c = b - a;
    printf("%d\n", c); // -24
    return 0;
}

```

如何判断两个浮点数是否相等

不能直接用`==`来比较两个浮点数是否相等，浮点数的计算涉及到精度问题和舍入误差，因此判断方法是看这两个浮点数的差值是否足够小，并且在某个误差范围内。

```
#include <cmath> // 引入数学库以使用fabs函数

bool areAlmostEqual(double a, double b, double epsilon = 1e-9) {
    return std::fabs(a - b) < epsilon;
}
```

指针参数传递和引用参数传递

指针参数传递：拷贝实参的地址，对指针的改变不会改变实参（想改变用指针的指针或引用），解引用会改变实参的值

引用参数传递：实参的别名，任何改变都会影响到实参

类如何实现只能静态分配和只能动态分配

静态分配：编译器为对象在栈空间分配内存

动态分配：使用`new`为对象在堆空间中分配内存

只能静态分配：将new和delete运算符重载为private属性

只能动态分配：构造函数和析构函数设置为private或protected，再提高一个静态方法来创建对象

想使用某个类作为基类，为什么基类必须定义而非声明

派生类需要继承基类的属性和方法，为了使用这些成员，基类必须定义

继承机制中对象如何转换？指针和引用之间如何转换

1. 向上转换，定义一个派生类的指针或引用再转换为基类指针，会自动进行，这是安全的
2. 向下转换，将基类指针或引用转换为派生类指针或引用，不会自动进行，因为不知道对应哪个派生类，需要使用`dynamic_cast`

C++中的组合，相比继承有什么优点

继承是is-A，组合是has-A

优点：

1. 可以隐藏类的实现细节，只能通过成员类来访问对应类的功能，无需了解其实现
2. 耦合度低，继承中基类的某个属性和方法改变，派生类需要修改
3. 过深的继承可能会难以维护
4. 更好的代码复用

函数指针

函数指针是指一个指针指向一个由特定参数列表和返回值的函数类型，比如下面的函数指针pf，指向函数列表为两个const int&参数并且返回值是int的函数，注意括号是必须的：

```
int (*pf) (const int&, const int&);
```

可以将函数指针作为参数传递给函数，比如实现回调函数。

内存对齐及其原因

1. 分配内存的顺序是声明的顺序
2. 每个变量相较于起始位置偏移量必须是其类型大小的整数倍，如果不是则需要补齐即内存对齐
3. 整个结构体大小默认是变量类型大小最大的整数倍

结构体变量比较是否相等

重载“==”操作符

函数调用过程栈的变化，返回值和参数变量哪个先入栈

1. **参数入栈**：在大多数调用约定中，调用者（caller）将参数按顺序（从右到左）。
2. **调用指令执行**：执行 CALL 指令，这时会将返回地址（即调用指令之后的下一条指令的地址）压入栈中。这个返回地址用于在函数执行完后跳回到调用者的代码。
3. **栈帧设置**：被调用的函数（callee）可能会创建自己的栈帧，用于存储局部变量、保存寄存器状态等。这一步骤通常包括移动栈指针来为局部变量腾出空间。
4. **函数执行**：函数体内的代码执行，使用栈上的参数和局部变量。
5. **返回值处理**：函数的返回值通常存放在寄存器中，或者通过栈传递（对于较大的数据类型，如大的结构体）。
6. **栈帧清理**：在函数返回前，需要清理局部变量，并将栈指针恢复到调用前的状态。
7. **返回到调用点**：通过 RET 指令使用之前压入栈的返回地址跳回到调用者的代码。

define、const、typedef、inline

通常define定义常量，并且不带类型，只在预处理阶段进行替换

const定义带类型的常量，在编译、链接过程中起作用，更加安全，会进行类型检查，推荐

typedef通常定义类型的别名，在编译阶段有效，做类型检查

inline是函数，也是替换，但在编译阶段起作用，有类型检查，更安全

printf的原理

函数调用压栈，参数从右到左入栈，出栈相反

模板类为什么都是放在一个h文件中

普通类分布在头文件和源文件

1. **编译器实例化**: 模板类或函数的代码在编译时需要根据具体的类型进行实例化。编译器必须在编译时看到模板定义的完整内容，以生成对应特定类型的实例化代码。
2. **链接问题**: 如果模板定义放在源文件中，每个源文件编译时只会实例化它所用到的模板类型。这会导致在其他源文件中使用相同模板但不同类型时，因为缺乏必要的模板实例化代码而无法链接。
3. **避免重复实例化**: 将模板定义放在头文件中，每次包含这个头文件的时候，相关的模板定义就会在该编译单元中可见，从而允许编译器为所需的具体类型生成模板实例。这样做确保了每种类型的模板实例在项目中只被编译一次。

C++中类成员中的访问权限和继承权限

cout和printf

printf是函数，cout是std::ostream的全局对象，<<对各种数据类型进行了重载，会自动识别数据类型。

cout是有缓冲输出，即输出字符先放入缓冲区，再输出到屏幕。

printf是行缓冲输出。

重载运算符

允许程序员对自定义类和结构重载运算符，以像基本数据类型那样使用运算符，比如定义一个复数类，通过重载+运算符实现复数运算。

```
class Complex {
public:
    double real, imag;

    // 构造函数
    Complex(double r = 0.0, double i = 0.0) : real(r), imag(i) {}

    // 重载+运算符
    Complex operator+(const Complex& b) const {
        return Complex(real + b.real, imag + b.imag);
    }
};
```

对于函数重载，函数的匹配原则和顺序是什么

1. 精确匹配：参数类型完全一致
2. 提升匹配：比如将char类型参数提升到int
3. 标准转换匹配：比如int转到double，double转float
4. 最佳匹配：多个匹配，会根据上述顺序选择最佳

定义和声明的区别

对于变量，声明就是告诉编译器变量的存在，不分配内存，定义则是为其分配内存，声明可以多次，定义只能一次。

函数的声明一般包括函数的签名，名字、参数列表和返回值。函数的定义包括具体的函数体

```
extern int x; // 声明一个变量，但不定义它
void foo(int); // 函数声明，没有函数体
class Bar; // 类的前向声明
int x; // 变量定义，分配内存
void foo(int a) {
    // 函数定义，具体实现
    cout << a;
}
class Bar {
    // 类定义
    int b;
    void method() { cout << b; }
};
```

全局变量和static变量

全局变量在整个程序都可见，static变量作用域限定为该文件，两者都在main启动前初始化。

静态成员和普通成员

普通成员是每个对象都有，单独存储，只能通过类的对象访问

静态成员是整个类共享，可以通过类名访问，也可以通过类的对象访问

ifndef endif

如果没定义则定义，避免重复定义

隐式转换，如何消除隐式转换

隐式转换是编译器自动进行的转换，比如将int类型赋值给float类型

在函数构造时加上explicit禁止隐式转换

C++如何处理多个异常

try throw catch (多个catch块)

不使用额外空间交换两个数

1) 算术

```
x = x + y;  
y = x - y;  
  
x = x - y;
```

2) 异或

```
x = x^y; // 只能对int,char..  
y = x^y;  
x = x^y;  
x ^= y ^= x;
```

strcpy和memcpy的区别

```
char *strcpy(char *dest, const char *src)  
void *memcpy(void *str1, const void *str2, size_t n)
```

strcpy只能复制字符串，不用指定长度，会在遇到'\0'停止

memcpy可以复制任何内容，需要指定长度

```
int main(int argc, char* argv[])
```

argc: 参数个数

argv: 存储参数, 第一个参数为程序名称

如果有一个空类, 它会默认添加哪些函数

1. 默认构造函数
2. 拷贝构造函数
3. 赋值运算符
4. 析构函数

const char*和string

string: c++标准库、封装了对字符串的操作, 是类, 长度自动扩展

char*: c风格字符串, 以'\0'结尾, 长度限定

a) string转const char*

```
string s = "abc";  
  
const char* c_s = s.c_str();
```

b) const char* 转string, 直接赋值即可

```
const char* c_s = "abc";  
string s(c_s);
```

c) string 转char*

```
string s = "abc";  
char* c;  
const int len = s.length();  
c = new char[len+1];  
strcpy(c, s.c_str());
```

d) char* 转string

```
char* c = "abc";  
string s(c);
```

e) const char* 转char*

```
const char* cpc = "abc";  
char* pc = new char[strlen(cpc)+1];  
strcpy(pc, cpc);
```

f) char* 转const char*, 直接赋值即可

```
char* pc = "abc";  
const char* cpc = pc;
```

什么时候使用指针作为参数，什么时候使用引用

指针：

1. 基础类型数组
2. 可选参数，传递nullptr

引用：

1. 类
2. 确保参数非空

如何设计一个计算仅单个子类的对象的个数

类中使用static变量作为计数并初始化（未显式初始化默认为0）。

构造函数、拷贝构造函数、赋值构造函数对count+1，析构函数-1

将引用作为返回值

最大的好处是内存中不产生被返回值的副本

不能返回局部变量的引用

不能返回new分配的内存的引用

可以返回类的引用

sprintf

sprintf用于格式化数据写入字符串，是printf的变体，printf是将格式化的输出到标准输出。

```
int sprintf(char *str, const char *format, ...);  
#include <stdio.h>  
  
int main() {  
    char buffer[100];  
    int num = 250;  
    float pi = 3.14159;  
  
    sprintf(buffer, "Number: %d, Pi: %.2f", num, pi);  
    printf("%s\n", buffer); // 输出: Number: 250, Pi: 3.14  
  
    return 0;  
}
```

如何禁止程序自动生成拷贝构造函数

将拷贝构造函数标记为delete来明确禁止生成拷贝构造函数。

main的返回值

0传递给调用者表示程序正常退出

static_cast优点

1. 更安全，不允许逻辑上的大跨步，比如将整数直接转换为指针，在编译时会检查
2. 可读性更高，类型转换更明确

回调函数

发生某种事件时，系统或其它函数会自动调用这段函数。

声明

定义

设置触发条件

将函数指针作为参数传入

友元函数和友元类

将一个函数或一个类声明为某个类的“友元”，可以允许这些函数或类访问当前类的private和protected成员。

优势：

1. 灵活
2. 封装
3. 友元关系不能被继承、是单向的、不能有传递性

```
class Box {
public:
    Box(double w, double h, double d) : width(w), height(h), depth(d) {}

    // 声明一个非成员函数为友元
    friend double calculateVolume(const Box& b);

private:
    double width, height, depth;
};

double calculateVolume(const Box& b) {
    return b.width * b.height * b.depth;
}

class Engine;

class Car {
public:
    Car() : engine(nullptr) {}

    void setEngine(Engine* eng) {
        engine = eng;
    }

    // 声明另一个类为友元
    friend class Mechanic;
```

```

private:
    Engine* engine;
};

class Mechanic {
public:
    void repairEngine(Car& car) {
        // 访问 Car 类的私有成员
        car.engine = new Engine();
        // 可以执行更多与引擎相关的操作
    }
};

```

C语言模拟C++的继承

用结构体和函数指针

静态编译和动态编译

静态编译：依赖的库在编译中进行链接，运行时不依赖动态链接库

动态编译：运行时加载动态链接库，所有用到该库的程序都共享这个文件

介绍一下几种锁

读写锁：

1. 多个读者读
2. 一个写者写
3. 读优先锁、写优先锁

互斥锁：

1. 只有一个线程能拥有锁，等待的线程进入睡眠直到锁的状态改变被唤醒，也可能是自旋一段时间，超过阈值再睡眠，因为会涉及CPU的上下文切换

条件变量：

1. 只有锁定和非锁定两种状态
2. 与互斥锁使用
3. 锁状态变化通过条件变量唤醒一个或多个等待的线程，是同步机制

自旋锁：

1. 拿不到锁就一直尝试获取CPU直到获取为止，加锁时间比较短的场景效率比较高

内联函数

只适合简单的代码比较少的函数

虚函数表和虚函数指针

虚函数表和虚函数指针都是实现多态的关键。

1. 虚函数表：某个类包含至少一个虚函数时，编译器会为这个类生成虚函数表，这个表包含指向类的虚函数指针的数组

2. 虚函数指针：指向与对象类型相对应的虚函数表，当通过基类指针调用虚函数时，实际调用的函数通过虚函数指针找到虚函数表中对应的函数，实现动态绑定。

基础语法

= 和 := 的区别

=是赋值变量， :=是定义并赋值变量。

指针的作用

指向任意变量的地址，它所指向的地址在32位或64位机器上分别固定占4或8个字节。

- 获取变量的值 (*ptr)
- 改变变量的值
- 使用指针代替值作为方法接收器（指定了方法绑定到的类型）
 - **值接收器** 使用结构体类型的副本调用方法。这意味着**对接收器的任何修改都不会影响原始结构体实例**。
 - **指针接收器** 使用指向结构体的指针，这样方法中**对接收器的任何修改都会反映到原始结构体实例上**。

Go 允许多个返回值吗

可以。通常函数除了一般返回值还会返回一个error。

Go 有异常类型吗？

Go使用 error 类型来处理错误，这与其他语言中的异常处理（如Java的 try...catch）不同。Go的错误处理倾向于显式地检查错误，这样做可以增加代码的可读性和控制性，同时避免了异常处理可能引入的复杂性和性能开销。

可以通过 errors.New()，用于创建一个简单的错误实例。

什么是协程 (Goroutine)

协程是**用户态轻量级线程**，由调度器对G、M、P进行调度实现高并发。通常在函数前加上go关键字就能实现并发。一个Goroutine会以一个很小的栈启动2KB或4KB，当遇到栈空间不足时，栈会**自动伸缩**，因此可以轻易实现成千上万个goroutine同时启动。

如何高效地拼接字符串

字符串拼接方法有： + , fmt.Sprintf , strings.Builder , strings.Join , bytes.Buffer ,

1 "+"

使用 + 操作符进行拼接时，**会对字符串进行遍历，计算并开辟一个新的空间来存储原来的两个字符串**。

2 fmt.Sprintf

由于采用了接口参数，必须要用反射获取值，因此有性能损耗。

3 strings.Builder.WriteString() + String()

内部实现是指针+切片，同时String()返回拼接后的字符串，它是直接把[]byte转换为string，从而避免变量拷贝。

4 strings.join

`strings.join`也是基于`strings.builder`来实现的，并且可以自定义分隔符，会根据传入切片长度提前进行容量分配可以减少内存分配，很高效。

5 bytes.Buffer+Write()

`bytes.buffer`底层也是一个`[]byte`切片。

性能：`strings.Join`≈`strings.Builder`>`bytes.Buffer`>"+">>`fmt.Sprintf`

```
func main(){
    a := []string{"a", "b", "c"}
    //方式1: +
    ret := a[0] + a[1] + a[2]
    //方式2: fmt.Sprintf
    ret := fmt.Sprintf("%s%s%s", a[0], a[1], a[2])
    //方式3: strings.Builder
    var sb strings.Builder
    sb.WriteString(a[0])
    sb.WriteString(a[1])
    sb.WriteString(a[2])
    ret := sb.String()
    //方式4: bytes.Buffer
    buf := new(bytes.Buffer)
    buf.Write(a[0])
    buf.Write(a[1])
    buf.Write(a[2])
    ret := buf.String()
    //方式5: strings.Join
    ret := strings.Join(a, "")
}
```

什么是 rune 类型

rune是go中int32的别名，几乎在所有方面等同于int32，用来区分字符值和整数值。常规的英文字符是ascii码是通过一个字节(2^8 其实还有一位是不用的)来存储，中国文字、日本文字常用文字就有4000+，通过 2^8 肯定表达不了，于是有了Unicode字符。go语言的编码是按照UTF-8编码规则，UTF使用1至4个字节来表示一个字符，所以在go语言中引进了rune的概念。在我们对字符串进去处理的时候只需要将字符串通过range去遍历，会按照rune为单位自动去处理，极其便利。

如何判断 map 中是否包含某个 key

将map[key]赋值给匿名变量和ok，通过ok的值来判断。

```
var sample map[int]int
if _, ok := sample[10]; ok {
} else {
}
```

Go 支持默认参数或可选参数吗

不支持，支持使用参数切片数组实现可变参数。

```
// 这个函数可以传入任意数量的整型参数
func sum(nums ...int) {
    total := 0
    for _, num := range nums {
        total += num
    }
    fmt.Println(total)
}
```

defer 的执行顺序

defer在return之后，但在函数退出之前执行。多个defer语句会被添加到栈中，也就是后进先出(LIFO)，后面写的defer语句会先执行。

```
func test() int {
    i := 0
    defer func() {
        fmt.Println("defer1")
    }()
    defer func() {
        i += 1
        fmt.Println("defer2")
    }()
    return i
}

func main() {
    fmt.Println("return", test())
}
// defer2
// defer1
// return 0
// 对于无名返回，go会创建一个临时变量保存返回值，由于defer在return后执行，因此是0
```

对于有名返回，defer是有可能改变其值的。对于有名返回值，不会为其创建临时变量保存。

```
func test(i int) {
    i = 0
    defer func() {
        i += 1
        fmt.Println("defer2")
    }()
    return i
}

func main() {
    fmt.Println("return", test())
}
// defer2
// return 1
```

如何交换 2 个变量的值

对于变量而言 `a, b = b, a`； 对于指针而言 `*a, *b = *b, *a`

Go 语言 tag 的用处

tag可以为结构体成员提供属性。常见的：

- json/toml序列化或反序列化时字段的名称
- db: sqlx模块中对应的数据库字段名

如何获取一个结构体的所有tag

使用反射

```
import reflect
type Author struct {
    Name      int      `json:Name`
    Publications []string `json:Publication,omitempty`
}

func main() {
    t := reflect.TypeOf(Author{})
    for i := 0; i < t.NumField(); i++ {
        name := t.Field(i).Name
        s, _ := t.FieldByName(name)
        fmt.Println(name, s.Tag)
    }
}
```

`reflect.TypeOf`方法获取对象的类型，之后`NumField()`获取结构体成员的数量。通过`Field(i)`获取第*i*个成员的名字。再通过其`Tag`方法获得标签。

如何判断 2 个字符串切片 (slice) 是相等的

1. 自己手写遍历比较
2. 使用反射，`reflect.DeepEqual()`，但反射非常影响性能，并且会比较切片的底层数组是否相同，可能影响结果。

结构体打印时，`%v` 和 `%+v` 的区别

下面是一个示例代码，演示了使用`%v`、`%+v`和`%#v`格式化输出结构体成员的值、名称和值的方法：

```
package main

import "fmt"

type Person struct {
    Name    string
    Age     int
    Country string
}

func main() {
    p := Person{
```

```

        Name: "Alice",
        Age: 30,
        Country: "USA",
    }

    // %v 输出结构体各成员的值
    fmt.Printf("Using %%v: %v\n", p)

    // %+v 输出结构体各成员的名称和值
    fmt.Printf("Using %%+v: %+v\n", p)

    // %#v 输出结构体名称和结构体各成员的名称和值
    fmt.Printf("Using %%#v: %#v\n", p)
}

```

在上面的示例中，结构体 `Person` 有三个成员：`Name`、`Age` 和 `Country`。我们创建了一个 `Person` 类型的实例 `p`，并使用 `%v`、`%+v` 和 `%#v` 格式化输出不同类型的信息。

运行这段代码后，你会看到类似下面的输出：

```

Using %v: {Alice 30 USA}
Using %+v: {Name:Alice Age:30 Country:USA}
Using %#v: main.Person{Name:"Alice", Age:30, Country:"USA"}

```

- `%v` 输出了结构体 `Person` 各成员的值
- `%+v` 输出了结构体各成员的名称和值
- `%#v` 输出了结构体名称和结构体各成员的名称和值。

Go 语言中如何表示枚举值(enums)

在 Go 语言中，没有内置的枚举类型，但是可以使用 `const` 常量和 `iota`（从 0 开始）来模拟枚举值。

```

// 定义一个枚举类型
type Color int

const (
    Red Color = iota
    Green
    Blue
)

```

空 struct{} 的用途

- 用 map 模拟 set，就要把值置为 `struct{}`

```

type Set map[string]struct{}

func main() {
    set := make(Set)

    for _, item := range []string{"A", "A", "B", "C"} {
        set[item] = struct{}{}
    }
    fmt.Println(len(set)) // 3
    if _, ok := set["A"]; ok {
        fmt.Println("A exists") // A exists
    }
}

```

- 仅有方法的结构体
- 空结构体可以用作通道的元素，用于实现某些特殊的通信模式，例如只关注通道是否关闭而不传输任何数据。**

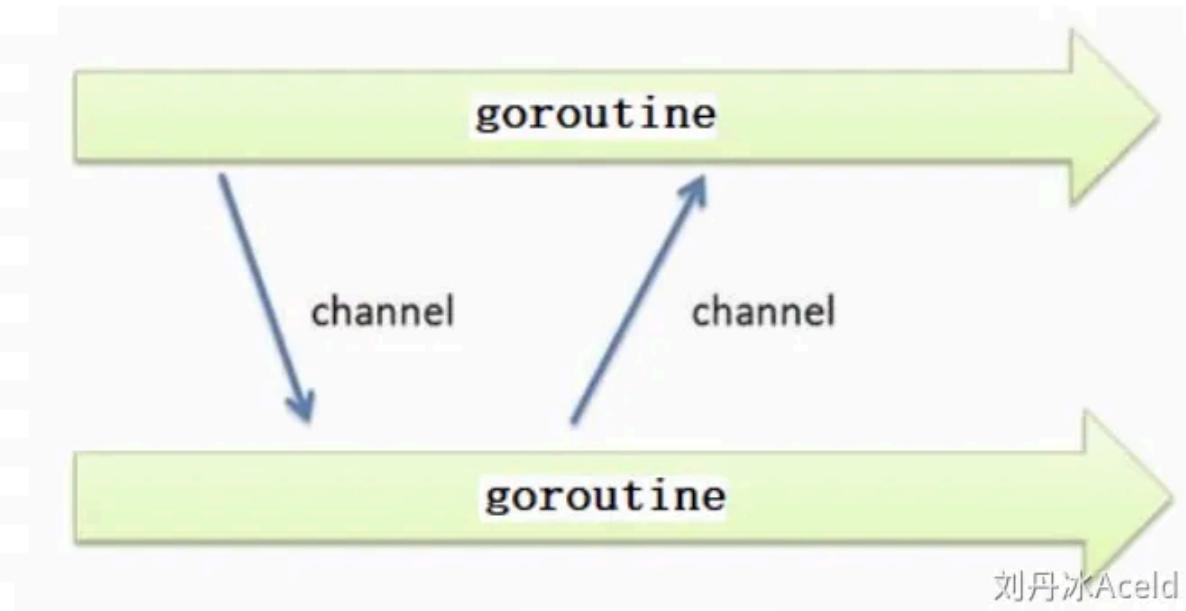
go里面的int和int32是同一个概念吗

go语言中的int的大小是和操作系统位数相关的，如果是32位操作系统，int类型的大小就是4字节。如果是64位操作系统，int类型的大小就是8个字节。除此之外uint也与操作系统有关。

int8占1个字节，int16占2个字节，int32占4个字节，int64占8个字节。

channel

channel是一个数据类型，可以看作管道，主要用来解决go协程的同步问题以及go协程之间数据共享（数据传递）的问题。



创建channel: **chan**是创建channel所需使用的关键字。Type 代表指定channel收发数据的类型。

```

make(chan Type) //等价于make(chan Type, 0)
make(chan Type, capacity)

```

- 当参数capacity= 0 时，channel 是无缓冲阻塞读写的
- 当capacity > 0 时，channel 有缓冲、是非阻塞的，直到写满 capacity个元素才阻塞写入。

channel接收和发送数据：

```
channel <- value      //发送value到channel
<-channel           //接收并将其丢弃
x := <-channel       //从channel中接收数据，并赋值给x
x, ok := <-channel   //功能同上，同时检查通道是否已关闭或者是否为空
```

默认情况下，channel接收和发送数据都是阻塞等待的，除非另一端已经准备好，这样就使得goroutine同步变的更加的简单，而不需要显式的lock和条件变量。

无缓冲的channel：在接收前没有能力保存任何数据值的通道。这种类型的通道要求发送goroutine和接收goroutine同时准备好，才能完成发送和接收操作。否则，通道会导致先执行发送或接收操作的goroutine阻塞等待。

有缓冲的channel：一种在被接收前能存储一个或者多个数据值的通道。这种类型的通道并不强制要求goroutine之间必须同时完成发送和接收。通道会阻塞发送和接收动作的条件也不同。只有通道中没有要接收的值时，接收动作才会阻塞。只有通道没有可用缓冲区容纳被发送的值时，发送动作才会阻塞。

可以使用close去关闭channel。

单向channel：单向 channel 可以是只发送或只接收：

- 只发送 channel 的类型为 `chan<- Type`，表示这个 channel 只能用来发送类型为 `Type` 的数据。
- 只接收 channel 的类型为 `<-chan Type`，表示这个 channel 只能用来接收类型为 `Type` 的数据。

```
func produce(ch chan<- int) {
    for i := 0; i < 10; i++ {
        ch <- i // 发送数据
    }
    close(ch)
}

func consume(ch <-chan int) {
    for value := range ch {
        fmt.Println("Received:", value)
    }
}

func main() {
    ch := make(chan int)
    go produce(ch)
    consume(ch)
}
```

实现原理

init() 函数是什么时候执行的

init()在main()之前执行。

init()函数是go初始化的一部分，由runtime初始化每个导入的包，初始化不是按照从上到下的导入顺序，而是按照解析的依赖关系，没有依赖的包最先初始化。

- 每个包首先初始化包作用域的常量和变量（常量优先于变量），然后执行包的 `init()` 函数。
- 同一个包，甚至是同一个源文件可以有多个 `init()` 函数。

- `init()` 函数没有入参和返回值，不能被其他函数调用，同一个包内多个 `init()` 函数的执行顺序不作保证。

执行顺序: `import` -> `const` -> `var` -> `init()` -> `main()`

一个文件可以有多个 `init()` 函数！

如何知道一个对象是分配在栈上还是堆上

编译器自动决定一个对象是分配在栈上还是堆上，这主要基于逃逸分析（escape analysis）。逃逸分析在编译时进行，目的是确定数据分配的位置（栈或堆）以优化内存使用和性能。

栈上分配

- 如果一个对象在函数内部创建，并且它不会在函数外部被引用，那么这个对象通常会被分配在栈上。
- 栈上分配的对象在函数执行完毕后会自动被清理，这种方式的内存分配和回收效率非常高。

堆上分配

- 如果对象在函数外部仍然存在引用，或者因为其大小不确定或太大，它就可能被分配到堆上。
- 堆上的对象需要通过垃圾回收器来管理，这可能会引入额外的性能开销。

总体来说就是一个变量离开其作用域没有被引用，则优先分配给栈，否则分配给堆。但是变量类型大小不确定等情况也可能发生逃逸。

编译时标志：使用 `go build -gcflags='-m'` 命令可以看到编译器的逃逸分析报告，其中会标明哪些对象逃逸到堆上。

2 个 interface 可以比较吗

两个接口（interface）类型的值是可以比较的，但有特定的规则和限制。

1. **相同类型的动态值：**如果两个接口类型的动态值是相同类型，并且这个类型支持比较（例如基本类型如 `int`、`string` 等），那么可以将这两个接口值进行比较。比较的结果基于它们的动态值。
2. **不同类型的动态值：**如果两个接口的动态值类型不同，即使它们的实际值相同，比较的结果也是 `false`。
3. **零接口值：**如果两个接口都是零值（即它们的动态类型和动态值都为 `nil`），它们是相等的。
4. **一个接口值为 nil：**如果一个接口值为 `nil` 而另一个不是，它们不相等。

2 个 nil 可能不相等吗

两个 `nil` 只有在类型相同时才相等，比如一个变量是指针类型的 `nil`，一个是接口类型的 `nil`，那么他们不相等。

Go 语言GC(垃圾回收)的工作原理

1.3 标记清除

1.5 三色标记

1.7 并行栈

1.8 混合写屏障

1.9 彻底移除暂停程序的重新扫描栈

函数返回局部变量的指针是否安全

和c++不一样，在Go里面返回局部变量的指针是安全的。因为Go会进行逃逸分析，如果发现局部变量的作用域超过该函数则会把指针分配到堆区，避免内存泄漏。

非接口的任意类型 T() 都能够调用 *T 的方法吗？反过来呢

如果T是可寻址的，可以调用*T的方法，反过来也可以，因为指针可以解引用。

go slice是怎么扩容的

一个切片在Go中是由三个部分组成的：

1. **指针**：指向底层数组的第一个元素。
2. **长度** (length)：切片中元素的数量。
3. **容量** (capacity)：从切片的起始元素开始，底层数组中可用的元素数量。

扩容的过程大致如下：

1. 新容量的计算：

1. 在Go 1.18之前，切片的扩容策略相对简单：
 - 当切片的容量小于1024时，新的容量通常是旧容量的两倍。
 - 当切片的容量大于或等于1024时，新的容量会增加旧容量的25%（即增长因子为1.25）。
 2. 从Go 1.18开始，扩容策略进行了调整，以更好地适应不同大小的切片：
 - 当切片的容量小于256时，新的容量仍然是旧容量的两倍。
 - 当切片的容量大于或等于256时，新的容量计算方式变为 `newcap = oldcap + (oldcap + 3*256)/4`。这种方法相比以前更加复杂，其设计目的是在保持较快增长的同时，更加平滑地过渡到较大的容量。
 3. 当然如果期望容量大于了当前容量的两倍，则会等于期望容量。
2. **分配新数组**：根据计算出的新容量，分配一个新的底层数组。
 3. **复制旧元素**：将原切片中的元素复制到新的底层数组中。
 4. **更新切片描述符**：更新切片的指针、长度和容量，使其指向新的底层数组。

扩容是昂贵的，如果预先知道切片的大小，应该在make的时候标明。

无缓冲的 channel 和有缓冲的 channel 的区别

对于无缓冲区channel：发送的数据如果没有被接收方接收，那么**发送方阻塞**；如果一直接收不到发送方的数据，**接收方阻塞**；

有缓冲的channel：发送方在缓冲区满的时候阻塞，接收方不阻塞；接收方在缓冲区为空的时候阻塞，发送方不阻塞。

为什么有协程泄露

协程泄漏是指协程创建之后没有得到释放。主要原因有：

1. 下面两个或三个都可以归结为无限期等待某事件
2. 缺少接收器，导致发送阻塞
3. 缺少发送器，导致接收阻塞

4. **死锁**: 多个协程由于竞争资源导致死锁。
5. **资源未释放**: 例如, 协程打开的文件或网络连接没有关闭, 可能导致资源长时间占用。

Go 可以限制运行时操作系统线程的数量吗? 常见的goroutine操作函数有哪些

使用`runtime.GOMAXPROCS(num int)`可以设置线程数目, 但这个数目是同时执行go代码的线程数量, 不包括被阻塞的线程。该值默认为CPU逻辑核数, 如果设的太大, 会引起频繁的线程切换, 降低性能。对于 CPU 密集型的任务, 若该值过大, 例如设置为 CPU 逻辑核数的 2 倍, 会增加线程切换的开销, 降低性能。对于 I/O 密集型应用, 适当地调大该值, 可以提高 I/O 吞吐率。

- `runtime.Gosched()`, 用于让出CPU时间片, 让出当前goroutine的执行权限, 调度器安排其它等待的任务运行, 并在下次某个时候从该位置恢复执行。
- `runtime.Goexit()`, 调用此函数会立即使当前的goroutine的运行终止 (终止协程), 而其它的goroutine并不会受此影响。`runtime.Goexit`在终止当前goroutine前会先执行此goroutine的还未执行的defer语句。请注意千万别在主函数调用`runtime.Goexit`, 因为会引发panic。

如何控制协程数目

可以使用带缓冲区的channel来控制:

```
var wg sync.WaitGroup // 等待一组协程执行完成
ch := make(chan struct{}, 1024)
for i:=0; i<20000; i++{
    wg.Add(1)          // 增加waitGroup的计数器, 表示有一个新的协程需要等待其完成
    ch<-struct{}{}
    go func(){
        defer wg.Done() // 减少waitGroup的计数器
        <-ch
    }
}
wg.Wait()           // 等待所有注册的协程完成
```

还可以用协程池, 原理和上面的一样。

new和make的区别

- new只用于分配内存，返回一个指向地址的**指针**。它为每个新类型分配一片内存，初始化为0且返回类型*T的内存地址，它相当于&T{}
- make**只可用于slice, map, channel的初始化,返回的是引用。**

请你讲一下Go面向对象是如何实现的？

go的面向对象不同于传统的C++和java语言，go没有对象的概念。通过struct和interface来实现面向对象。

1. 封装：

- 如果一个结构体的字段名或者方法名以大写字母开头，那么它是公开的（可以被其他包访问）。
- 如果以小写字母开头，则是私有的（只能在同一包内访问）。

2. 继承：通过嵌入结构体来实现类似继承的功能，或者说组合。

3. 多态：通过interface，interface定义了一组方法签名，任何实现了这些方法的类型都被认为实现了该接口。这样，你可以通过接口类型引用具体的实例，实现运行时的多态性。

uint型变量值分别为 1, 2, 它们相减的结果是多少？

会溢出，从最大可能的数开始回绕。如果是32位系统就是 $2^{32}-1$ ，如果是64位系统，结果 $2^{64}-1$ 。

讲一下go有没有函数在main之前执行？怎么用？

init函数：

- 不能被其他函数调用
- 没有入参和返回
- 程序运行前执行注册、初始化变量等
- 每个源文件可能有多个init函数，执行顺序不确定
- 不同包的init函数按照导入顺序决定执行顺序

下面这句代码是什么作用，为什么要定义一个空值

```
type GobCodec struct{
    conn io.ReadWriteCloser
    buf *bufio.Writer
    dec *gob.Decoder
    enc *gob.Encoder
}

type Codec interface {
    io.Closer
    ReadHeader(*Header) error
    ReadBody(interface{}) error
    Write(*Header, interface{}) error
}

var _ Codec = (*GobCodec)(nil)
```

先把nil转换为*GobCodec类型，然后再转换为Codec接口类型，如果转换失败则说明GobCodec没有实现Codec中的所有方法。

golang的内存管理的原理清楚吗？简述go内存管理机制。

go内存管理本质上是一个内存池，使用了一个“tcmalloc”（Thread-Caching Malloc）风格的内存分配器，但内部做了很多优化：自动伸缩内存池大小，合理的切割内存块。

Page: Go向操作系统申请和释放内存都是以页为单位的。

span: 内存块，一个或多个连续的 page 组成一个 span。

sizeclass: 空间规格，每个 span 都带有一个 sizeclass，标记着该 span 中的 page 应该如何使用。

object: 对象，用来存储一个变量数据内存空间，一个 span 在初始化时，会被切割成一堆等大的 object。所谓内存分配，就是分配一个 object 出去。

该分配器将内存分为多个大小的类别，以适应不同大小的对象：

- 小对象：使用固定大小的内存块进行分配，这些内存块从预分配的页中获得。
- 大对象：直接从堆中分配，绕过固定大小的内存块机制。

每个 Goroutine 拥有自己的缓存（mcache），用于小对象的快速分配，而较大的对象分配则可能需要锁定和更多的协调。

GC：并发，三色标记清楚、混合写屏障。

mutex有几种模式？

在正常模式中，等待者按照 FIFO 的顺序排队获取锁，但是一个被唤醒的等待者有时候并不能获取 mutex，它还需要和新到来的 goroutine 们竞争 mutex 的使用权。新到来的 goroutine 存在一个优势，它们已经在 CPU 上运行且它们数量很多，因此一个被唤醒的等待者有很大的概率获取不到锁，在这种情况下它处在等待队列的前面。如果一个 goroutine 等待 mutex 释放的时间超过 1ms，它就会将 mutex 切换到饥饿模式。公平性：否。

在饥饿模式中，mutex 的所有权直接从解锁的 goroutine 递交到等待队列中排在最前方的 goroutine。新到达的 goroutine 们不要尝试去获取 mutex，即使它看起来是在解锁状态，也不要试图自旋，而是排到等待队列的尾部。公平性：是。

正常模式下的性能会更好，因为一个 goroutine 能在即使有很多阻塞的等待者时多次连续的获得一个 mutex，饥饿模式的重要性则在于避免了病态情况下的尾部延迟。

go如何进行调度的。GMP中状态流转。

GMP。

go进行调度过程：

- 某个线程尝试创建一个新的G，那么这个G就会被安排到这个线程的G本地队列LRQ中，如果LRQ满了，就会分配到全局队列GRQ中；
- 尝试获取当前线程的M，如果无法获取，就会从空闲的M列表中找一个，如果空闲列表也没有，那么就创建一个M，然后绑定M与P运行。
- 进入调度循环：
 - G进行调度
 - 找到一个合适的G
 - 执行G，完成以后退出

Go什么时候发生阻塞？阻塞时，调度器会怎么做

比如系统调用，channel阻塞。

如果是系统调用阻塞，G和M会脱离P，然后P去找一个空闲的M进行绑定，执行P列表中的G或者全局列表的G或者work stealing。

如果是channel阻塞，G会脱离M，然后重新调度新的G放入M。

Go中GMP有哪些状态

G的状态：可简化为等待中（系统调用、阻塞），可运行，运行中，暂停（GC）

- **_Gidle**：刚刚被分配并且还没有被初始化，值为0，为创建goroutine后的默认值
- **_Grunnable**：没有执行代码，没有栈的所有权，存储在运行队列中，可能在某个P的本地队列或全局队列中(如上图)。
- **_Grunning**：正在执行代码的goroutine，拥有栈的所有权(如上图)。
- **_Gsystcall**：正在执行系统调用，拥有栈的所有权，与P脱离，但是与某个M绑定，会在调用结束后被分配到运行队列(如上图)。
- **_Gwaiting**：被阻塞的goroutine，阻塞在某个channel的发送或者接收队列(如上图)。
- **_Gdead**：当前goroutine未被使用，没有执行代码，可能有分配的栈，分布在空闲列表gFree，可能是一个刚刚初始化的goroutine，也可能是执行了goexit退出的goroutine(如上图)。
- **_Gcopystac**：栈正在被拷贝，没有执行代码，不在运行队列上，执行权在
- **_Gscan**：GC正在扫描栈空间，没有执行代码，可以与其他状态同时存在。

P的状态：空闲，运行中，处于GC被停止，不在使用

- **_Pidle**：处理器没有运行用户代码或者调度器，被空闲队列或者改变其状态的结构持有，运行队列为空
- **_Prunning**：被线程 M 持有，并且正在执行用户代码或者调度器(如上图)
- **_Psystcall**：没有执行用户代码，当前线程陷入系统调用(如上图)
- **_Pgcstop**：被线程 M 持有，当前处理器由于垃圾回收被停止
- **_Pdead**：当前处理器已经不被使用

M的状态：

- **自旋线程**：处于运行状态但是没有可执行goroutine的线程，数量最多为GOMAXPROC，若是数量大于GOMAXPROC就会进入休眠。
- **非自旋线程**：处于运行状态有可执行goroutine的线程。

GMP能不能去掉P层？会怎么样？

不能，如果没有P，那么会依赖全局队列，就会开销很大，比如锁和同步，P的队列是CAS的。此外，如果直接把G放到M，那么要是阻塞，那其他G也不能执行了，就浪费CPU资源了。

如果有任何一个G一直占用资源怎么办？什么是work stealing算法

GMP调度器采用基于协作的抢占机制，每个G执行10ms，就换其他G执行；但也可能出现其它情况，比如长时间的for循环或垃圾回收时间太长，GO1.14后引入了基于信号的抢占和基于函数调用的抢占机制
work stealing就是假如有个线程空闲，并且持有P，那么就会去全局队列中拿G，大小是队列的一半和队列/GOMAXPROC +1的小的那个。如果没有则会去其他P队列中偷G过来，一般取后面一半。

goroutine什么情况会发生内存泄漏？如何避免。

1. 长生命周期的 Goroutine 未能正确退出：当 Goroutine 因为阻塞在通道操作、锁等待或其他同步原语上而无法退出时，它们可能会持续占用内存。
2. Goroutine 持有的资源未被释放：如果 Goroutine 持有对某些资源的引用（如大型数据结构、文件句柄等），并且这些资源在 Goroutine 结束前未被释放或解引用，那么这些资源可能永远不会被垃圾回收。
3. 使用不当的定时器和 Ticker：
如果使用 `time.After` 或 `time.Tick` 并且返回的通道没有被适当处理，可能会导致 Goroutine 泄漏。

如何避免：

1. 确保 Goroutine 可以正确退出
2. 合理管理 Goroutine 的生命周期
3. 避免 Goroutine 对资源的无限制持有
4. 正确使用定时器和 Ticker
5. 利用工具进行监控和检测：使用 Go 的 pprof 工具来监控 Goroutine 的数量和状态，帮助识别潜在的内存泄漏。

Go GC有几个阶段

目前的go GC采用三色标记法和混合写屏障技术。

Go GC有四个阶段：

- STW，开启混合写屏障，扫描栈对象；
- 将所有对象加入白色集合，从根对象开始，将其放入灰色集合。每次从灰色集合取出一个对象标记为黑色，然后遍历其子对象，标记为灰色，放入灰色集合；
- 如此循环直到灰色集合为空。剩余的白色对象就是需要清理的对象。
- STW，关闭混合写屏障；
- 在后台进行GC（并发）。

go竞态条件了解吗

所谓竞态竞争，就是当两个或以上的goroutine访问相同资源时候，对资源进行读/写。比如 `var a int = 0`，有两个协程分别对`a+=1`，我们发现最后a不一定为2.这就是竞态竞争。

我们可以用 `go run -race xx.go` 来进行检测。通过对临界区加锁或原子操作来解决。

如果若干个goroutine，有一个panic会怎么做

有一个panic，那么剩余goroutine也会退出，程序退出。如果不希望程序退出，那么必须通过调用 `recover()` 方法来捕获 panic 并恢复将要崩掉的程序。

defer可以捕获goroutine的子goroutine吗？

每个 Goroutine 拥有自己的执行堆栈，而 `defer` 只能作用于它被声明的同一函数或方法中。这意味着 `defer` 不能跨 Goroutine 工作，也无法影响或处理从一个 Goroutine 内部启动的其他 Goroutine。

channel 死锁的场景

channel中没数据去读

channel中数据满了去写

往关闭的channel中写数据

解决办法通过select

对已经关闭的chan进行读写会怎么样？

写会panic

读已经关闭的chan能一直读到东西，但是读到的内容根据通道内关闭前是否有元素而不同。

- 如果chan关闭前，buffer内有元素还未读，会正确读到chan内的值，且返回的第二个bool值（是否读成功）为true。
- 如果chan关闭前，buffer内有元素已经被读完，chan内无值，接下来所有接收的值都会非阻塞直接成功，返回 channel 元素的零值，但是第二个bool值一直为false。

channel底层实现？是否线程安全。

channel内部是一个循环链表。内部包含buf, sendx, recvx, lock, recvq, sendq几个部分。

buf是有缓冲的channel所特有的结构，用来存储缓存数据，是一个循环数组。

- sendx和recvx：这两个索引用于管理 buf 数组中的位置。`sendx` 指向下一个数据应该存放的位置，而 `recvx` 指向下一个数据应该被接收的位置。
- lock是个互斥锁；
- recvq和sendq分别是这些是等待接收和等待发送的 goroutine 队列。当一个 goroutine 尝试从空的非缓冲 channel 接收数据，或者尝试向满的非缓冲 channel 发送数据时，它将会被放入相应的队列中阻塞等待。

channel是线程安全的。

map的底层实现

map 是一种基于哈希表的关联数组实现，用于存储键值对。它的核心组件包括：

1. **哈希函数**：用于将键转换成数组索引，确保键值对分布均匀。
2. **数组**：这是哈希表的主体，存储指向键值对的桶（bucket）。
3. **桶 (bucket)**：每个桶可以存储多个键值对，以处理哈希冲突。当多个键经过哈希函数处理后落在同一个桶中时，通过链表或者开放寻址法解决冲突。
4. **扩容机制**：当 map 中的元素数量达到一定阈值时，系统会自动进行扩容操作，即创建一个更大的哈希表并重新分配现有的元素，以保持操作的效率。

Go 的 `map` 是非线程安全的，如果在多个 goroutine 中并发读写，需要使用锁（如 `sync.Mutex` 或 `sync.RWMutex`）来保证安全。

select的实现原理

在 Go 语言中，`select` 语句允许同时处理多个通道（channel）的发送和接收操作。其工作原理如下：

1. **检查通道状态**：`select` 检查每个 case 语句中的通道是否准备好进行操作（发送或接收）。
2. **随机选择**：如果多个通道同时准备好，`select` 会随机选择一个执行，以保证公平性。

3. **阻塞行为**: 如果没有通道准备好, `select` 将阻塞当前 goroutine, 直到至少一个通道可用。

这种机制使得 `select` 成为处理并发通道操作的有效工具, 特别是在需要对多个通道进行监听和响应时。

go的interface怎么实现的

`interface` 规定了一组方法签名, 但不实现这些方法。任何实现了这些方法的具体类型都可以被视为该接口类型的实例。Go 的接口提供了一种方式来实现多态性, 即不同类型的对象可以以相同的方式被处理。

具体实现是通过一个类型指针和数据指针:

1. **类型指针 (type)** : 这是一个指向表示接口具体类型的数据结构的指针。这个数据结构包含了类型的元数据 (如类型的名称和包路径) 以及与类型相关的方法集合。
2. **数据指针 (data)** : 这是一个指向具体数据的指针。对于值类型 (如结构体), 它直接指向值的内存地址; 对于引用类型 (如指针、切片、映射等), 它指向实际数据结构的指针。

go的reflect 底层实现

`reflect.Type` 和 `reflect.Value`。

1. **reflect.Type**: 提供了关于 Go 值的类型信息, 包括类型的名称、种类 (如 int、slice、struct 等) 和可以执行的操作。这个接口通过内部的类型描述符来实现, 这些描述符包含了类型的详细元数据。
2. **reflect.Value**: 代表一个 Go 值的运行时表示。它持有一个值的实际数据, 并可以用来修改值、获取值或调用关联的方法。`reflect.Value` 通过内部存储实际的值 (使用空接口 `interface{}`) 和类型信息来实现其功能。

go的调试/分析工具用过哪些。

go的自带工具链相当丰富,

- go cover : 测试代码覆盖率;
- godoc: 用于生成go文档;
- pprof: 用于性能调优, 针对cpu, 内存和并发;
- race: 用于竞争检测;

进程被kill, 如何保证所有goroutine顺利退出

在Go语言中, 确保一个进程被kill时所有goroutine顺利退出, 需要设计一种优雅的退出机制。这通常涉及到信号处理和goroutine之间的协调。以下是一种实现方法:

1. **信号监听**: 首先, 需要监听操作系统发送的终止信号 (如SIGINT或SIGTERM)。可以使用 `os/signal` 包来监听系统信号。
2. **同步机制**: 使用 `sync.WaitGroup` 或类似机制来跟踪所有正在运行的goroutine。每个goroutine在开始时调用 `WaitGroup.Add(1)`, 完成时调用 `WaitGroup.Done()`。
3. **通知机制**: 使用 `context.Context` 来传递取消信号给所有goroutine。当接收到终止信号时, 可以通过调用 `cancel()` 函数来广播取消事件。

具体实现步骤如下:

```
package main
```

```
import (
    "context"
    "fmt"
    "os"
    "os/signal"
    "sync"
    "syscall"
    "time"
)

func main() {
    // 创建一个监听系统终止信号的channel
    sigs := make(chan os.Signal, 1)
    // 注册要接收的信号, syscall.SIGINT是Ctrl+C, syscall.SIGTERM是kill发送的默认信号
    signal.Notify(sigs, syscall.SIGINT, syscall.SIGTERM)

    // 创建一个context, 用来通知goroutine退出
    ctx, cancel := context.WithCancel(context.Background())
    defer cancel() // 确保所有路径上都调用cancel

    // 使用WaitGroup等待所有goroutine完成
    var wg sync.WaitGroup

    // 启动goroutine
    wg.Add(1)
    go func() {
        defer wg.Done()
        dowork(ctx)
    }()

    // 等待信号
    sig := <-sigs
    fmt.Printf("Received signal: %s\n", sig)
    cancel() // 收到信号后取消context

    // 等待所有goroutine清理完毕
    wg.Wait()
    fmt.Println("All goroutines have finished.")
}

func dowork(ctx context.Context) {
    for {
        select {
        case <-ctx.Done():
            fmt.Println("Goroutine exiting...")
            return
        default:
            // 模拟一些工作
            fmt.Println("Goroutine working...")
            time.Sleep(1 * time.Second)
        }
    }
}
```

在这个例子中：

- 当接收到SIGINT或SIGTERM信号时，`main`函数中的`cancel()`被调用，这会使得通过`ctx`传递给`dowork`函数的`ctx.Done()`通道被关闭。
- `dowork`函数中的`select`语句会响应`ctx.Done()`的关闭，从而退出循环并返回，这样goroutine就优雅地结束了。
- `main`函数中的`wg.Wait()`确保主程序等待所有goroutine完成后才继续向下执行，最终输出“All goroutines have finished.”表示所有goroutine已经正确清理并退出。

这种方法可以确保在程序需要强制退出时，所有的资源都能被正确释放，goroutine也能有序地关闭。

说说context包的作用？

Go语言的`context`包主要用于管理和传递取消信号、超时通知和其他请求范围的值。其主要用途包括：

- 取消信号**：在多个goroutine之间传递取消信号，用于停止操作和释放资源。
- 超时控制**：设定操作的超时时间，超时后自动发送取消信号。
- 传递请求相关的数据**：在API的调用链中传递请求特定的数据，如用户身份信息、追踪ID等。

defer、panic和recover

- defer**：预定一个函数调用，这个函数会在包含他的函数结束时执行，不管是正常退出还是panic，常用于资源清理，比如关闭文件和数据库连接。如果一个函数中有多个`defer`，执行顺序为后进先出（栈，以及资源依赖）。
- panic**：触发一个运行时错误，会立刻停止当前函数的运行，逐层向上执行`defer`语句，除非被`recover`捕获，用于处理不可恢复的错误。
- recover**：用于捕获或恢复一个panic，只有在`defer`中调用`panic`才有用，会终止panic的连锁反应，返回panic时传递的错误值。

```
package main

import "fmt"

func riskyFunction() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Recovered from panic:", r)
        }
    }()
    fmt.Println("Performing some task")
    if true {
        panic("something bad happened")
    }
    fmt.Println("This will not be printed")
}

func main() {
    riskyFunction()
    fmt.Println("Program continues after recovery")
}
```

go中的slice和数组有什么区别，slice的底层是什么

1. 数组大小在声明时就确定了，并且不能改变
2. 切片可以根据需要动态扩展或收缩
3. 数组是值类型，赋值给新变量，会进行整个数组的改变；切片是引用类型，赋值时不会拷贝数据本身，而是拷贝对底层数据的引用

切片的底层是一个数组，切片结构体包括三个元素：

1. 指针：指向底层数组的第一个元素的位置
2. 容量：当前底层数组所能包含的最大元素数量
3. 长度：当前底层数组中的元素数量

切片的动态扩展是通过创建一个更大的数组并复制现有元素来实现的。这个过程通常由 `append` 函数自动管理。当切片的元素超过其容量时，Go运行时会分配一个新的数组，一般是当前容量的两倍，然后将原有数据复制到新数组中，从而实现切片的扩容。

```
package main

import "fmt"

func main() {
    arr := [5]int{1, 2, 3, 4, 5} // 数组
    s1c := []int{1, 2, 3, 4, 5} // 切片

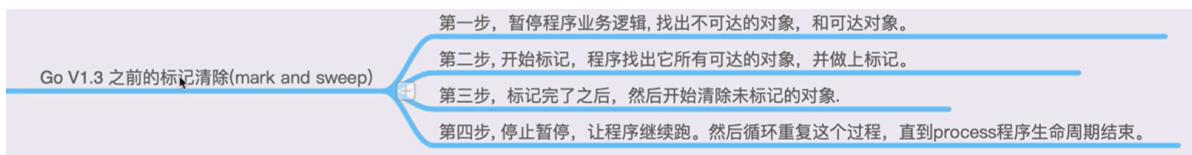
    arrs1c := arr[:] // 从数组创建切片
    s1c2 := s1c[:] // 创建切片的另一个视图

    fmt.Println(arr) // 输出数组
    fmt.Println(s1c) // 输出切片
    fmt.Println(arrs1c) // 输出从数组创建的切片
    fmt.Println(s1c2) // 输出切片的另一个视图
}
```

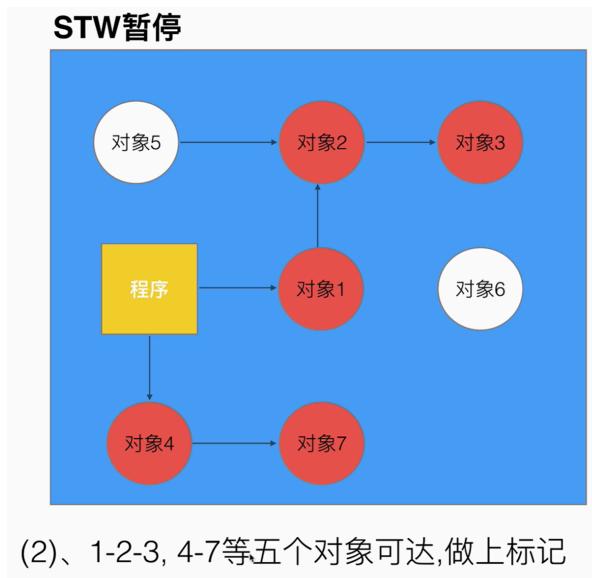
Golang GC

Mark and Sweep

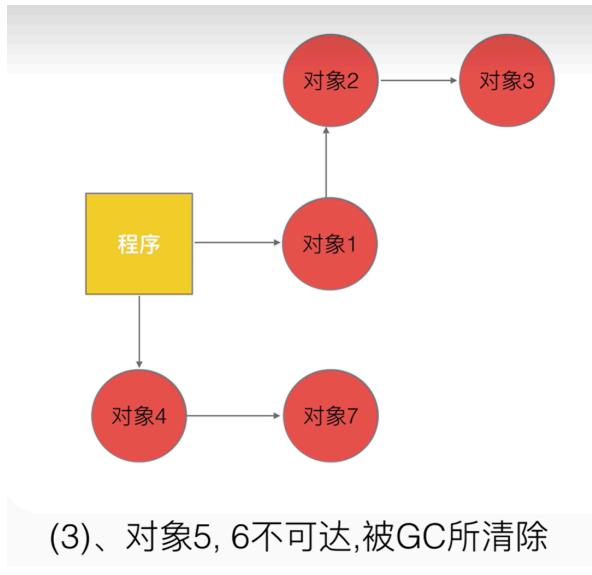
早期的golang使用最简单的GC算法，即mark and sweep。



首先STW，然后从根集合开始，访问并标记堆中可达的对象。



标记完成后，再遍历堆中的全部对象，并将未标记的对象进行垃圾回收，并将内存进行释放加入空闲链表。完成后再停止STW，继续工作。



mark and sweep的缺点：



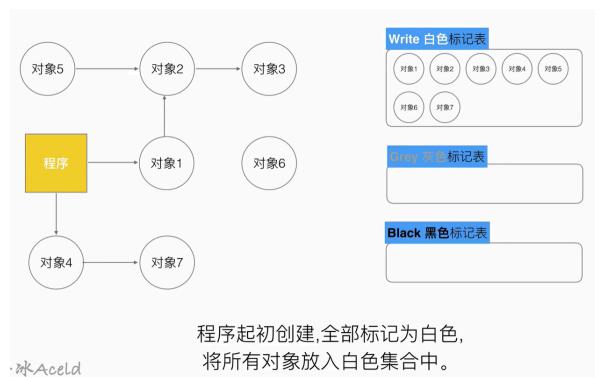
早期的尝试：将sweep放到暂停STW后，但STW的时间依然很长。



三色标记法 (v1.5)

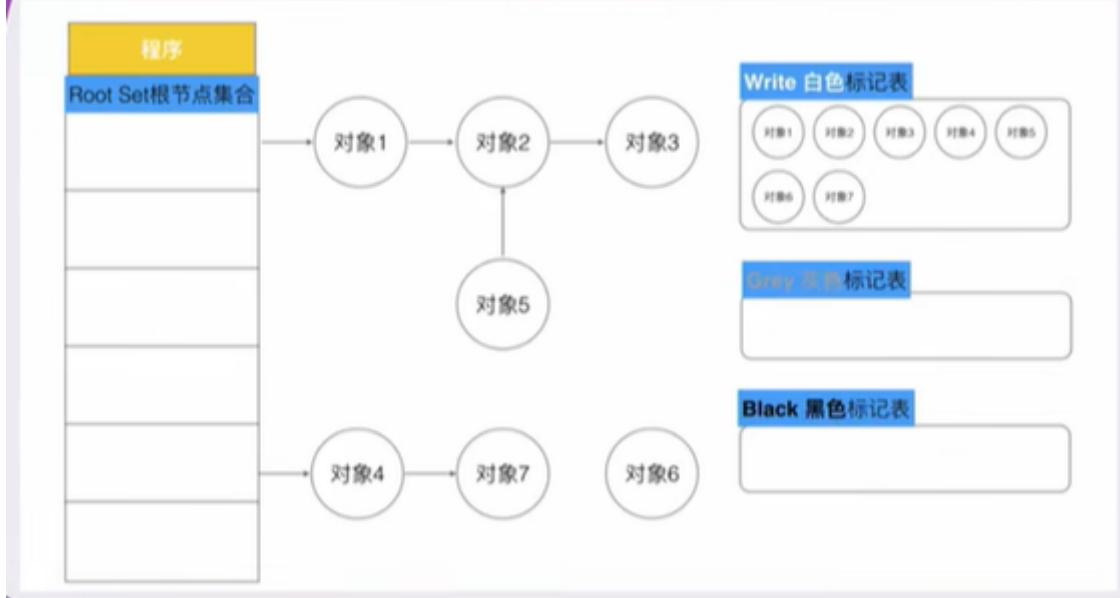
三色标记法将对象分为白色、灰色和黑色：

- 白色：尚未被访问的对象。
- 灰色：已被访问但其引用的对象尚未全部访问的对象。
- 黑色：已被访问，且其引用的对象也被访问的对象。



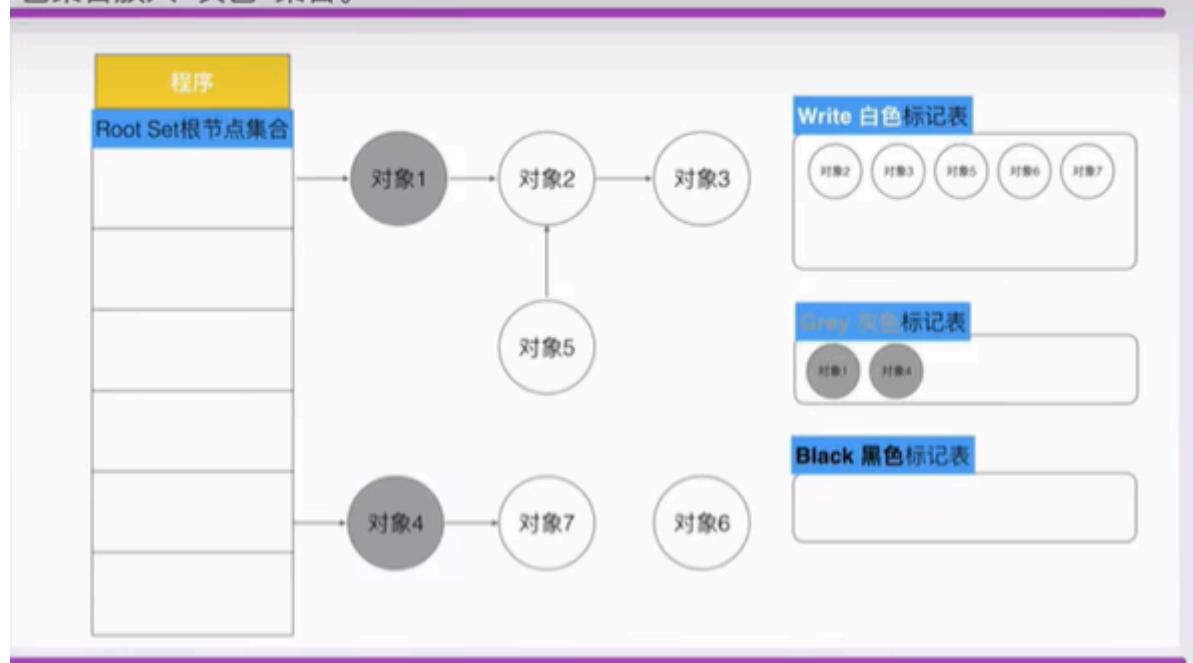
第一步

第一步，就是只要是新创建的对象，默认的颜色都是标记为“白色”。



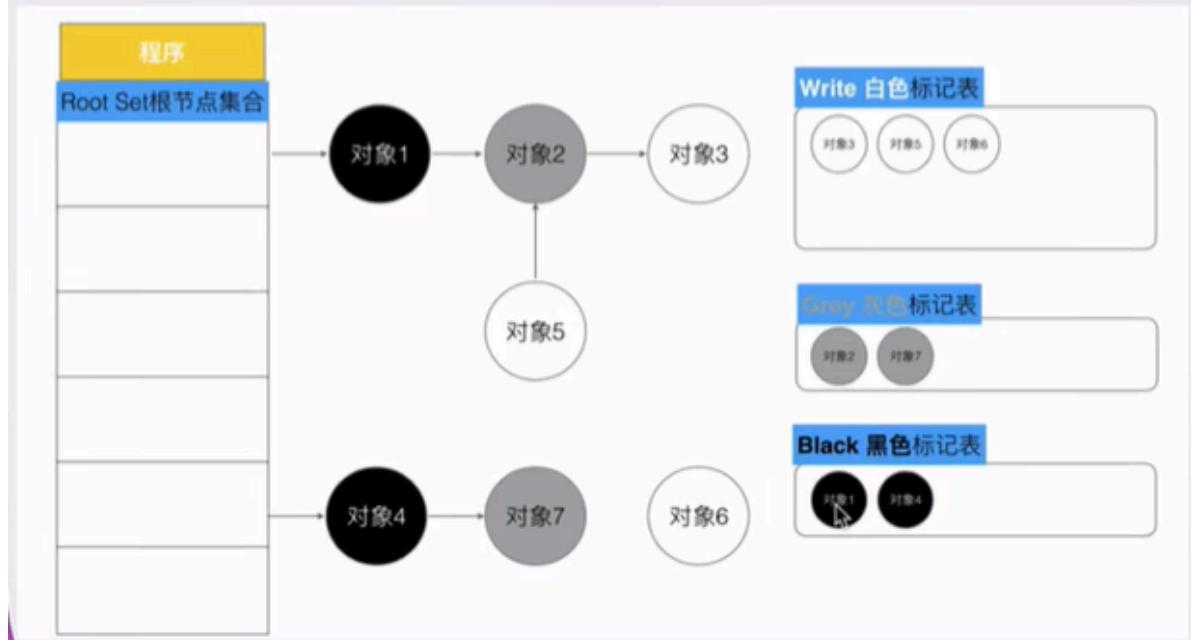
第二步

第二步，每次GC回收开始，然后从根节点开始遍历所有对象，把遍历到的对象从白色集合放入“灰色”集合。



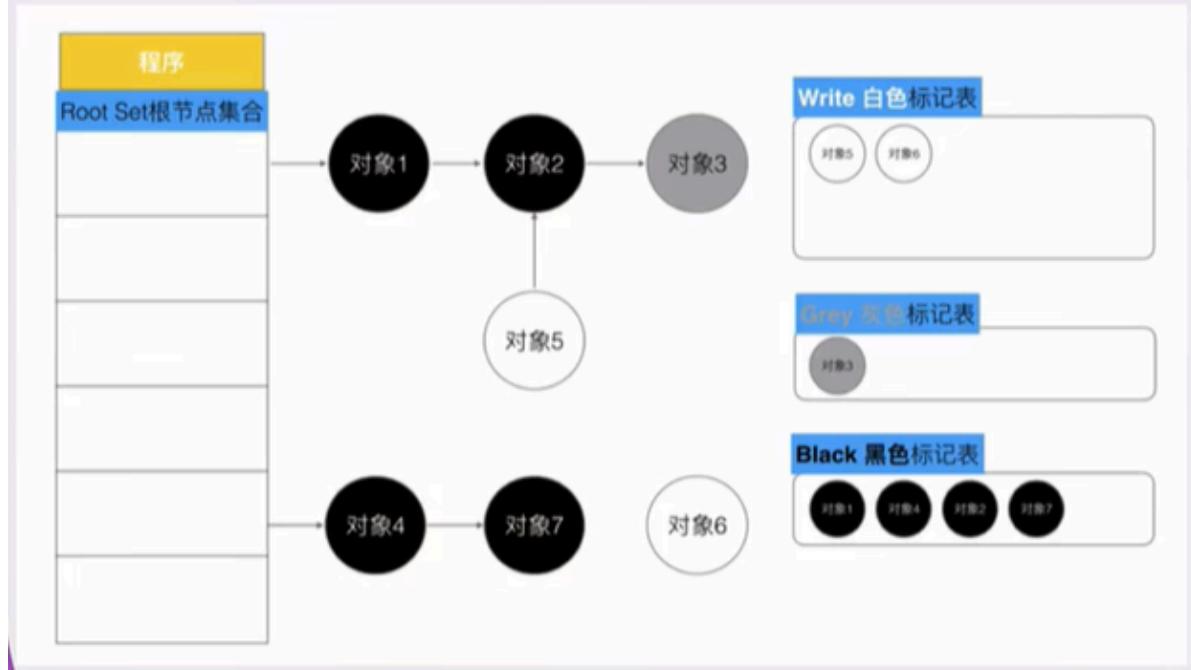
第三步

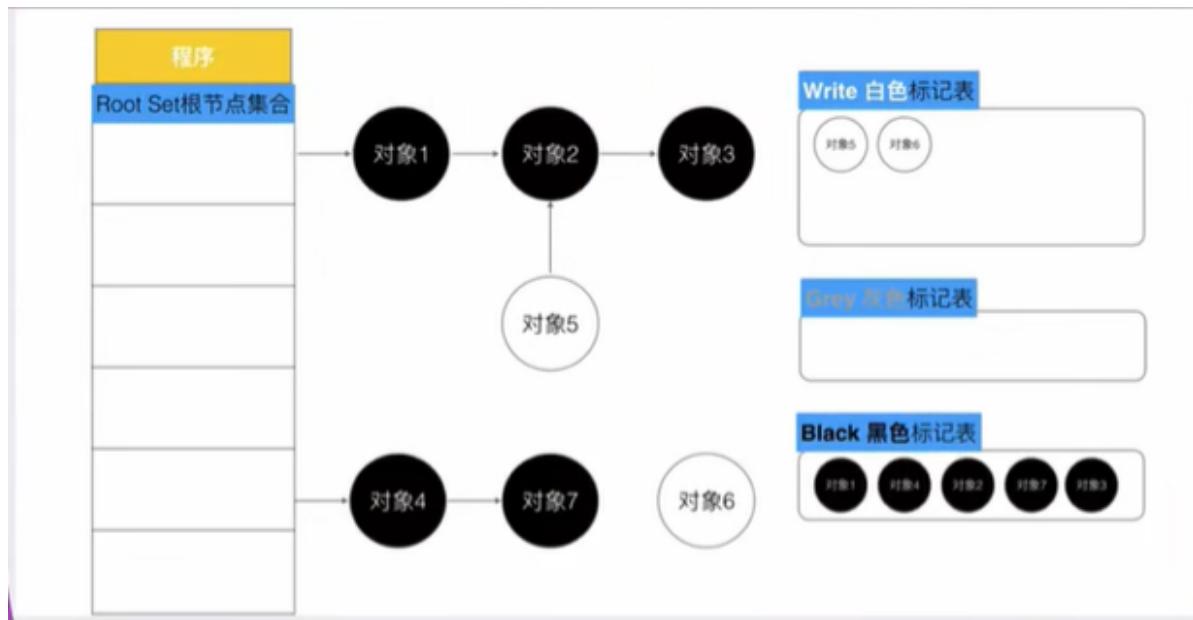
第三步, 遍历灰色集合, 将灰色对象引用的对象从白色集合放入灰色集合, 之后将此灰色对象放入黑色集合



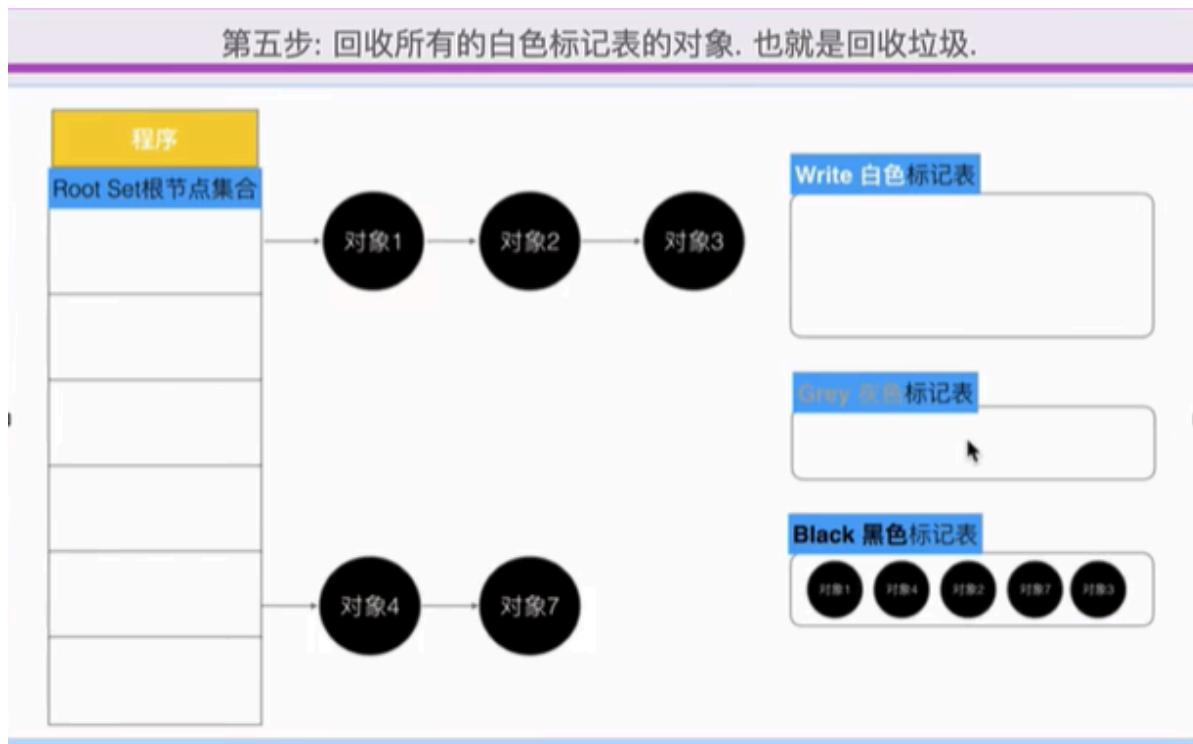
第四步

第四步, 重复第三步, 直到灰色中无任何对象.



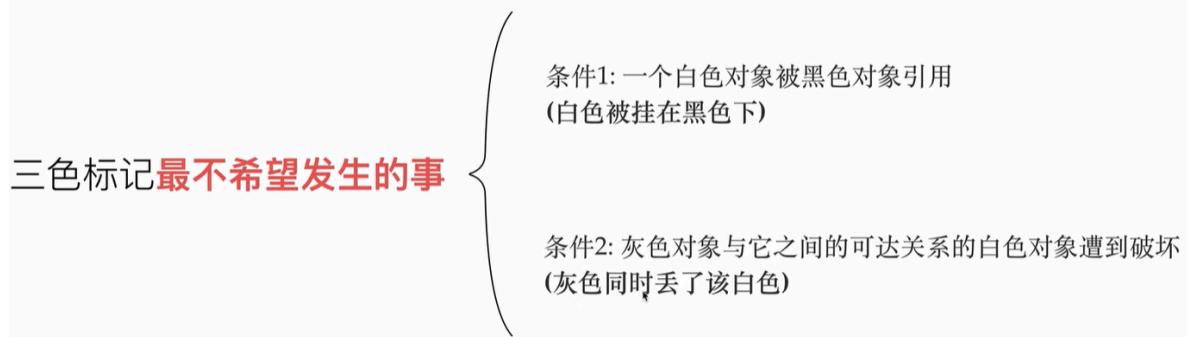


第五步



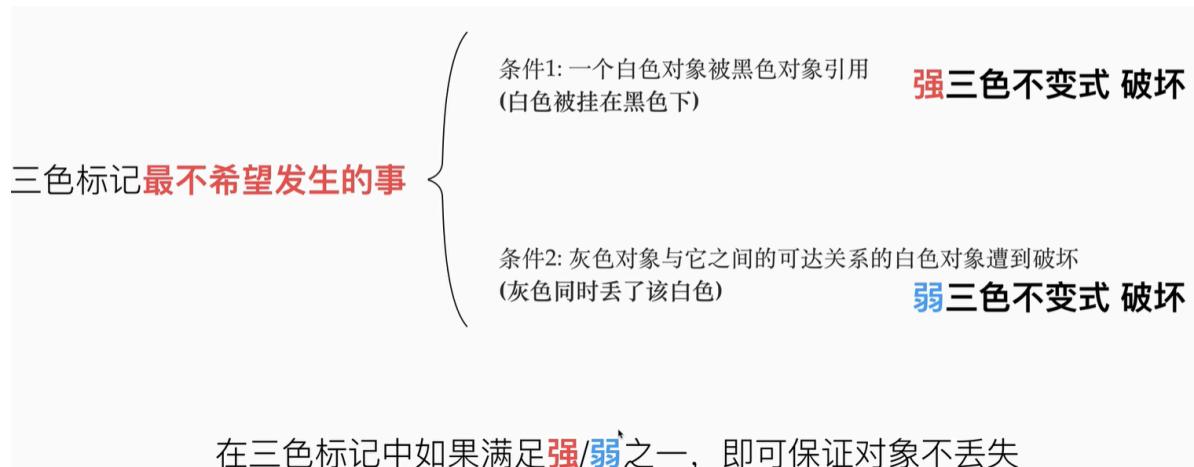
三色标记的问题

假如三色标记法不需要STW，当在标记的过程种，白色对象被黑色对象引用，但无灰色对象或有灰色对象的链路引用，就会触发错误。也就是说下面两种条件同时达到时，就会造成这个白色对象被错当作垃圾对象被GC。但是如果使用三色标记又使用STW，那么岂不是更复杂了。



强三色不变式和弱三色不变式

回顾三色标记法的问题，只有当2种条件同时达成才会造成错误。因此只需破坏其中一个条件成立即可，就和破坏死锁一样。破坏两个条件，分别是强三色不变式和弱三色不变式。



强三色不变式



弱三色不变式



强/弱三色不变式都是对黑色对象引用白色对象进行的一些限制。

屏障机制

垃圾收集中的屏障技术更像是一个钩子方法，它是在用户程序读取对象、创建新对象以及更新对象指针时执行的一段代码，根据操作类型的不同，我们可以将它们分成读屏障（Read barrier）和写屏障（Write barrier）两种，因为读屏障需要在读操作中加入代码片段，对用户程序的性能影响很大，所以编程语言往往都会采用写屏障保证三色不变性。

Golang也不例外，通过插入写屏障和删除写屏障技术保证三色不变性和GC的正确性。



插入写屏障

插入屏障

具体操作: 在A对象引用B对象的时候，B对象被标记为灰色。(将B挂在A下游，B必须被标记为灰色)

满足: 强三色不变式。(不存在黑色对象引用白色对象的情况了，因为白色会强制变成灰色)

```
//伪码
添加下游对象(当前下游对象slot, 新下游对象ptr) {
    //1
    标记灰色(新下游对象ptr)
    ,
    //2
    当前下游对象slot = 新下游对象ptr
}
```

//场景

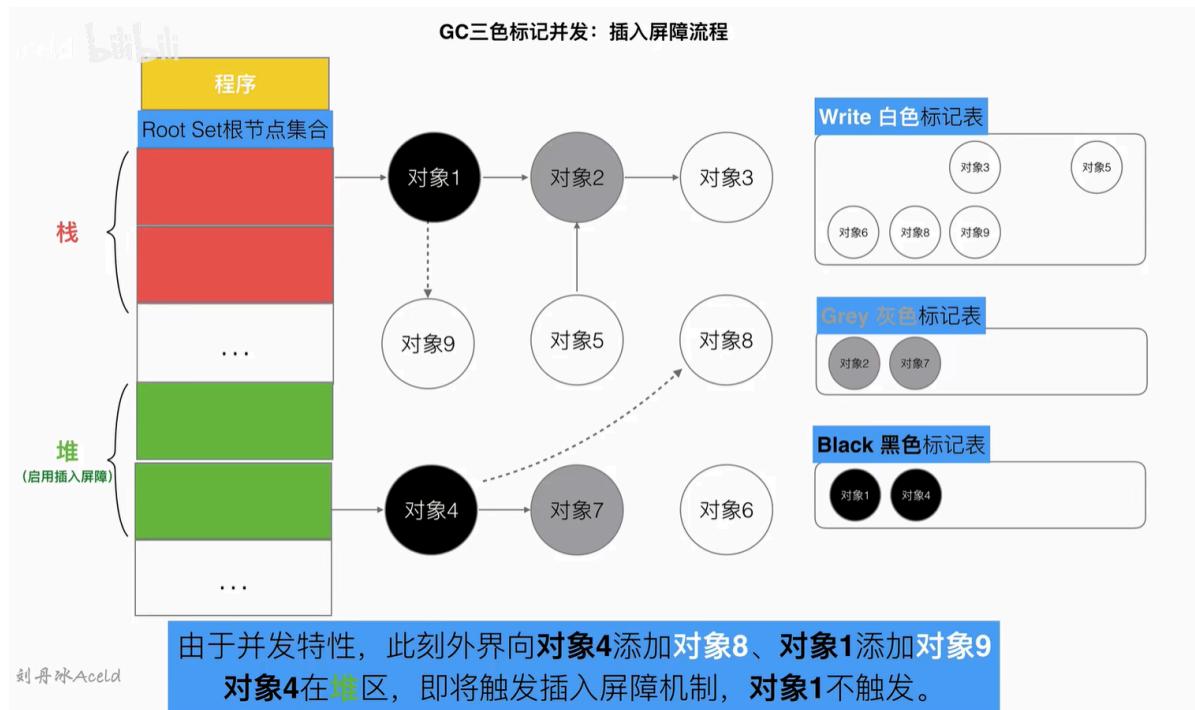
A. 添加下游对象(nil, B) //A 之前没有下游，新添加一个下游对象B，B被标记为灰色

A. 添加下游对象(C, B) //A 将下游对象C 更换为B，B被标记为灰色

插入写屏障流程

关于GC为什么处理栈上的对象：如果栈上的对象引用了堆上的对象，GC需要访问并标记这些堆对象。然而，栈上的对象本身通常不需要被GC回收，因为它们的生命周期由函数调用和返回控制，一旦函数返回，相关的栈帧即自动清理。尽管Go的GC不直接“回收”栈上的对象，它确实会在垃圾回收的标记阶段检查栈上的对象，以确保正确处理栈上对象可能引用的堆上对象。这样做是为了维护堆内存的正确引用图，确保只有真正不再被任何活跃对象引用的堆内存才被标记为可回收。

但一般来说为了程序稳定性和栈对象加入写指针开销，栈对象不启用插入屏障。而是在结束时通过STW重新扫描栈对象。



删除写屏障

删除屏障

具体操作: 被删除的对象, 如果自身为灰色或者白色, 那么被标记为灰色。

满足: 弱三色不变式. (保护灰色对象到白色对象的路径不会断)

//伪码

```

添加下游对象(当前下游对象slot, 新下游对象ptr) {
    //1
    if (当前下游对象slot是灰色 || 当前下游对象slot是白色) {
        标记灰色(当前下游对象slot)      //slot为被删除对象, 标记为灰色
    }

    //2
    当前下游对象slot = 新下游对象ptr
}

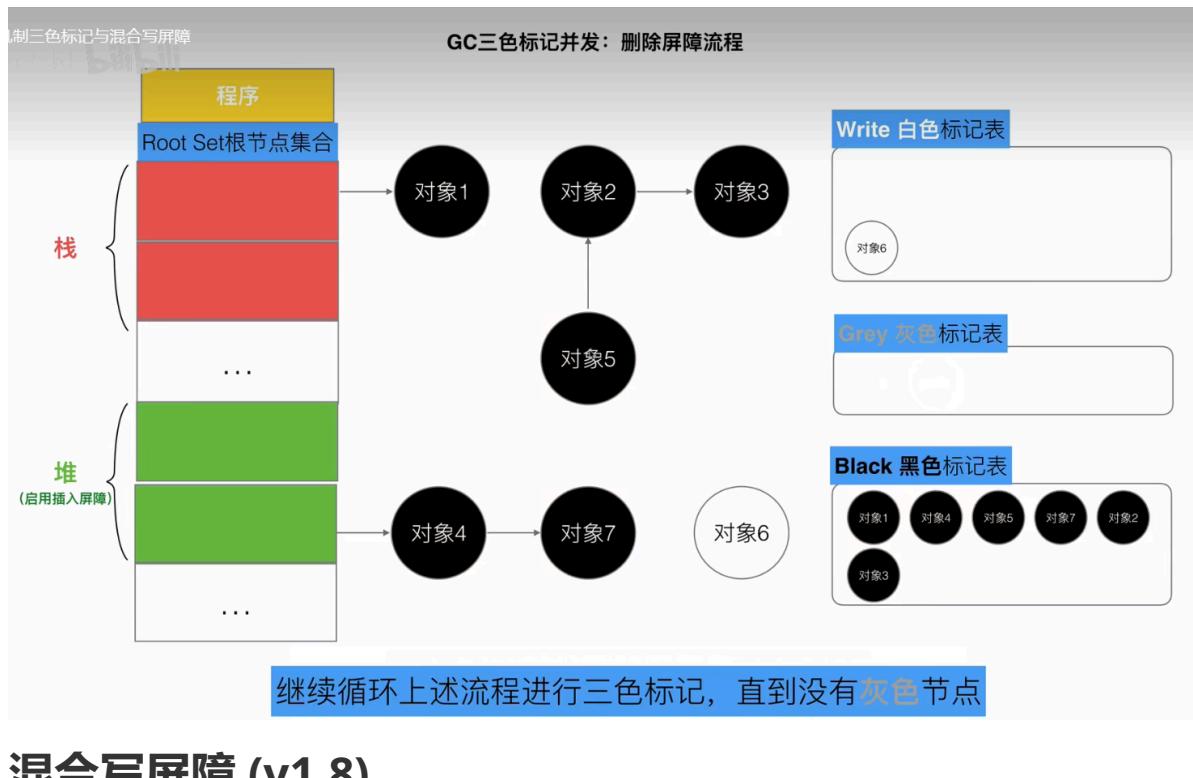
```

//场景

- A. 添加下游对象(B, nil) //A对象, 删除B对象的引用。 B被A删除, 被标记为灰(如果B之前为白)
- A. 添加下游对象(B, C) //A对象, 更换下游B变成C。 B被A删除, 被标记为灰(如果B之前为白)

删除写屏障的流程:

一开始5被1引用, 然后删除。被标记为灰色, 后续被遍历, 从而确保假如有一个黑色的去引用它, 后续遍历不到, 从而被标记成垃圾; 但回收精度低, 也就是说假如对象5没有引用它的对象了, 那么这一轮gc, 对象5不会被删除, 下一轮才会被删除



混合写屏障 (v1.8)

Go V1.8的三色标记法+混合写屏障机制

- 具体操作:**
- 1、GC开始将栈上的对象全部扫描并标记为黑色(之后不再进行第二次重复扫描，无需STW)
 - 2、GC期间，任何在栈上创建的新对象，均为黑色。
 - 3、被删除的对象标记为灰色。
 - 4、被添加的对象标记为灰色。

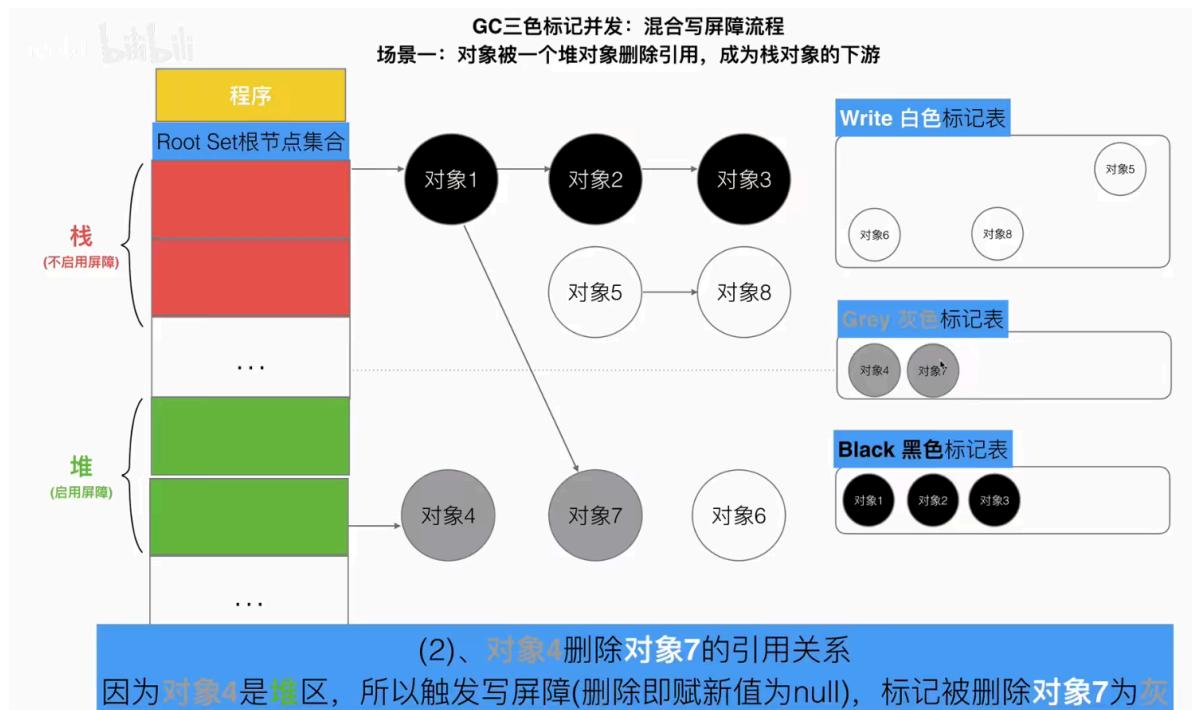
满足: 变形的弱三色不变式. (结合了插入、删除写屏障两者的有点)

```
//伪码
添加下游对象(当前下游对象slot, 新下游对象ptr) {
    //1
    标记灰色(当前下游对象slot)    //只要当前下游对象被移走, 就标记灰色

    //2
    标记灰色(新下游对象ptr)

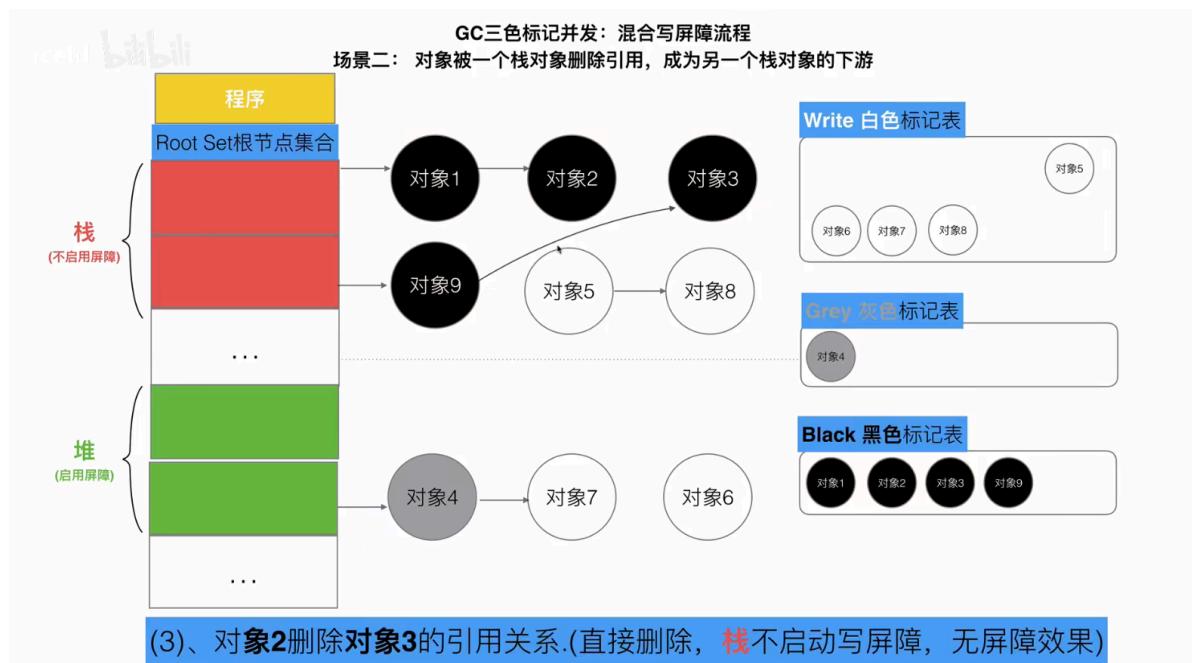
    //3
    当前下游对象slot = 新下游对象ptr
}
```

场景1

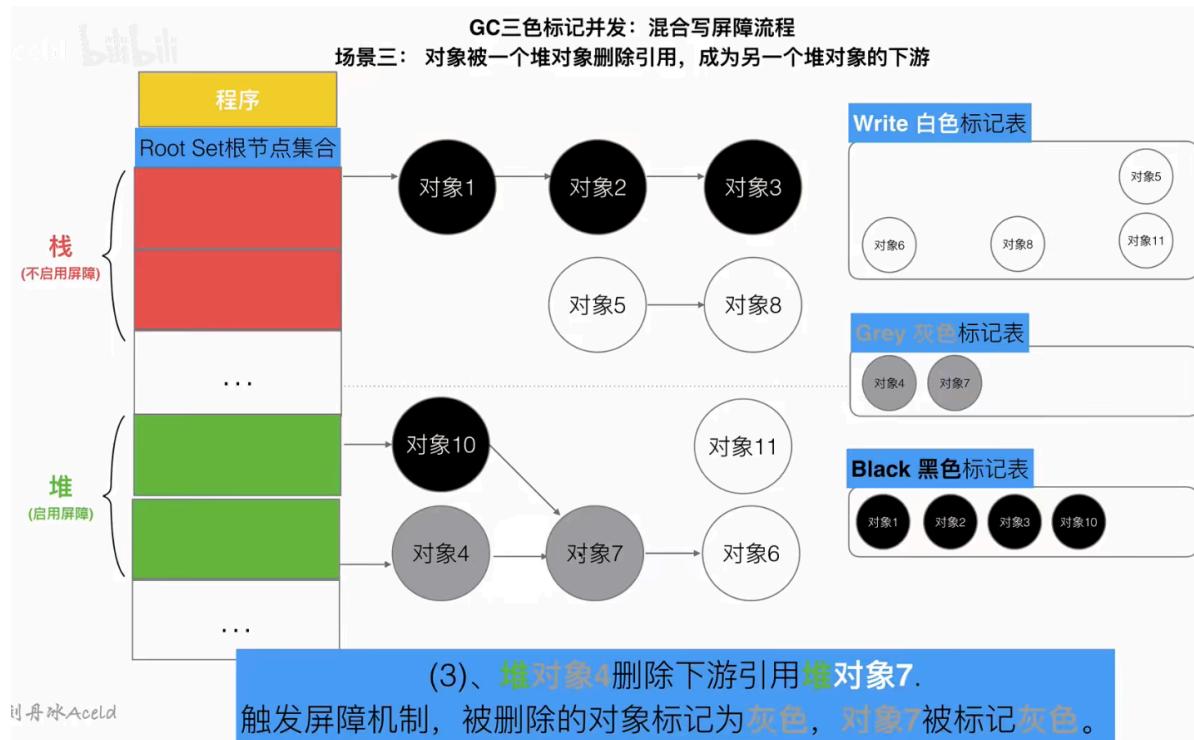


场景2

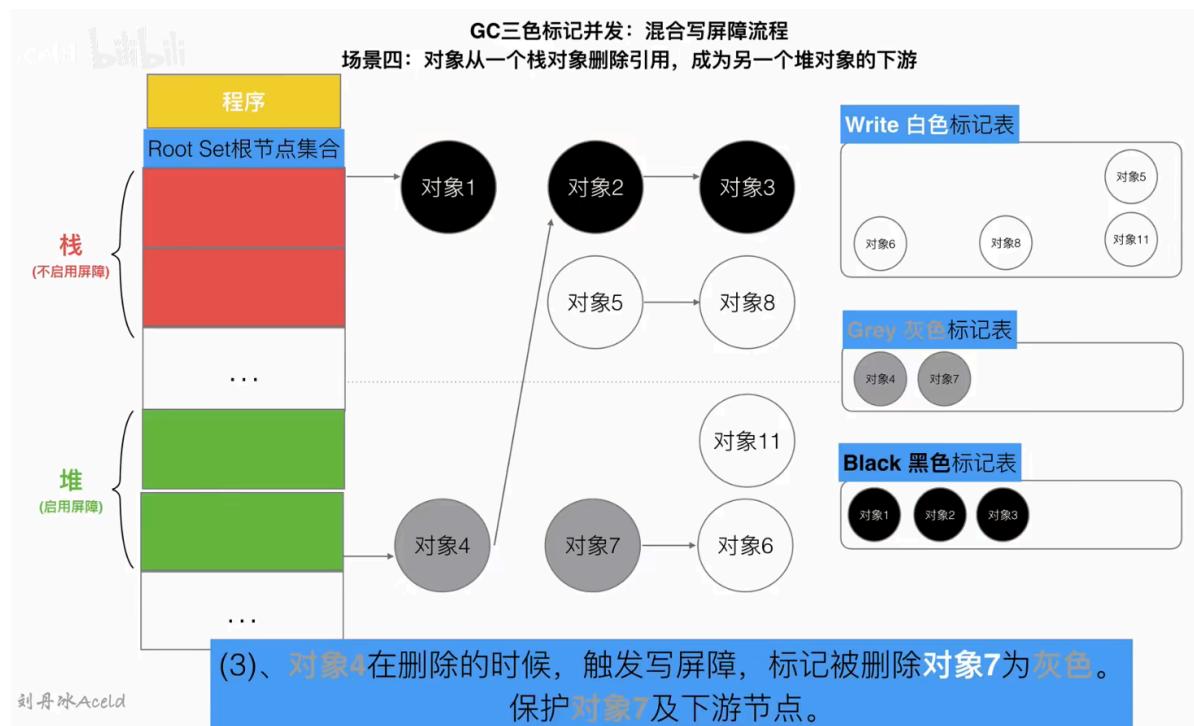
下面的描述有点问题，GC不会直接对栈对象进行删除。



场景3



场景4



总体来说只需要记住栈上的对象被标记为黑色后不变，堆上要删除或添加对象都标记为灰色。

增量和并发

今天的计算机往往都是多核的处理器，垃圾收集器一旦开始执行就会浪费大量的计算资源，为了减少应用程序暂停的最长时间和垃圾收集的总暂停时间，我们会使用下面的策略优化现代的垃圾收集器：

- 增量垃圾收集 — 增量地标记和清除垃圾，降低应用程序单次暂停的最长时间；例如，标记阶段可以分成多个小的标记步骤，每处理一定量的对象后，GC暂停，应用程序继续执行，然后GC再次启动处理下一批对象。

- 并发垃圾收集 — 利用多核的计算资源，在用户程序执行时并发标记和清除垃圾；在并发GC模式下，大部分垃圾收集活动（如标记和清除）是在应用程序线程正在运行时同时进行的，这减少了GC需要的总暂停时间。

当然上述两种机制都要配合屏障机制，并且GC需要在提前进行，不能等内存告急了才开始，那样无异于停止程序。

Golang GC的变更

下图中的变更和前面讲的差不多，比较大的节点是1.5的三色标记、1.7的并行栈收缩，1.8的混合写屏障（包括1.9的彻底移除STW前面也讲到了）。

1. [v1.0](#) – 完全串行的标记和清除过程，需要暂停整个程序；
2. [v1.1](#) – 在多核主机并行执行垃圾收集的标记和清除阶段¹¹；
3. [v1.3](#) – 运行时基于只有指针类型的值包含指针的假设增加了对栈内存的精确扫描支持，实现了真正精确的垃圾收集¹²：
 - 将 `unsafe.Pointer` 类型转换成整数类型的值认定为不合法的，可能会造成悬挂指针等严重问题；
4. [v1.5](#) – 实现了基于三色标记清扫的并发垃圾收集器¹³：
 - 大幅度降低垃圾收集的延迟从几百 ms 降低至 10ms 以下；
 - 计算垃圾收集启动的合适时间并通过并发加速垃圾收集的过程；
5. [v1.6](#) – 实现了去中心化的垃圾收集协调器：
 - 基于显式的状态机使得任意 Goroutine 都能触发垃圾收集的状态迁移；
 - 使用密集的位图替代空闲链表表示的堆内存，降低清除阶段的 CPU 占用¹⁴；
6. [v1.7](#) – 通过并行栈收缩将垃圾收集的时间缩短至 2ms 以内¹⁵；
7. [v1.8](#) – 使用混合写屏障将垃圾收集的时间缩短至 0.5ms 以内¹⁶；
8. [v1.9](#) – 彻底移除暂停程序的重新扫描栈的过程¹⁷；
9. [v1.10](#) – 更新了垃圾收集调频器（Pacer）的实现，分离软硬堆大小的目标¹⁸；
10. [v1.12](#) – 使用新的标记终止算法简化垃圾收集器的几个阶段¹⁹；
11. [v1.13](#) – 通过新的 Scavenger 解决瞬时内存占用过高的应用程序向操作系统归还内存的问题²⁰；
12. [v1.14](#) – 使用全新的页分配器优化内存分配的速度²¹；

Golang GC的时机

- Go 语言运行时的默认配置会在堆内存达到上一次垃圾收集的 2 倍时（因为并发垃圾收集器与程序一起运行，无法准确控制堆内存的大小，一般在达到目标前触发），触发新一轮的垃圾回收，这个行为可以通过 `GOGC` 变量调整。它的默认值为 100，即增长 100% 的堆内存才会触发 GC；
- 如果一定时间内没有通过方式 1 触发，也会触发新的循环，这个定时时长由 `runtime.forcegcperiod` 变量控制，默认为 2 分钟。
- 手动触发

Golang调度器的由来

- 单进程
- 多进程/多线程
- 协程及其与线程的数量模型
- 早期的Golang调度器



GMP模型

GMP模型简介



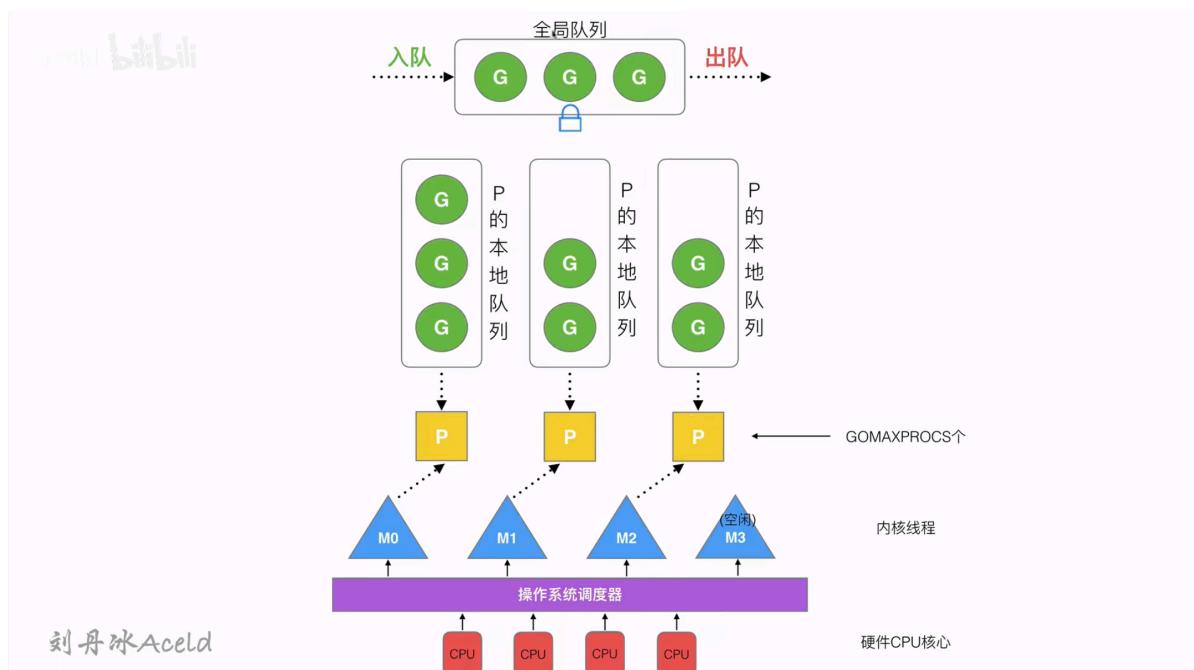


P和M的总数量关系不是1: 1的，M可能更多，P也可能更多，但一个P同一时刻最多只有一个M与其绑定。

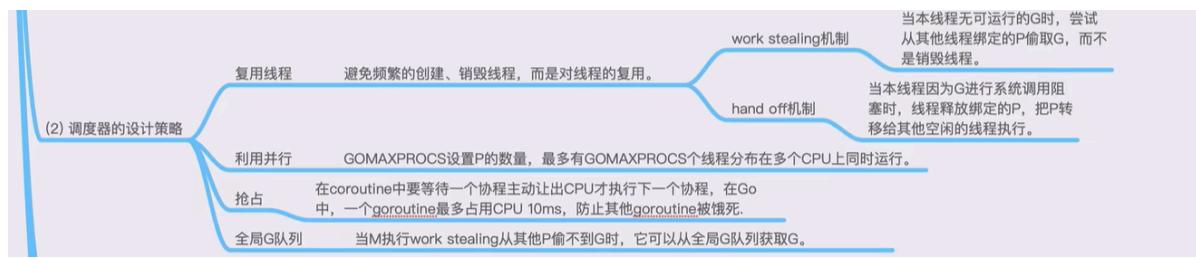
至于为什么不直接将本地队列放在M上、而是要放在P上呢？**这是因为当一个线程M阻塞（可能执行系统调用或IO请求）的时候，可以将和它绑定的P上的G转移到其他线程M去执行，如果直接把可运行G组成的本地队列绑定到M，则万一当前M阻塞，它拥有的G就不能给到其他M去执行了。**

在Go语言的调度器中，`P`结构体拥有一个本地的环形队列，用于存放可运行的 `Goroutine`（简称 `G`）。这个队列的长度为 256，设计为环形主要是为了高效地进行元素的插入和删除操作。**环形队列的一个关键特性是它支持无锁的访问，这主要是通过使用比较并交换（Compare-And-Swap, CAS）操作实现的。**CAS：先比较内存位置当前值是否匹配预期值，如果为true则进行写入，否则不会执行任何写入。通过CAS操作确保G入队和出队的原子性。更具体来说：

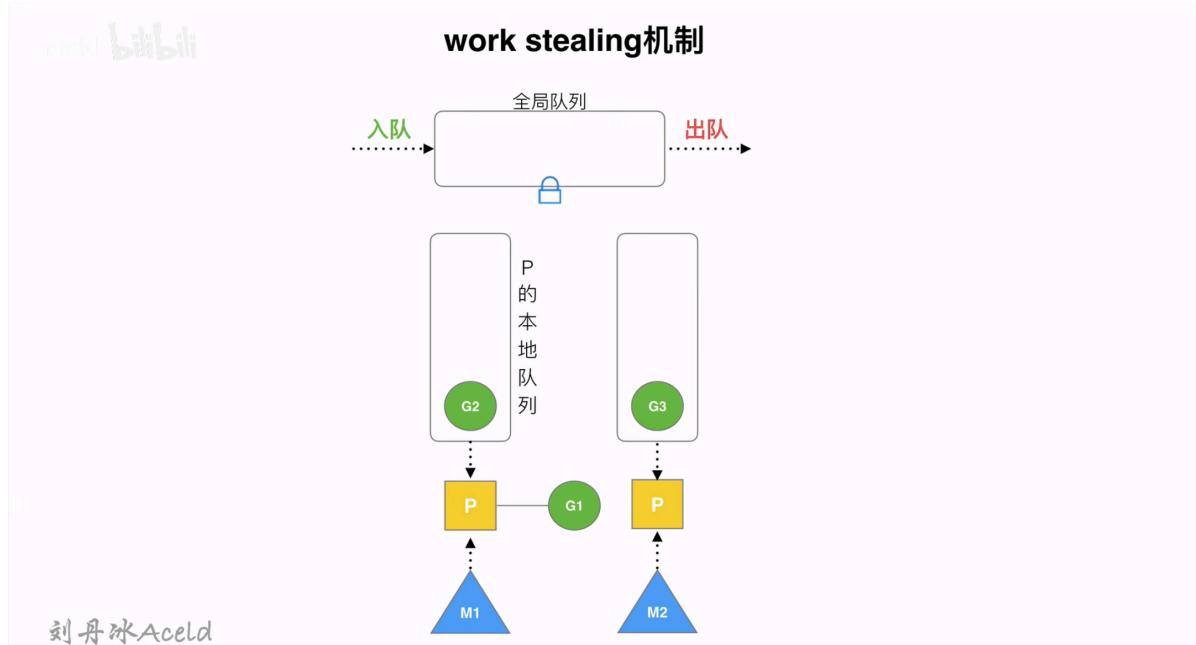
- 入队（Enqueue）：当一个Goroutine被创建或者需要调度时，它会被添加到P的本地队列中。入队操作首先读取队列的tail索引，然后尝试通过CAS将G放在tail位置，并更新tail索引。如果CAS操作失败（通常是因为其他线程已经修改了tail），则重试直到成功。
- 出队（Dequeue）：当一个M需要执行Goroutine时，它会从P的本地队列的head索引处取出一个G来执行。出队操作同样使用CAS来更新head索引，确保在多线程环境下的正确性和一致性。



Golang调度器的设计策略

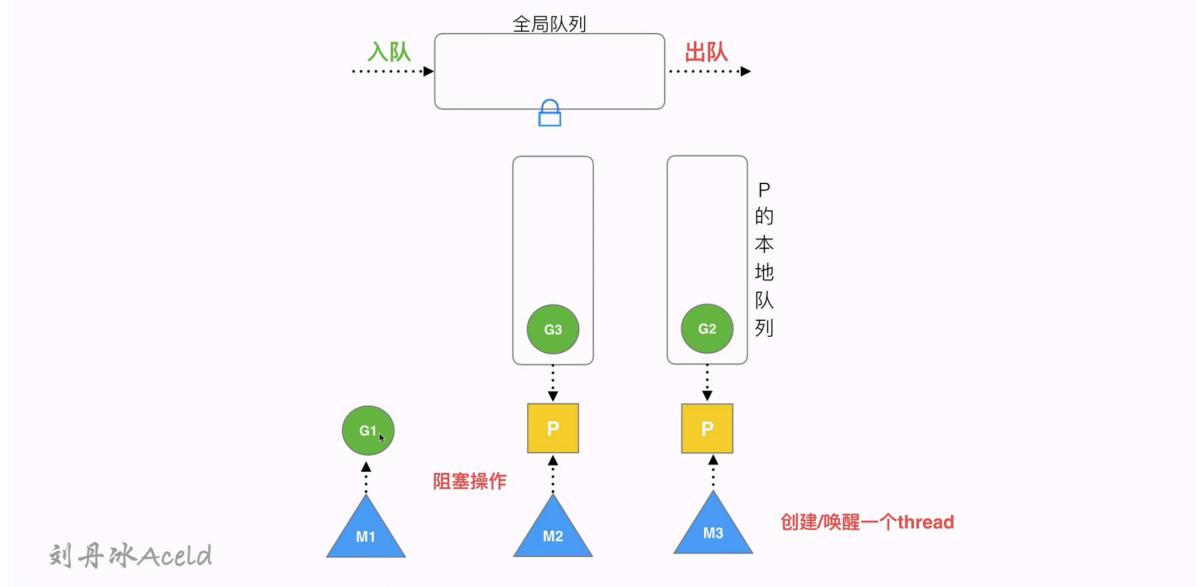


work stealing: 优先去全局队列中拿G，没有再去其他P的队列中stealing，一般是拿P队列的后面一半。下图中M2绑定的P队列中一开始无G执行，但此时全局队列中也没有G，只有从M1的P队列中偷G3过来调度。



hand off: 一开始M1绑定的是M3现在绑定的P，在执行G1的过程中发生了阻塞，为了不浪费P的资源，会从M列表中唤醒一个M，并将P进行转移。当G1阻塞完成，其会优先去寻找之前执行的P，如果P已满并且P列表为空则加入全局队列，同时M休眠或GC。一般来说G和M会一起阻塞，但如果是M上的G进入Channel阻塞，则该M不会一起进入阻塞，因为Channel数据传输涉及内存拷贝，不涉及系统资源等待。相反，它会从本地或全局运行队列中寻找另一个可运行的Goroutine来执行。这样做的目的是保持CPU的利用率，避免因单个Goroutine的阻塞而导致整个线程空闲。

hand off机制

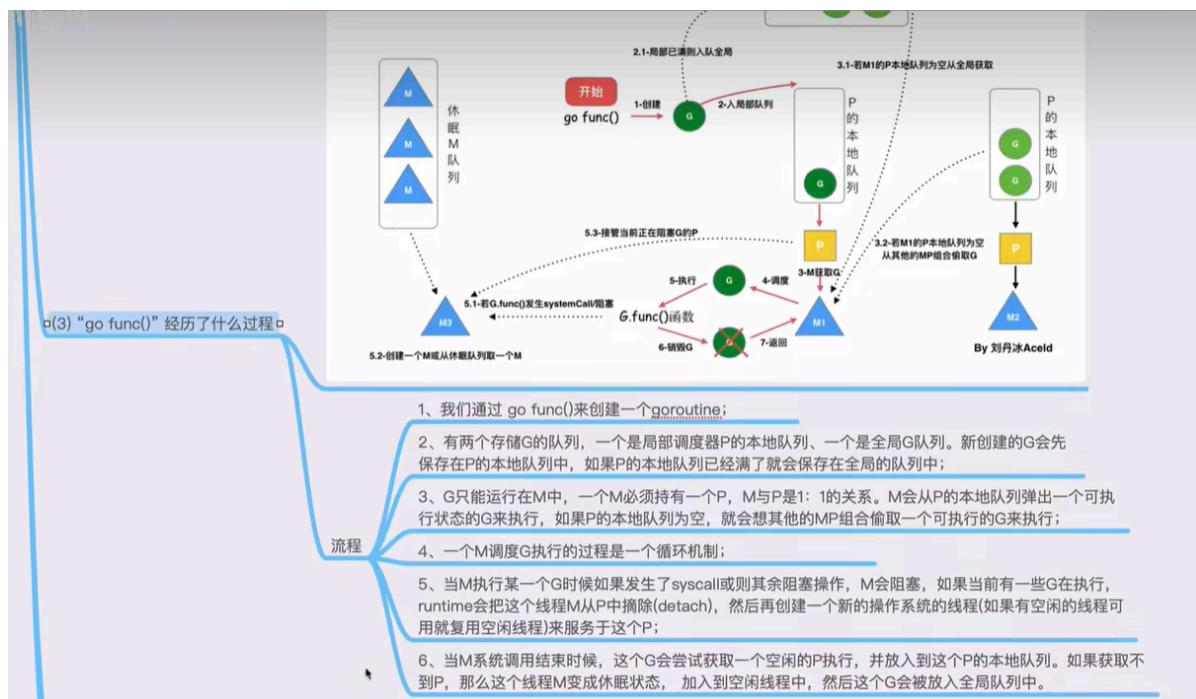


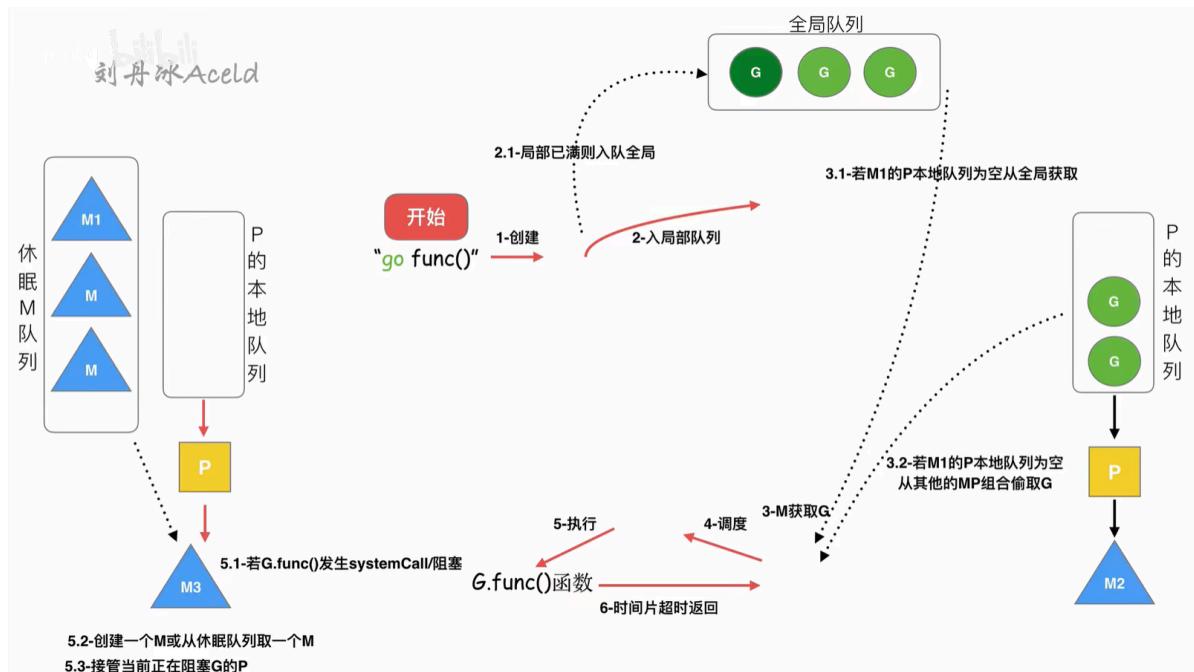
基于协作的抢占式调度：为了保证公平性和防止 Goroutine 饥饿问题，Go 程序会保证每个 G 运行 10ms 就让出 M，交给其他 G 去执行，这个 G 运行 10ms 就让出 M 的机制，是由单独的系统监控线程通过 `retake()` 函数给当前的 G 发送抢占信号实现的，如果所在的 P 没有陷入系统调用且没有满，让出的 G 优先进入本地 P 队列，否则进入全局队列。注意这里的信号更多的是 Go Runtime 内部的处理，而非操作系统级别的。

基于信号的真抢占机制：尽管基于协作的抢占机制能够缓解长时间 GC 导致整个程序无法工作和大多数 Goroutine 饥饿问题，但是还是有部分情况下，Go 调度器有无法被抢占的情况，例如，`for` 循环或者垃圾回收长时间占用线程，为了解决这些问题，**Go1.14 引入了基于信号的抢占式调度机制，能够解决 GC 垃圾回收和栈扫描时存在的问题。**

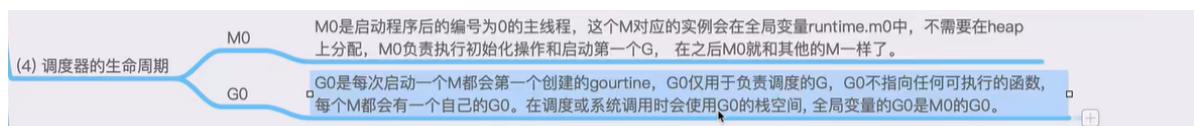
全局队列：全局队列的 G 永远会有 1/61 的机会被获取到，调度循环中，优先从本地队列获取 G 执行，不过每隔61次，就会直接从全局队列获取，至于为啥是 61 次，Dmitry 的视频讲解了，就是要一个既不大又不小的数，而且不能跟其他的常见的2的幂次方的数如 64 或 48 重合。

go func()的过程





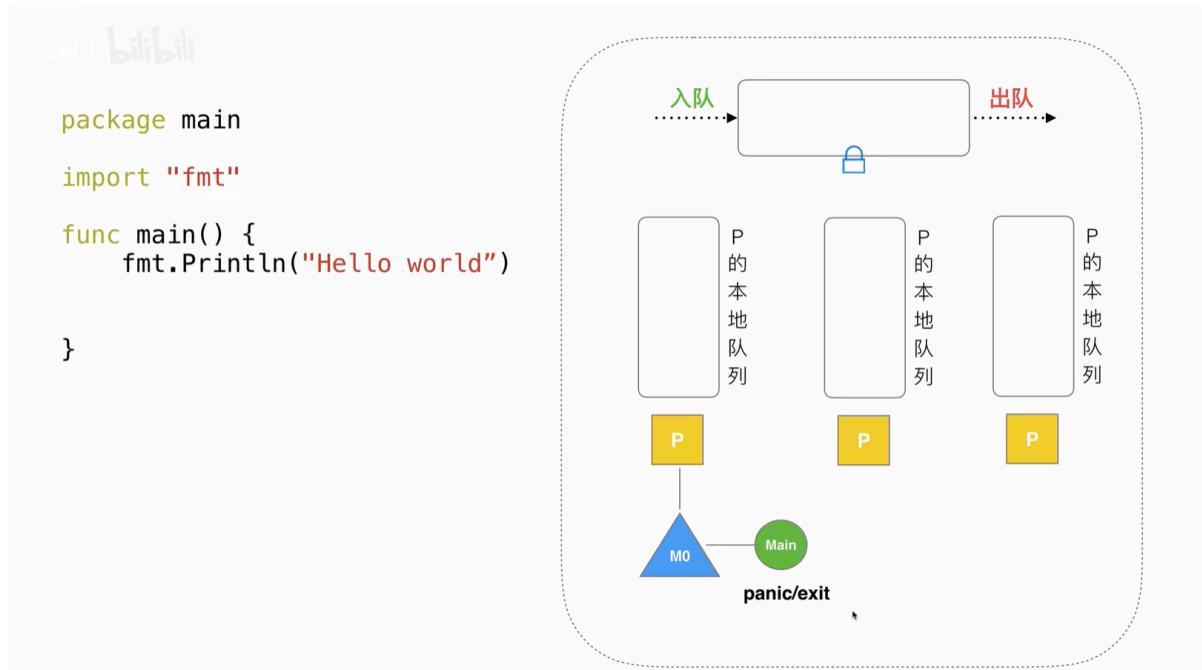
Golang调度器的生命周期



M0 & G0



以下面的例子，即使没有明确使用`go func()`创建goroutine，但`main`函数也是一个G，并被G0调度在M上执行。



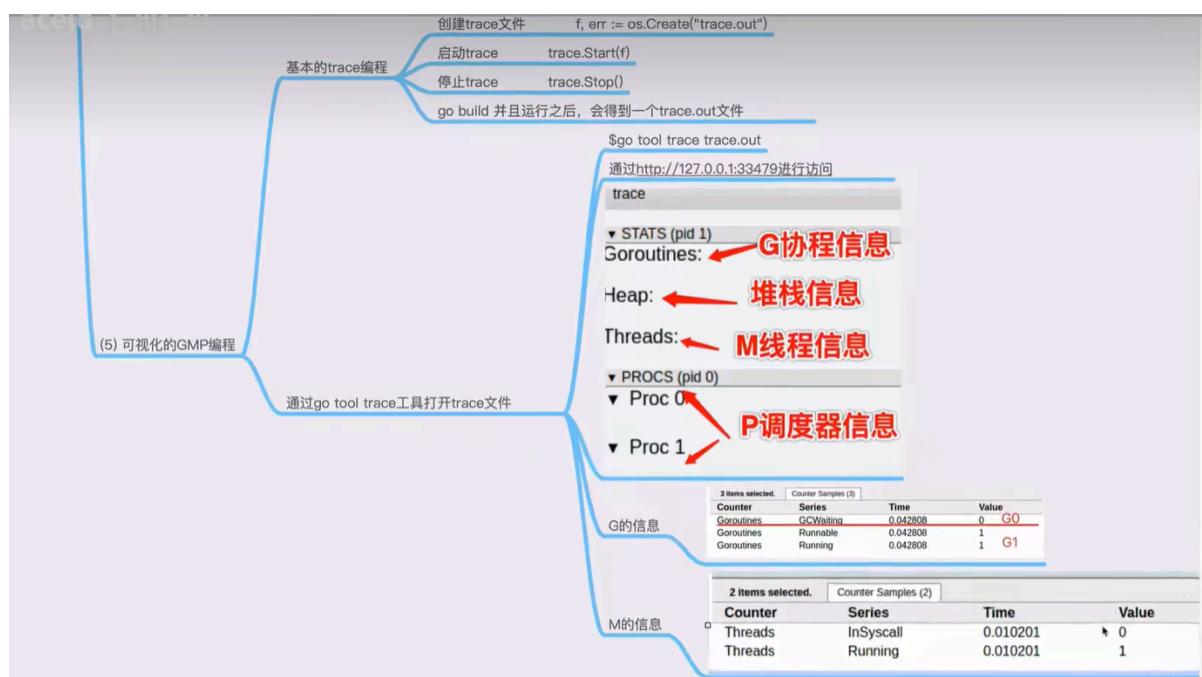
调度器的启动逻辑是：初始化 g0 和 m0，并将二者互相绑定，m0 是程序启动后的初始线程，**g0 是 m0 线程的系统栈代表的 G 结构体，负责普通 G 在 M 上的调度切换** --> `runtime.schedinit()`：M、P 的初始化过程，分别调用 `runtime.mcommoninit()` 初始化 M 的全局队列 allm、调用 `runtime.procresize()` 初始化全局 P 队列 allp --> `runtime.newproc()`：负责获取空闲的 G 或创建新的 G --> `runtime.mstart()` 启动调度循环；；

调度器的循环逻辑是：运行函数 `schedule()` --> 通过 `runtime.gobrunqget()` 从全局队列、通过 `runtime.runqget()` 从 P 本地队列、`runtime.findRunnable` 从各个地方，获取一个可执行的 G --> 调用 `runtime.execute()` 执行 G --> 调用 `runtime.gogo()` 在汇编代码层面上真正执行 G --> 调用 `runtime.goexit0()` 执行 G 的清理工作，重新将 G 加入 P 的空闲队列 --> 调用 `runtime.schedule()` 进入下一次调度循环。

G 在运行时中的状态可以简化成三种：等待中 `G.waiting`（比如阻塞等待）、可运行 `G runnable`、运行中 `G.running`，运行期间大部分情况是在这三种状态间来回切换。

可视化的GMP编程 (go tool trace)

go tool trace一般做性能分析。



使用debug trace。

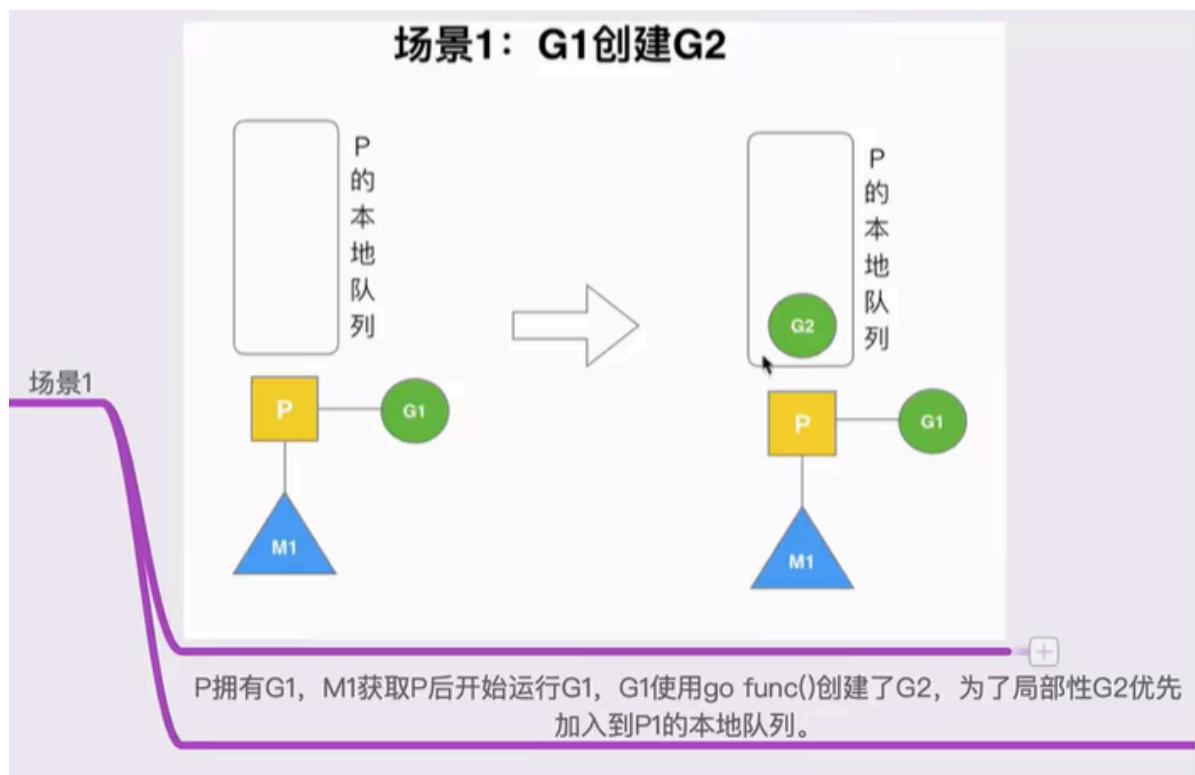
```
GODEBUG=schedtrace=1000 ./可执行程序
i@theima:~$ P3-debug_GMP$ GODEBUG=schedtrace=1000 ./trace2
SCHED 0ms: gomaxprocs=2 idleprocs=1 threads=4 spinningthreads=0 idlethreads=1 runqueue=0 [0 0]
hello GMP
SCHED 1007ms: gomaxprocs=2 idleprocs=2 threads=4 spinningthreads=0 idlethreads=2 runqueue=0 [0 0]
hello GMP
SCHED 2018ms: gomaxprocs=2 idleprocs=2 threads=4 spinningthreads=0 idlethreads=2 runqueue=0 [0 0]
hello GMP
SCHED 3015ms: gomaxprocs=2 idleprocs=2 threads=4 spinningthreads=0 idlethreads=2 runqueue=0 [0 0]
SCHED 4025ms: gomaxprocs=2 idleprocs=2 threads=4 spinningthreads=0 idlethreads=2 runqueue=0 [0 0]
hello GMP
```

通过Debug trace查看GMP信息

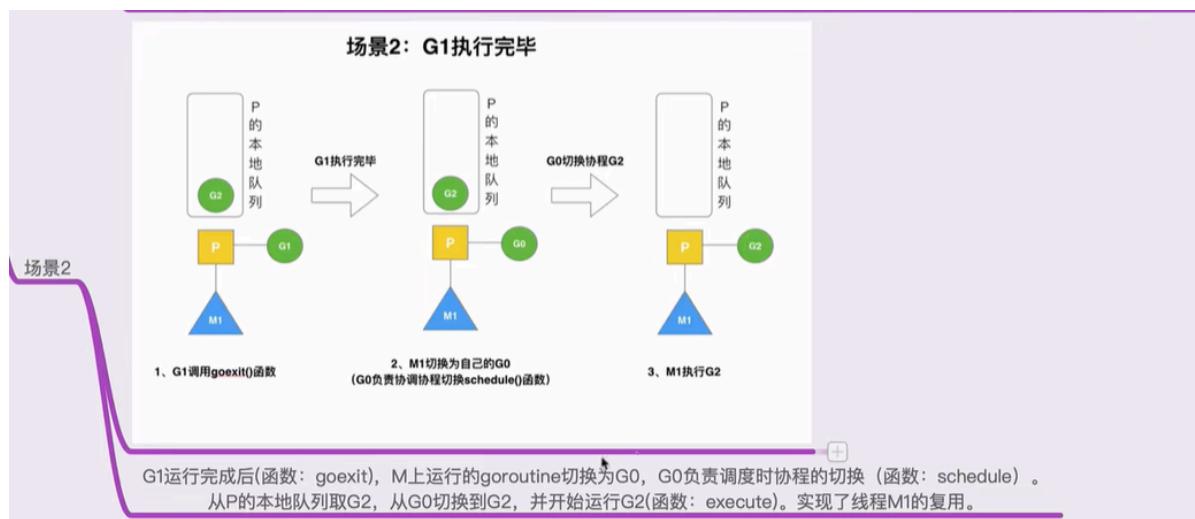
SCHED 调试的信息
0ms 从程序启动到输出经历的时间
gomaxprocs P的数量 一般默认是和CPU的核心数是一致的
idleprocs 处理idle状态的P的数量, gomaxprocs=idleprocs= 目前正在执行的p的数量
threads 线程数量(包括M0, 包括GODEBUG调试的线程)
spinningthreads 处于自旋状态的thread数量
idlethread 处理idle状态的thread
runqueue 全局G队列中的G的数量
[0,0] 每个P的local queue本地队列中, 目前存在G的数量

GMP场景分析

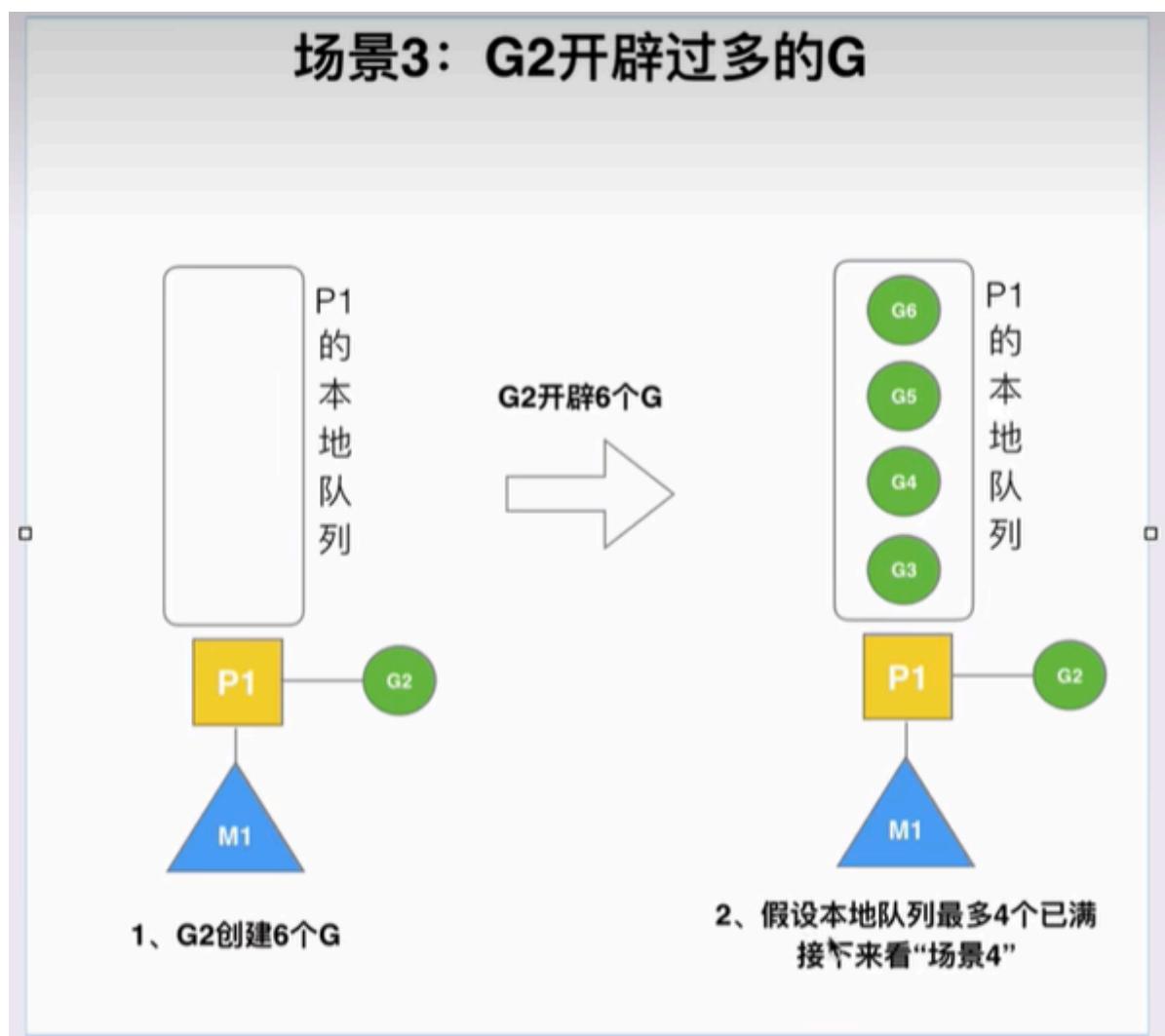
场景1



场景2



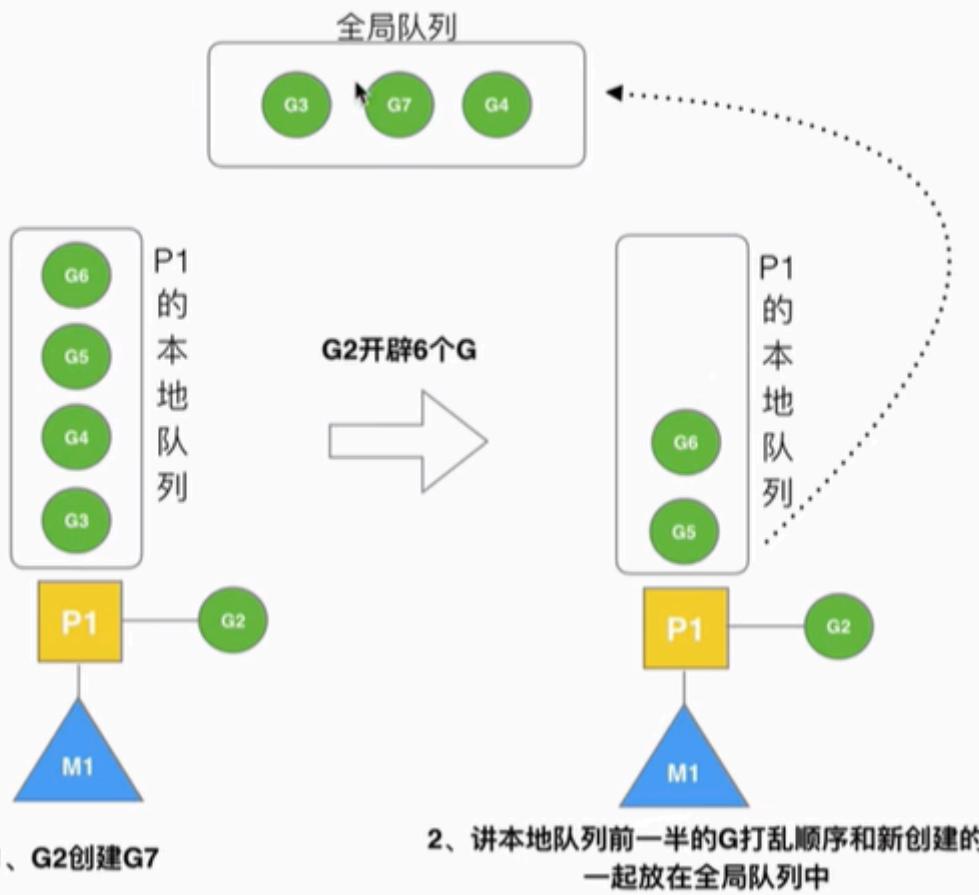
场景3



场景4

此时P1本地队列已满了，但G2要创建G7，调度器会将P1本地队列的前面一半和新创建的G7打乱顺序加入到全局队列中。

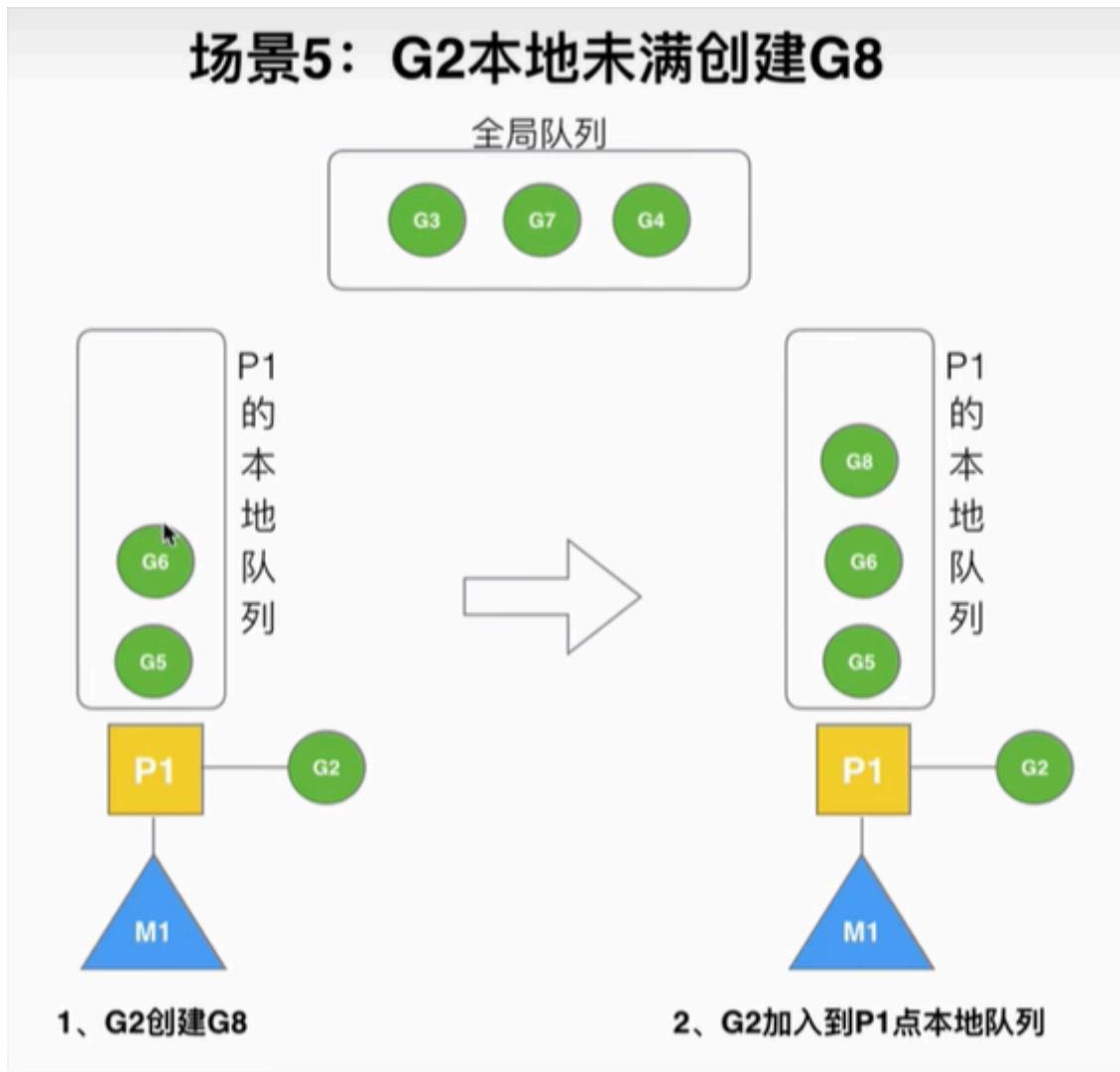
场景4：G2本地满再创建G7



场景5

此时P1全局队列未满，G2创建G8则会和场景1一样放入至P1的本地队列。

场景5：G2本地未满创建G8



场景6

如果有空闲的M和P，会尝试让这个组合去执行G，实现负载均衡。如果P和M没有G来执行，那么M此时为自旋状态，一直去寻找G。

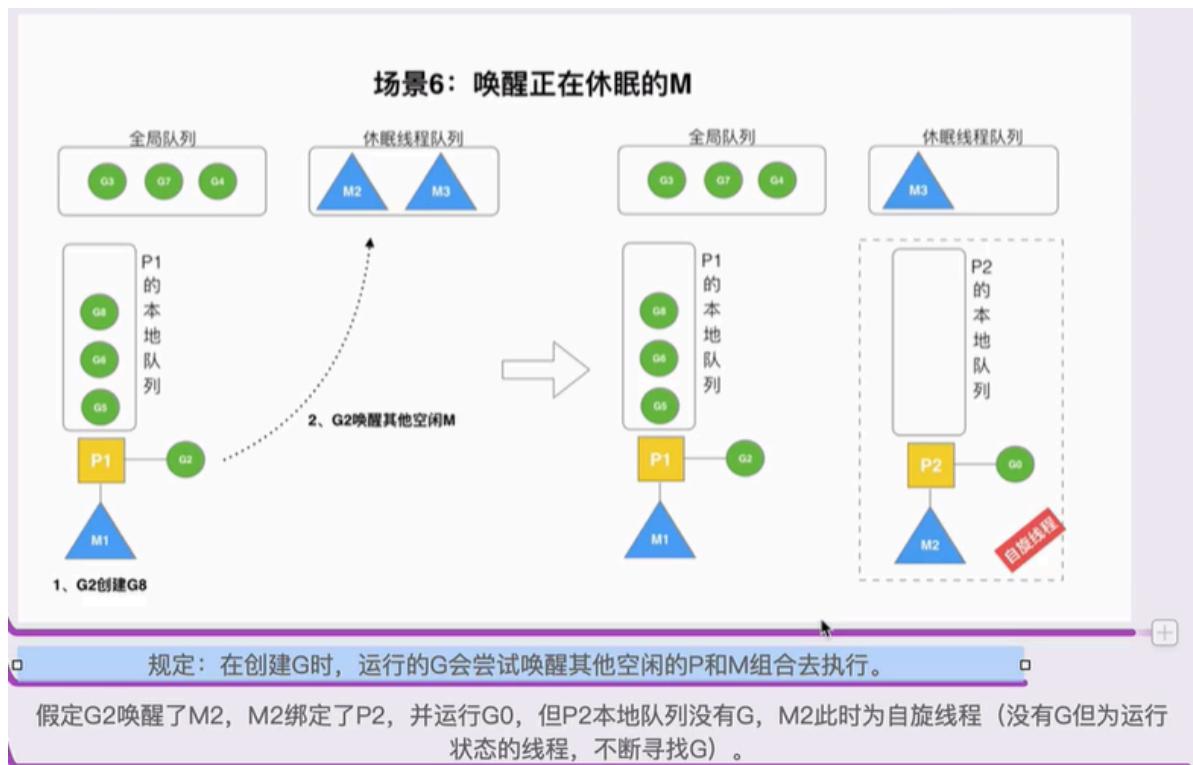
自旋线程M的寻找方式：当一个M（操作系统线程）在执行完其当前的Goroutine后，它会尝试寻找新的Goroutine来执行。寻找的顺序通常如下：

- 本地运行队列：每个P拥有一个本地的Goroutine队列。M首先检查其绑定的P的本地队列是否有待执行的Goroutines。
- 全局运行队列：如果本地队列为空，M会检查全局Goroutine队列。全局队列是一个系统级的队列，存储了那些还未被分配到某个P的Goroutines。
- 从其他P窃取：如果全局队列也为空，M会尝试从其他P的本地队列中“窃取”Goroutines。这通常是一种称为“work stealing”（工作窃取）的算法完成的，M会从其他P的队列的尾部尝试窃取Goroutines，以减少对那个P当前执行的干扰。

自旋线程的必要性：**自旋线程M的存在是为了减少系统在高并发环境下的延迟和提高CPU的利用率。**关于自旋线程的必要性，可以从以下几点理解：

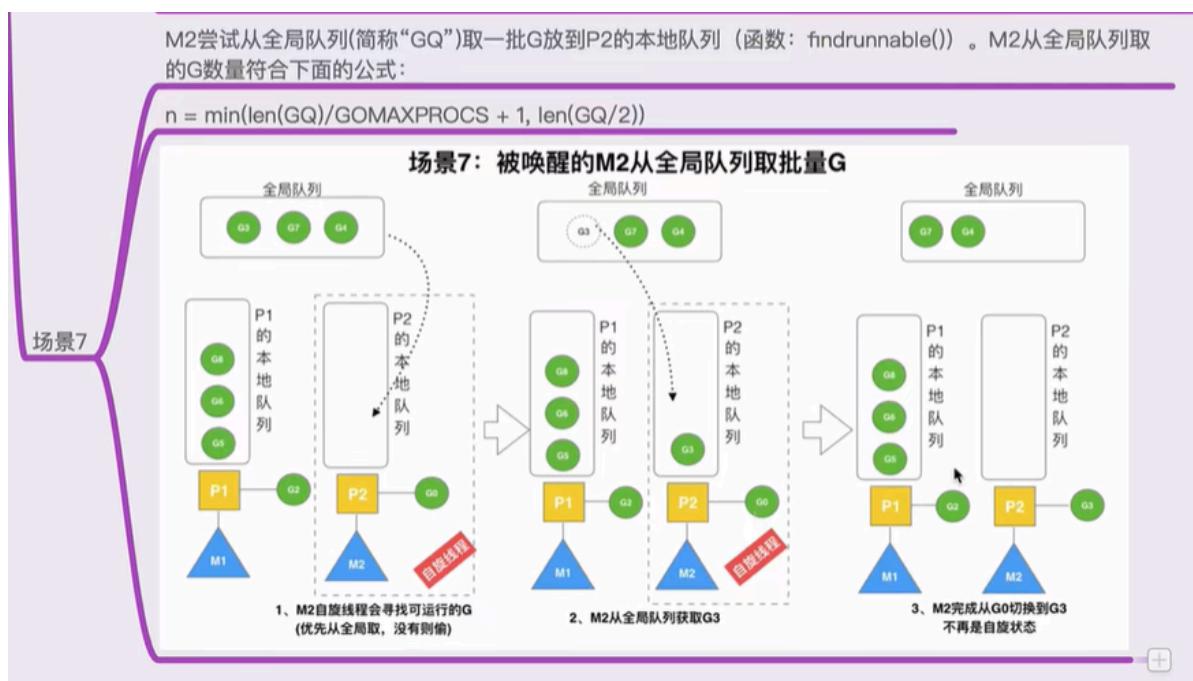
- 快速响应：**通过保持一些M在自旋状态，系统可以更快地响应新的Goroutines。当新的工作到来时，已经在自旋的M可以立即开始执行，而不需要等待操作系统调度新的线程，从而减少启动延迟。
- 平衡负载：**自旋可以帮助系统在多个P之间更平衡地分配工作负载。通过工作窃取，空闲的M可以帮助忙碌的P处理积压的工作，从而提高整体的处理效率。

- **效率与成本的权衡**: 虽然自旋会消耗CPU资源（可能看起来像是浪费），但在并发高、任务分布不均的情况下，这种成本是为了更大的效益—即减少总体的任务处理时间和提高响应速度。



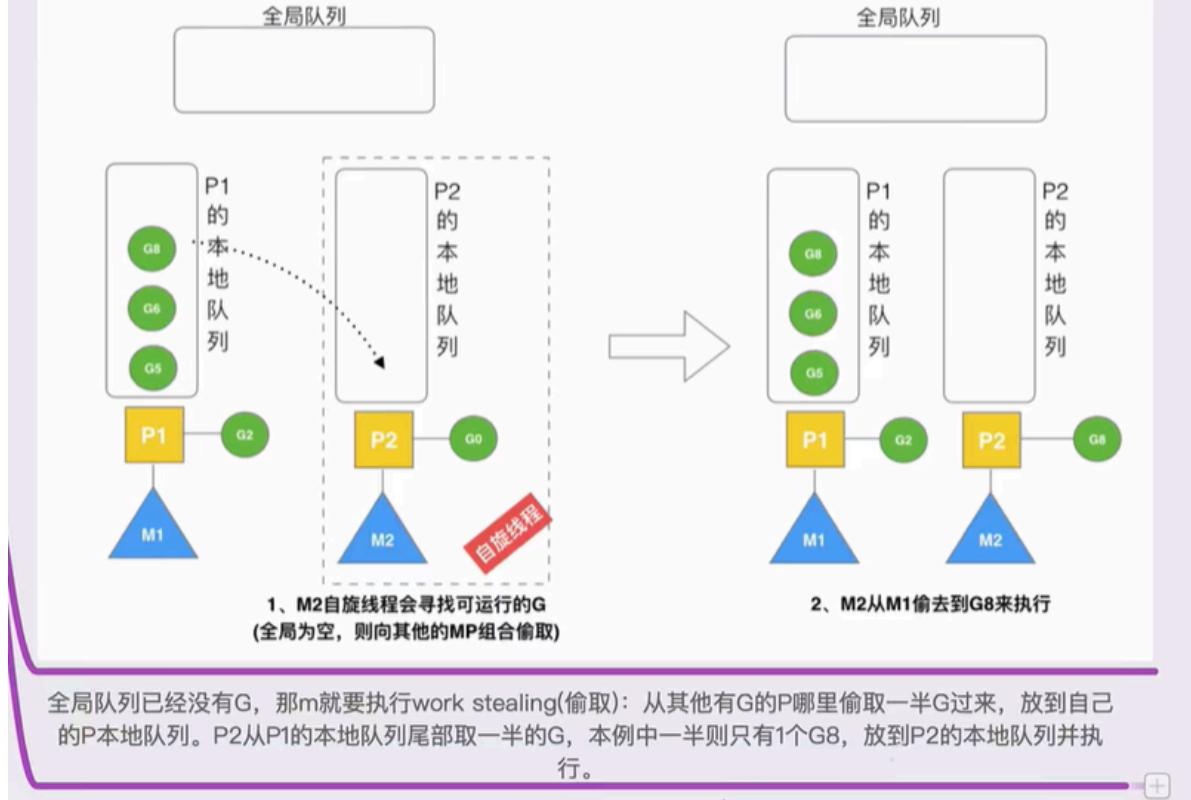
场景7

下面图中的函数有无，准确来说那个函数是从各个地方去找可执行的G。下图也是实现负载均衡，注意从全局队列取的数量，不会太多也不会太少，使得可以平均分到每一个P。



场景8

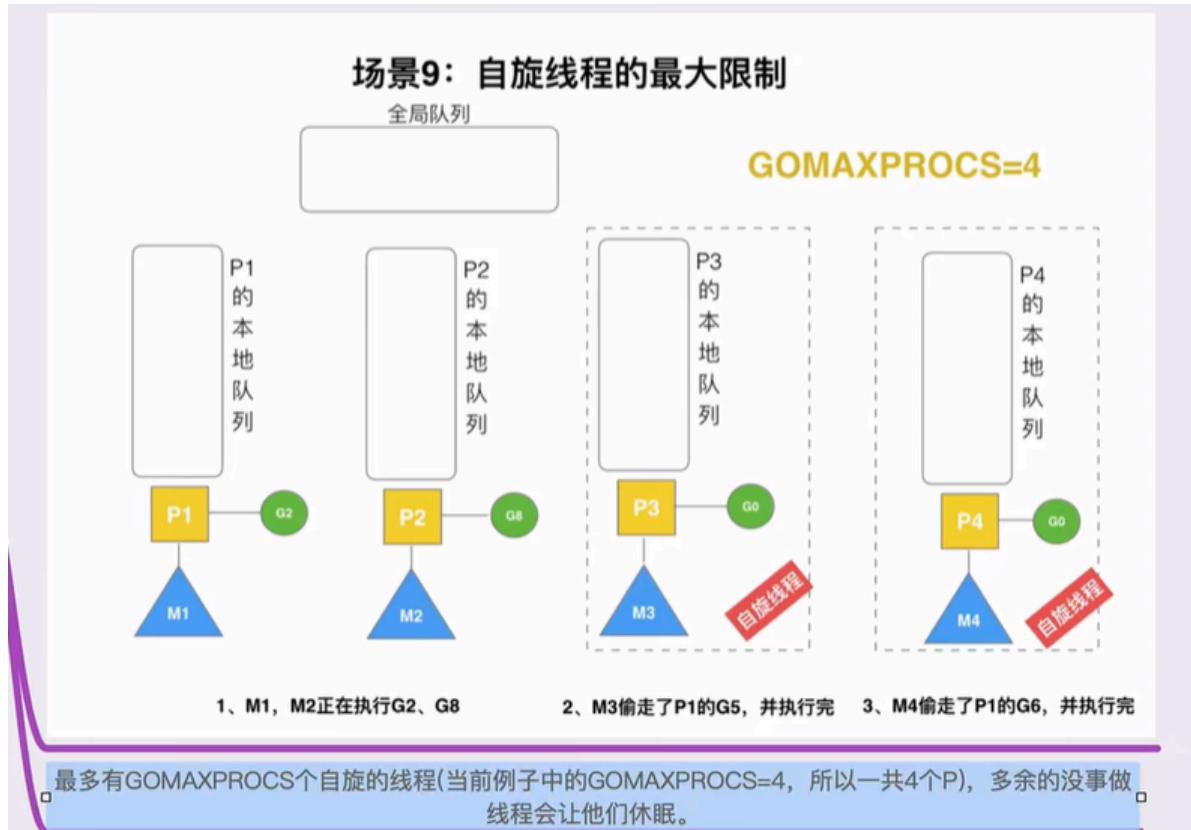
场景8：M2从M1中偷取G



场景9

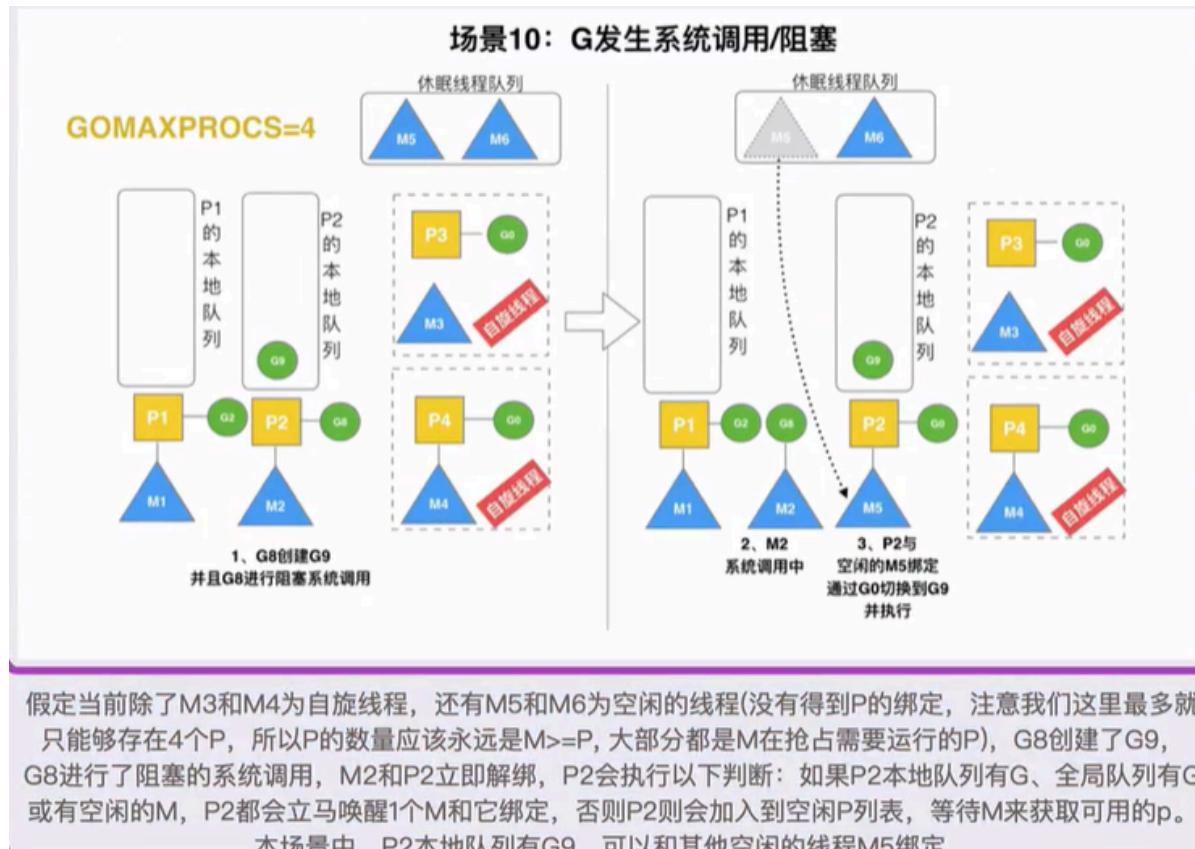
自旋线程是在运行的线程，是与P绑定的。因此自旋线程+在运行的线程<=GOMAXPROCS

场景9：自旋线程的最大限制

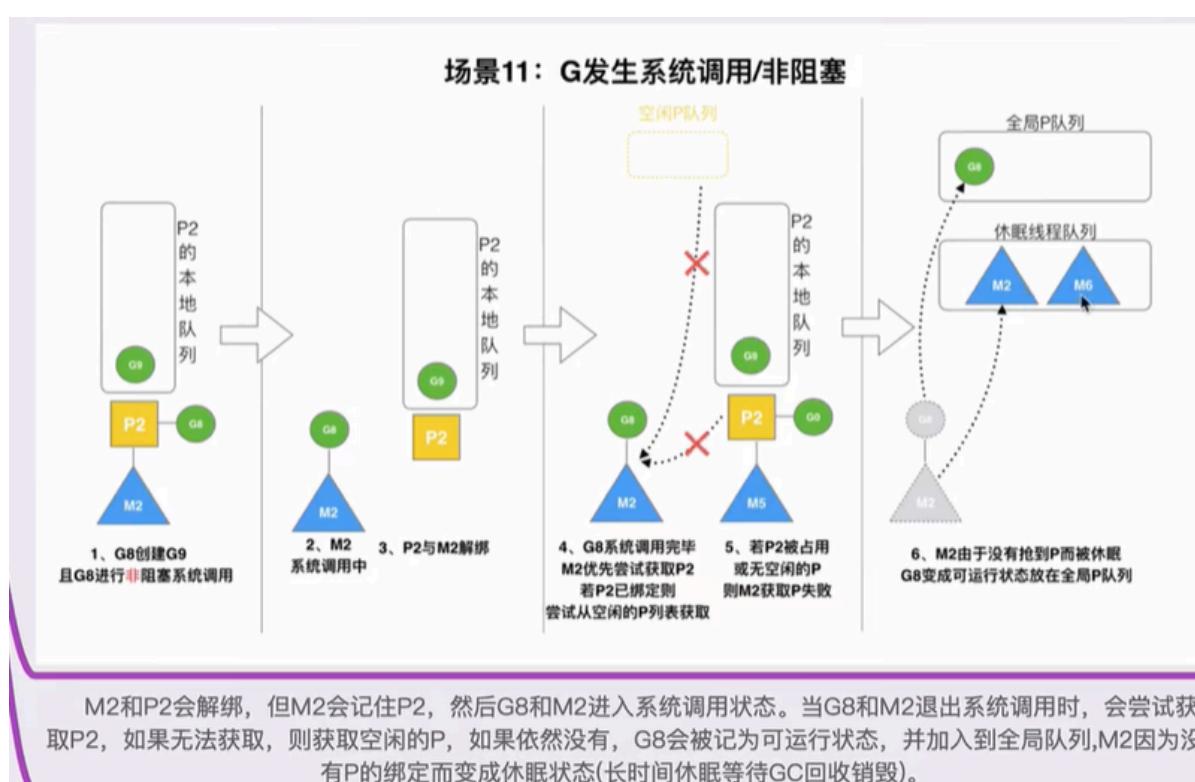


场景10

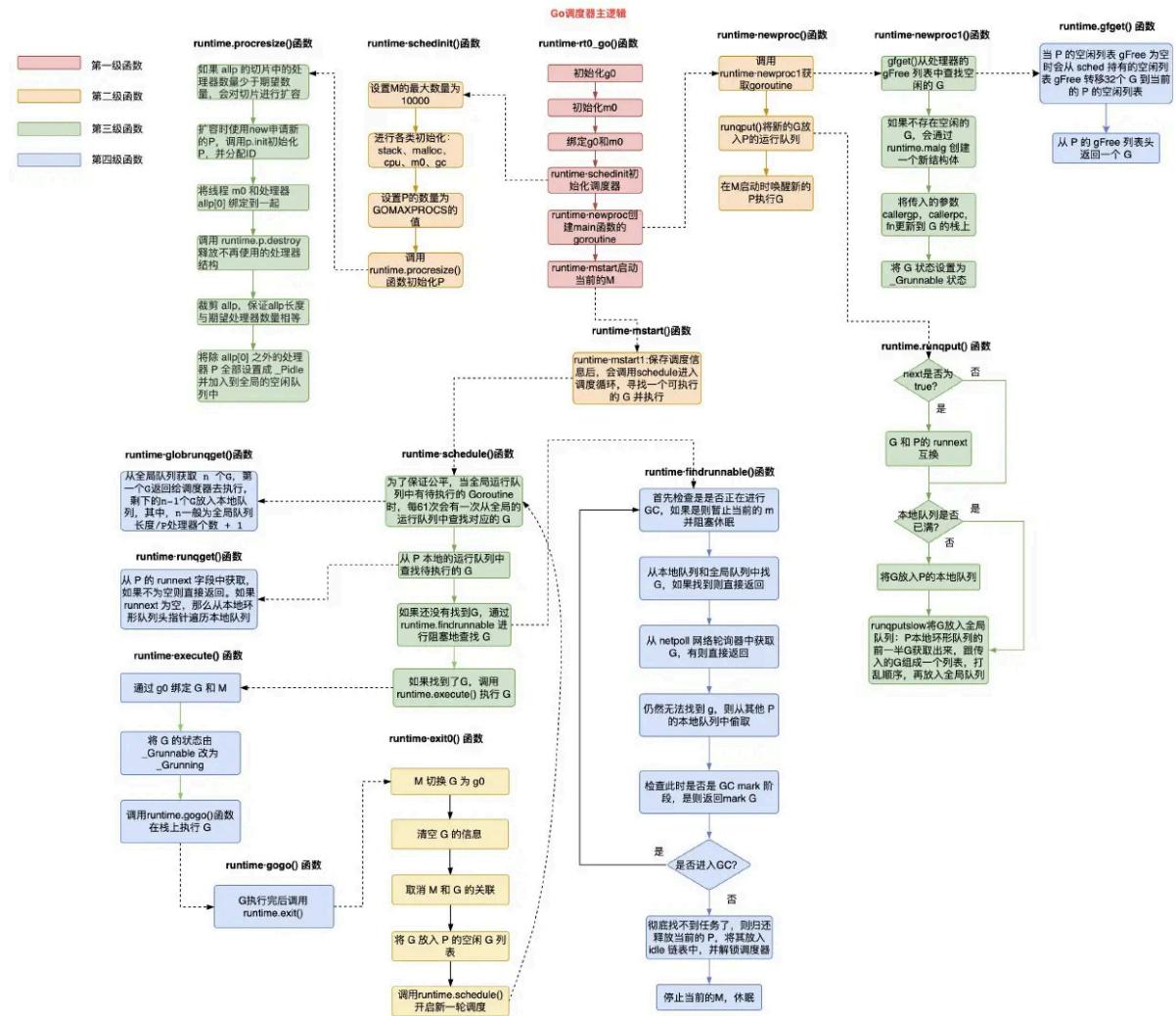
其实就是hand off，下图场景中如果P2本地队列为空，全局队列不为空，新版定的P2会从全局队列中取G来执行。



场景11



GMP源码



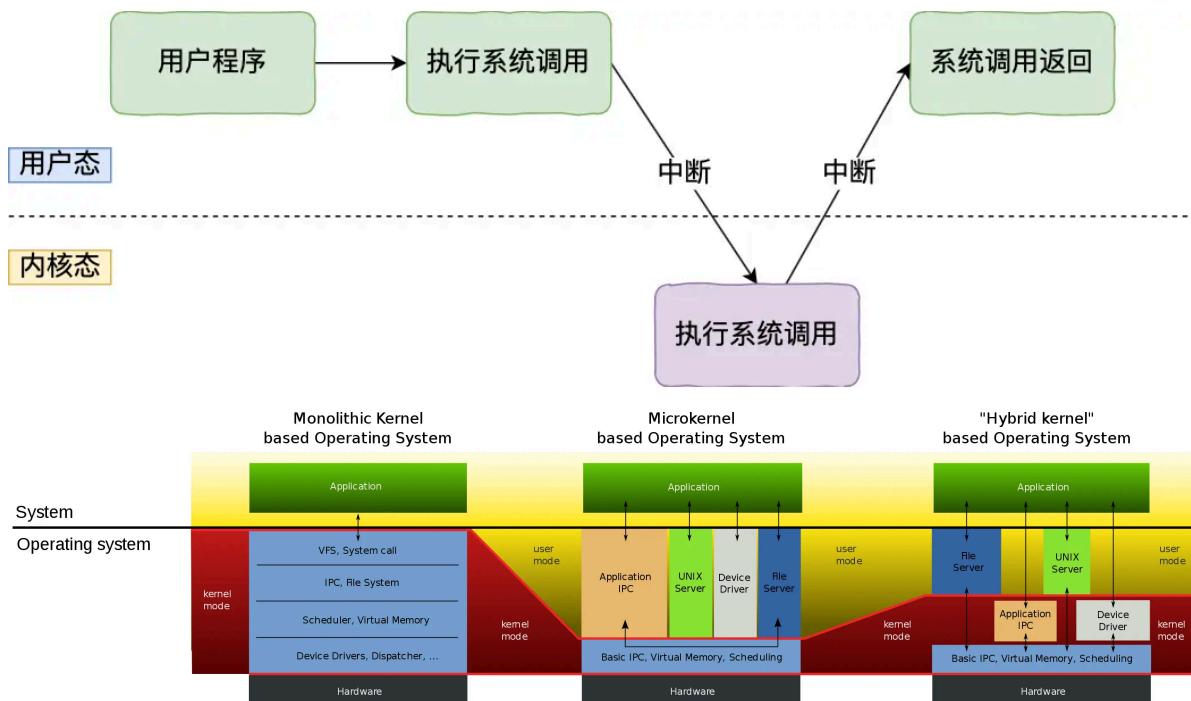
kernel和操作系统

kernel时操作系统中最核心的部分，提供了软件应用和硬件之间交互的接口，kernel的主要功能包括：

- 进程管理
- 内存管理
- 文件系统
- 设备管理
- 系统调用和安全

操作系统除了内核外，还包括支持系统运行的其他基本软件，比如：

- CLI/GUI
- 系统工具（文本编辑器，文件资源管理器）
- 服务（网络等）
- 标准库和API（提供给应用开发者）
- 应用软件

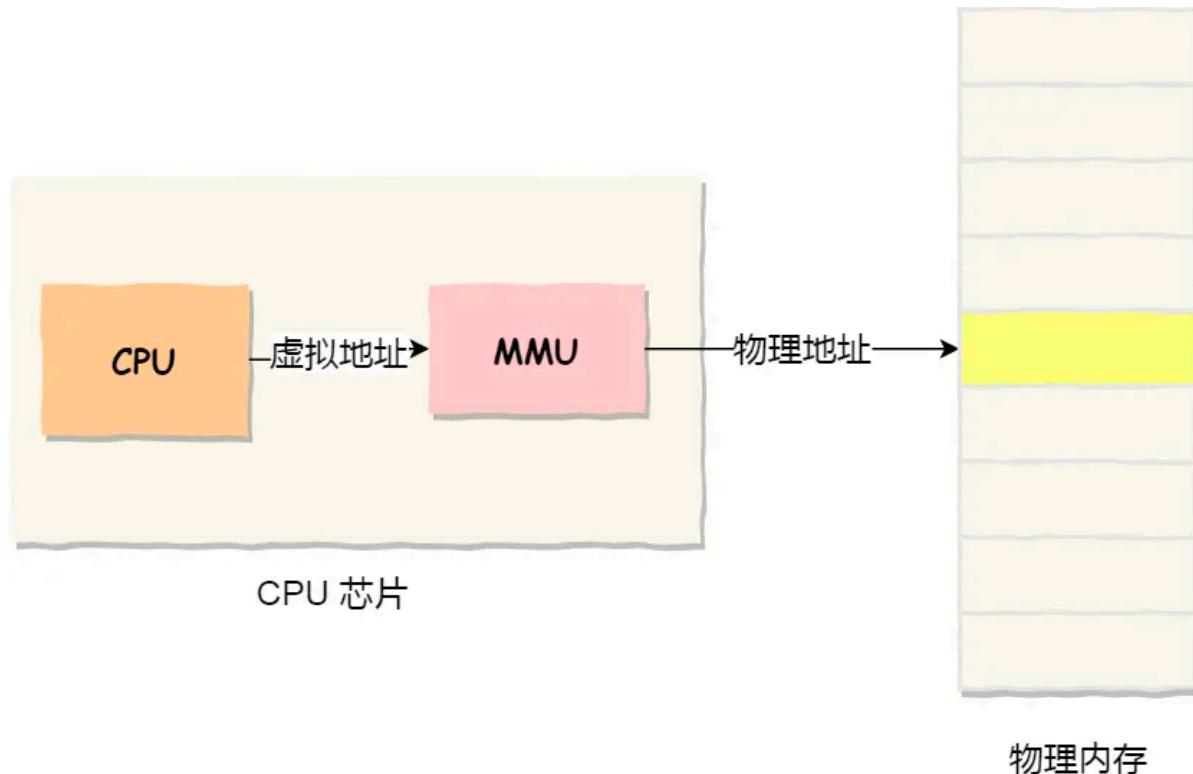


1. 宏内核，包含多个模块，整个内核像一个完整的程序；
2. 微内核，有一个最小版本的内核，一些模块和服务则由用户态管理；
3. 混合内核，是宏内核和微内核的结合体，内核中抽象出了微内核的概念，也就是内核中会有一个小型的内核，其他模块就在这个基础上搭建，整个内核是个完整的程序；

Linux的内核设计是采用了宏内核，Window的内核设计则是采用了混合内核。这两个操作系统的可执行文件格式也不一样，Linux可执行文件格式叫作ELF，Windows可执行文件格式叫作PE。

虚拟内存

单片机没有操作系统，每次写完程序需要通过工具烧录进去，程序才能跑起来。在单片机中，CPU直接操作物理内存，这对于同时运行多个程序而言是不可能，因为假如多个程序同时操作了同一个物理地址会有不确定性和安全隐患。因此，为了将每个进程地址空间隔离开来，引入了虚拟地址空间。操作系统为每个进程分配了一套完整的虚拟地址空间，让进程有拥有了整个地址空间的错觉。进程只能访问虚拟地址空间，不能直接访问物理地址空间，虚拟地址空间到物理地址空间的映射由CPU中的MMU（内存管理单元）完成。

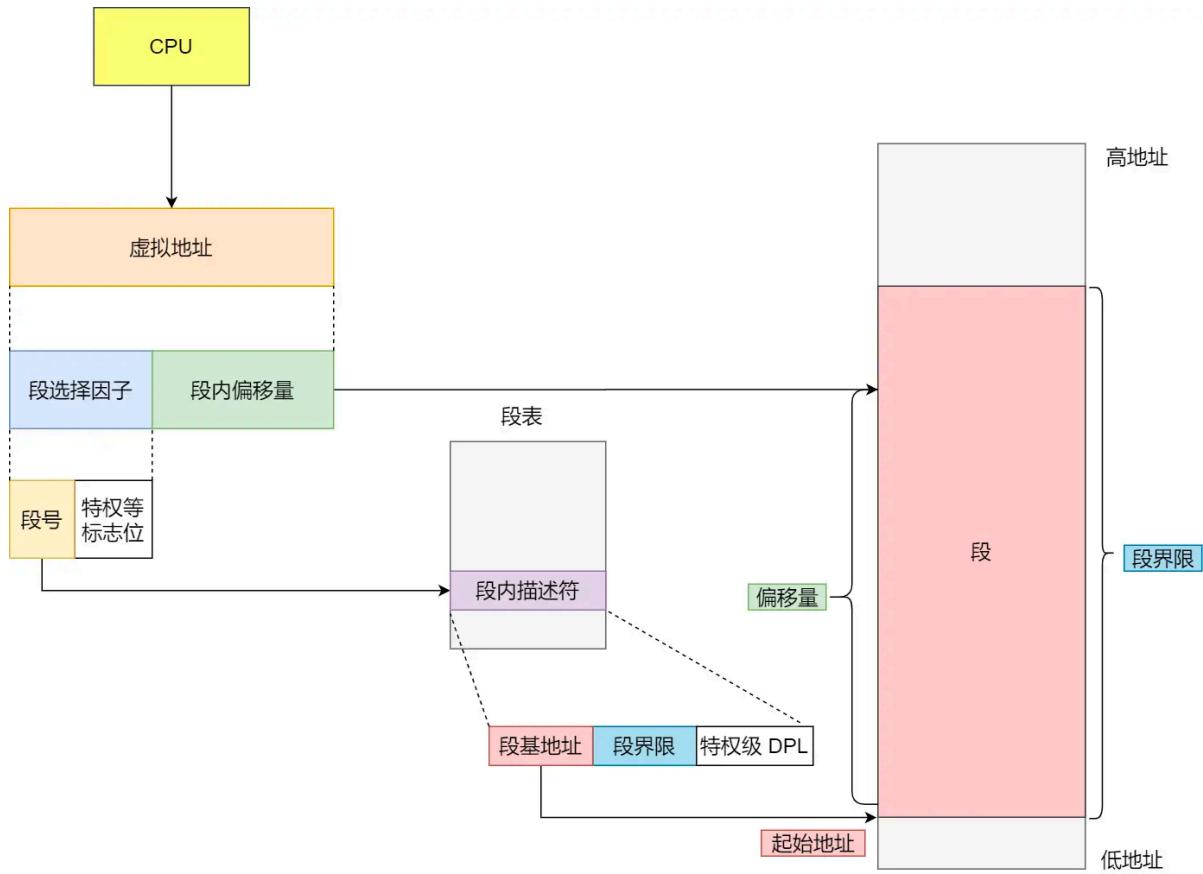


虚拟地址空间到物理地址空间的映射可分为：

- 内存分段
- 内存分页

内存分段

程序可分为不同的逻辑分段：代码段，数据段，堆段和栈段，不同的段有不同的属性。



内存分段机制下，虚拟地址由段选择因子和段内偏移量组成：

- 段选择因子通常包括段号和特权等标志位，通过段号可以在段表（GDT或LDT）中索引到唯一的段描述符。
- 段描述符包括段的基地址，界限和特权级等。
- 如果虚拟地址的段内偏移量在界限中，则通过段的基地址+段偏移量得到具体的物理地址。

全局描述符表 (GDT)：GDT是由操作系统维护的全局表，它包含了系统中所有内存段的描述符。这些段包括操作系统的代码段、数据段以及其他可能的系统级段。GDT对整个系统有效，操作系统和所有的应用程序都可以根据GDT中的描述符来访问内存。每个处理器都有一个寄存器 (GDTR) 存储GDT的位置和大小。

局部描述符表 (LDT)：LDT是为特定进程提供额外的段描述符表。它允许每个进程定义额外的、私有的内存段，这些段只对该进程可见。LDT是进程级的，每个进程可以有自己的LDT，用于存储特定于该进程的段信息。每个进程的任务状态段 (TSS) 中有一个指针指向其LDT。

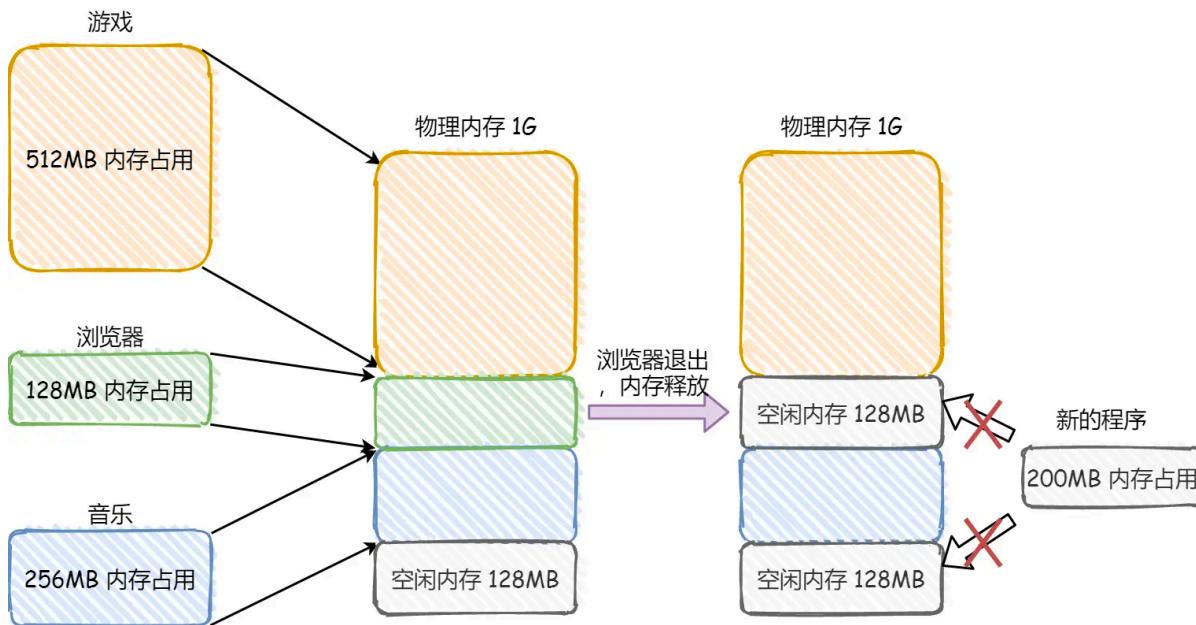
操作系统负责创建和管理每个进程的LDT。每个进程的LDT都通过一个称为LDTR的特殊寄存器来访问。当进程被调度运行时，操作系统负责加载该进程的LDT地址到LDTR寄存器。这确保了CPU在执行该进程时使用正确的LDT。虽然LDT本身是独立于GDT的，每个LDT都需要在GDT中有一个对应的描述符。这个描述符不包含LDT的实际内容，而是包含了指向该LDT的基地址和界限的信息。这个描述符被称为“LDT描述符”。当操作系统或CPU需要访问特定进程的LDT时，它会首先查找GDT中对应的LDT描述符，通过这个描述符获取LDT的物理地址，然后通过LDTR寄存器来使用这个LDT。

分段机制会将虚拟地址分为代码段，数据段，堆段和栈段。内存分段使得程序无需关心物理地址，但主要存在2个问题：

1. 内存碎片
2. 内存交换效率低

内存碎片

内存碎片分为内部内存碎片和外部内存碎片。内部内存碎片就是操作系统分配给程序远大于其所需的内存空间时，会导致未被利用的那部分内存空间被浪费，由于内存分段可以做到按需分配内存，因此内部内存碎片在内存分段中很少出现。而外部内存碎片则是内存中有许多小的，无法利用的内存，也就是不连续的内存空间，在内存分段中，由于每个段大小不一致，且动态加载和卸载，容易留下小的不连续的内存空间，造成外部内存碎片。比如下图中，浏览器退出后内存虽然还剩余512MB，但不连续，不能分配200MB的内存空间。



解决内存外部碎片可以通过内存交换，例如上图中，先将音乐占用的内存空间置换至硬盘中，再置换回来。

Swap空间（交换空间）是计算机操作系统中用于扩展物理内存（RAM）的一种机制。当系统的RAM被占满时，操作系统可以使用硬盘上的一个特定区域（即swap空间）来存储暂时不活跃的内存页面。这样做可以释放RAM中的空间，使其可以用于当前更需要的任务或进程。Swap空间是虚拟内存系统的一部分，总的可用空间为物理内存大小+Swap空间大小，但swap毕竟在硬盘中，速度会慢很多（SSD会好些）。

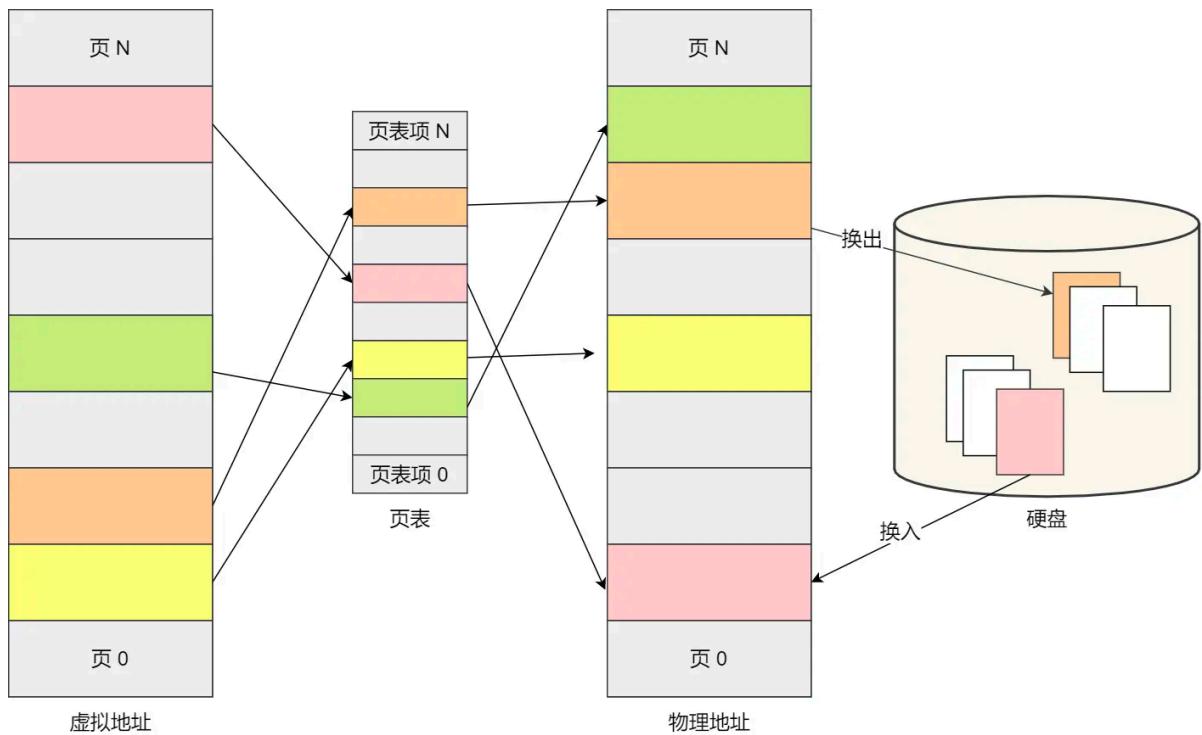
内存交换效率低

内存分段极易产生外部内存碎片，可能会导致频繁的内存交换，而硬盘的速度很慢，如果交换一个占用空间很大的程序，整个机器可能都稍显卡顿。

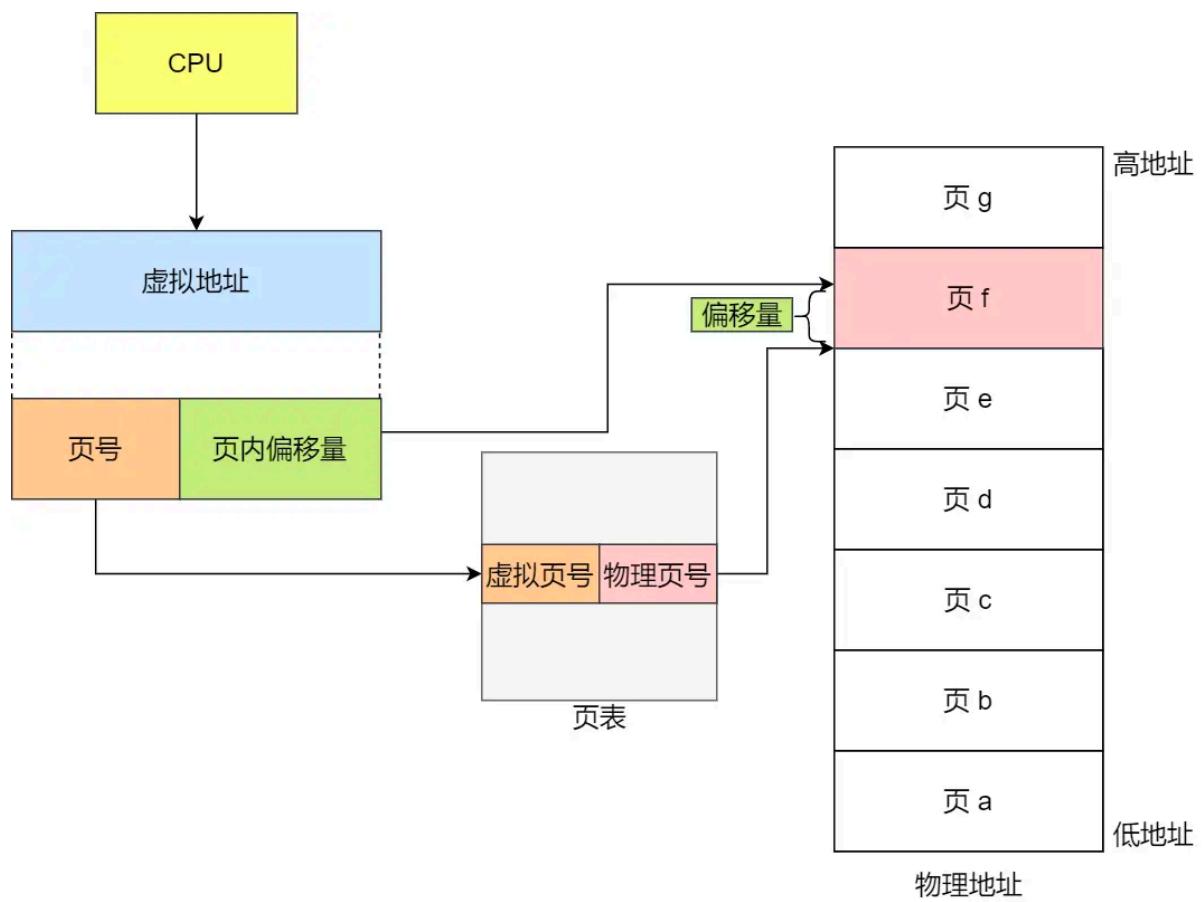
内存分页

内存分段容易出现内存碎片和内存交换效率低的主要原因是段的大小不一，因此出现了内存分页。内存分页将虚拟内存和物理内存分为一个个固定大小的页，Linux中页大小固定为4KB。内存分页中，虚拟地址和物理地址的转换通过页表和MMU完成。当进程访问的虚拟地址空间在页表中查询不到时，操作系统会产生一个缺页异常，并陷入内核空间中为其分配物理内存（可能是不存在，也可能是换出到硬盘了），更新页表，再返回用户内存空间，恢复进程运行。

正因为内存分页中页的大小是固定的，页与页之间紧密排列，不会出现微小的空隙，因此可以解决内存外部碎片问题；而段的大小不一，按需分配，动态加载和卸载，容易产生不能被利用的内存外部碎片。页分配的最小单位为1个页，因此页可能会产生内存内部碎片的问题。此外，操作系统会将最近未被使用的页换出到硬盘中（LRU），有需要再置换回来。相较于段，页的大小固定，程序每次写入的只有一个或几个页，因此内存交换效率更高。此外由于局部性原理，我们不需要一次性将进程的虚拟地址空间都映射到物理内存中，只映射正在使用的页。



单级页表



内存分页中虚拟地址由页号和页内偏移量组成，通过页号可以在页表中索引到唯一的页表项（PTE），页表项中保存了物理页号（还包括一些其他信息，比如是否可读写，是否在内存等），通过物理页号+页内偏移量可以得到唯一的物理地址。

单级页表虽然简单直观，但会导致页表存储占用大量的内存。

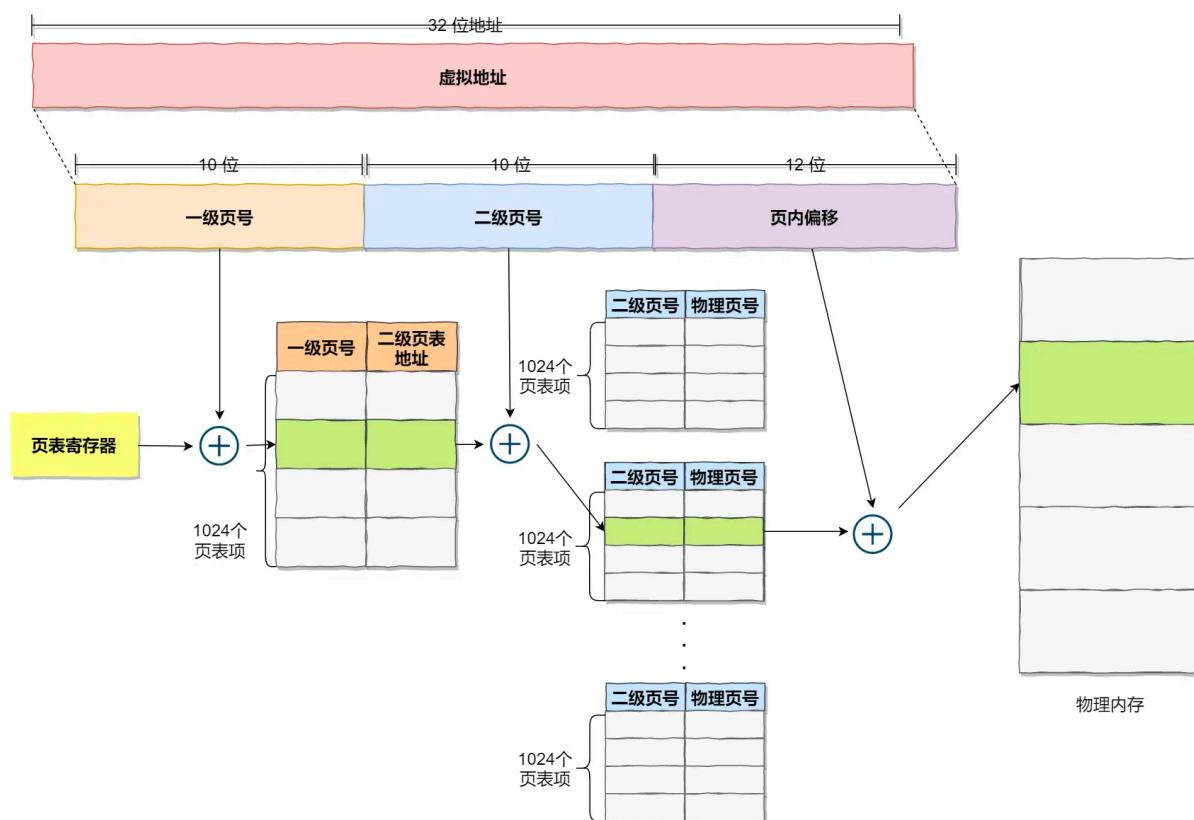
在32位的环境下，虚拟地址空间共有4GB (2^{32})，假设一个页的大小是4KB (2^{12})，那么就需要大约100万个页，每个页表项需要4个字节大小来存储，那么整个4GB空间的映射就需要有4MB的内存来存储页表。这4MB大小的页表，看起来也不是很大。但是要知道每个进程都是有自己的虚拟地址空间的，也就说都有自己的页表。那么，100个进程的话，就需要400MB的内存来存储页表，这是非常大的内存了，更别说64位的环境了。

页大小4KB是经过综合考虑，比如内存利用率，页表大小，性能，软硬件兼容等。

- 段表中的段描述符项通常比页表项少，因为段的大小不固定。
- x86架构中（32/64）使用CR3寄存器来保存进程页表的基地址，每次进行切换时，CR3都会刷新，此外CPU中的TLB也会被刷新。TLB是一个缓存，用于存储最近使用的虚拟地址到物理地址的映射，以加速地址解析过程。

多级页表

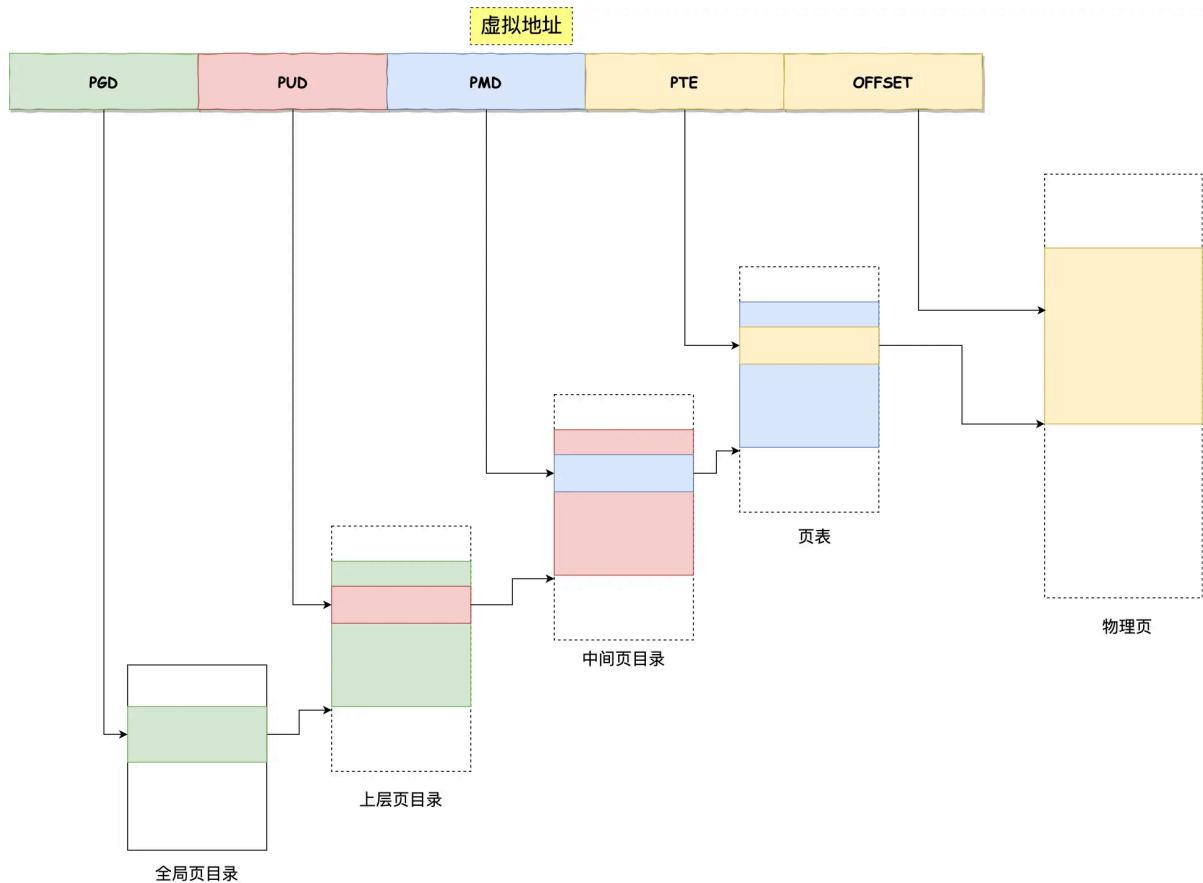
针对上述页表存储空间的问题，出现了多级页表。



多级页表中将虚拟地址分为了10位的一级页号，10位的二级页号和12位的页内偏移。一级页号用于在页表中索引出唯一的页表项，从而确定二级页表的虚拟地址（一级页表总共包含 2^{10} 个二级页表），二级页号在二级页表中确定物理页号，最后再加上页内偏移确定物理地址。

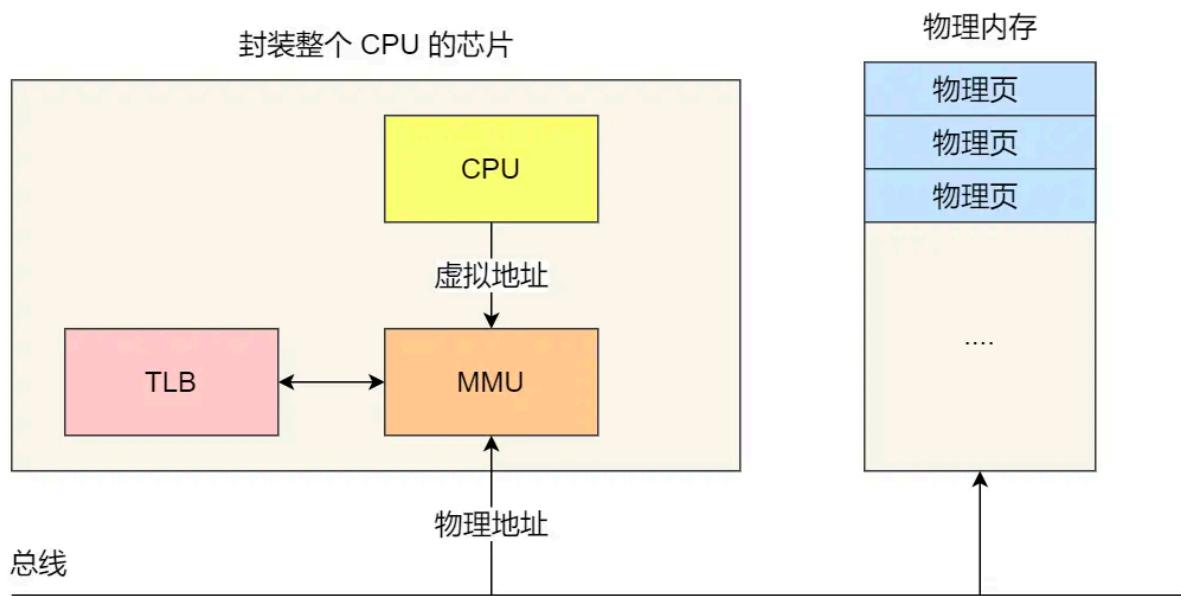
前面我们分析得到单级页表中保存单个进程32G的虚拟地址空间，需要4MB的页表大小；而上图却需要4KB+4MB（分别是1个一级页表+1024个二级页表）。但程序执行具有局部性原理，一级页表中已经包含了程序完整的虚拟地址空间，我们只需要创建需要用到的二级页表即可，比如只用到了20%的二级页表，总的内存空间占用为4KB+4MB*0.2。也就是说多级页表中节约的页表大小是暂用不到的二级页表，一级页表是需要全部创建的，因为页表需要覆盖进程的全部虚拟地址空间，因此单级页表需要创建完整的4MB页表大小。总结来说就是页表一定要覆盖全部虚拟地址空间，不分级的页表就需要有100多万个页表项来映射，而二级分页则只需要1024个页表项（32位，此时一级页表覆盖到了全部虚拟地址空间，二级页表在需要时创建）。

对于64位系统，2层分页不够，变成4层分页



TLB

多级页表降低了进程页表存储的空间大小，但使得虚拟地址到物理地址的转换更加复杂，并且程序的执行具有局部性原理，因此CPU中有一个专门保存最近使用的虚拟地址到物理地址转换的页表项的cache，也就是TLB。CPU在寻址时，首先会查找TLB，如果未命中，再去查页表，一般来说TLB的命中率很高。



段页式管理

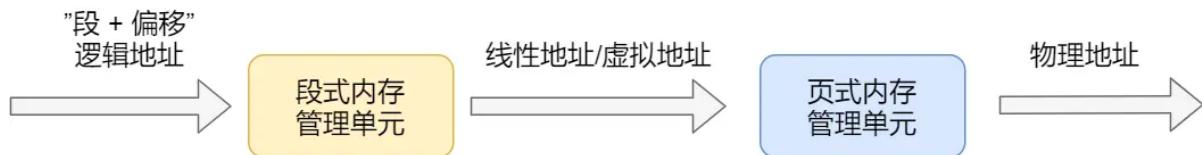
段页式管理将内存分段和内存分页结合起来，先将程序分段，再将段分为固定大小的页，程序地址由段号，段内页号和页内偏移构成：

- 每个进程一个段表，段号索引到唯一的段描述符，包含了页表地址（也就是每个段有一个页表）
- 段内页号索引到唯一的页表项，页表项包含物理页号，加上页内偏移获取物理地址。

段页式管理虽然增加了硬件成本和系统开销，但提高了内存利用率。

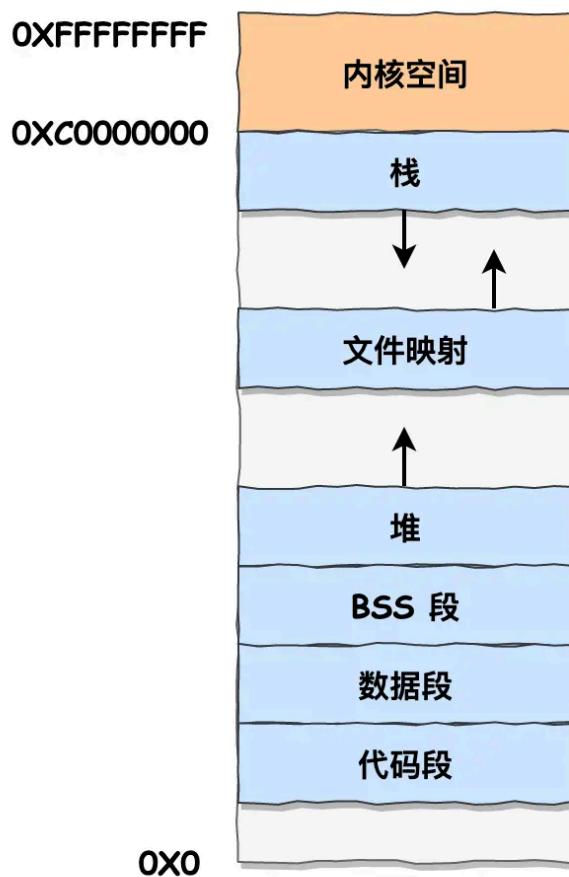
Linux内存布局

Intel的内存管理：逻辑地址（段前）->线性地址（页前）->物理地址



Linux系统中的每个段都是从0地址开始的整个4GB虚拟空间（32位环境下），也就是所有的段的起始地址都是一样的。这意味着，Linux系统中的代码，包括操作系统本身的代码和应用程序代码，所面对的地址空间都是线性地址空间（虚拟地址），这种做法相当于屏蔽了处理器中的逻辑地址概念，段只被用于访问控制和内存保护（特权级，界限等）。

段提供了基于长度和特权级的保护，但页提供了更细粒度的保护，允许操作系统控制每个页的访问权限。在现代操作系统中，如Windows和Linux，通常使用页式管理作为主要的内存管理机制，而段式管理则更多地用于兼容性和特定的底层任务（如在x86架构中处理特权级）。

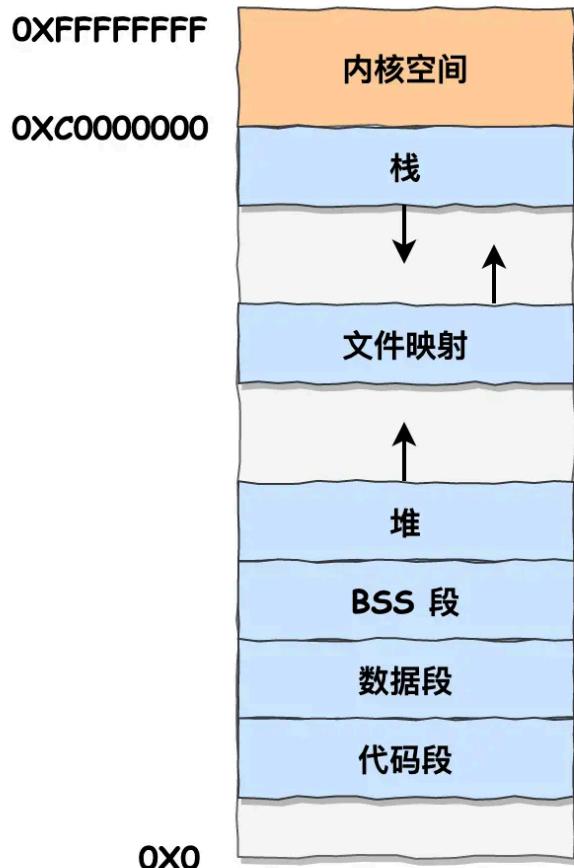


用户内存空间如上：

- 代码段，包括二进制可执行代码；
- 数据段，包括已初始化的静态常量和全局变量；
- BSS 段，包括未初始化的静态变量和全局变量；
- 堆段，包括动态分配的内存，从低地址开始向上增长；
- 文件映射段，包括动态库、共享内存等，从低地址开始向上增长（跟硬件和内核版本有关 (opens new window)）；

- 栈段，包括局部变量和函数调用的上下文等。栈的大小是固定的，一般是8MB。当然系统也提供了参数，以便我们自定义大小（注意是从高地址到低地址，历史原因，空间利用效率，安全，区分堆）；

代码段下面还有一段内存空间的（灰色部分），这一块区域是「保留区」，之所以要有保留区这是因为在大多数的系统里，我们认为比较小数值的地址不是一个合法地址，例如，我们通常在C的代码里会将无效的指针赋值为NULL。因此，这里会出现一段不可访问的内存保留区，防止程序因为出现bug，导致读或写了一些小内存地址的数据，而使得程序跑飞。在这7个内存段中，堆和文件映射段的内存是动态分配的。比如说，使用C标准库的malloc()或者mmap()，就可以分别在堆和文件映射段动态分配内存。



Linux中进程虚拟地址空间分布如上：

- 代码段
- 数据段
- bss段
- 堆段
- 文件映射段
- 栈段

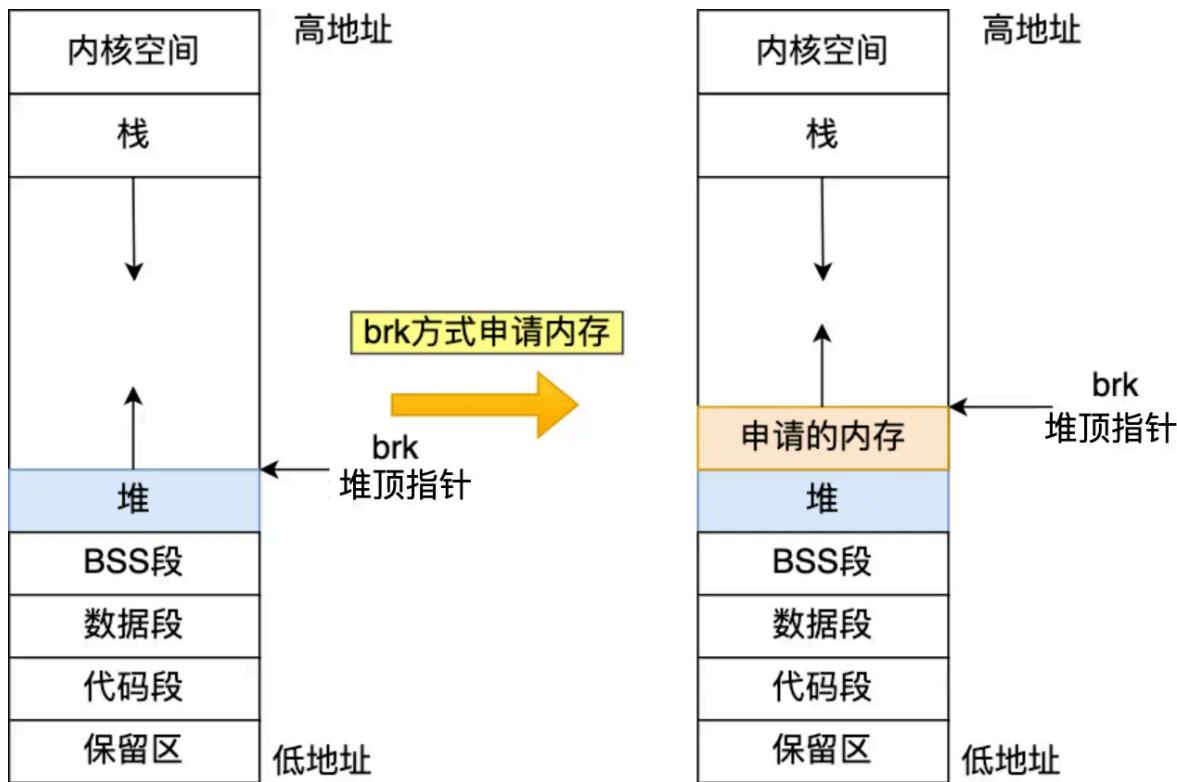
其中堆段和文件映射段的内存是动态分配的，分别由malloc和mmap。

malloc是如何分配内存的？

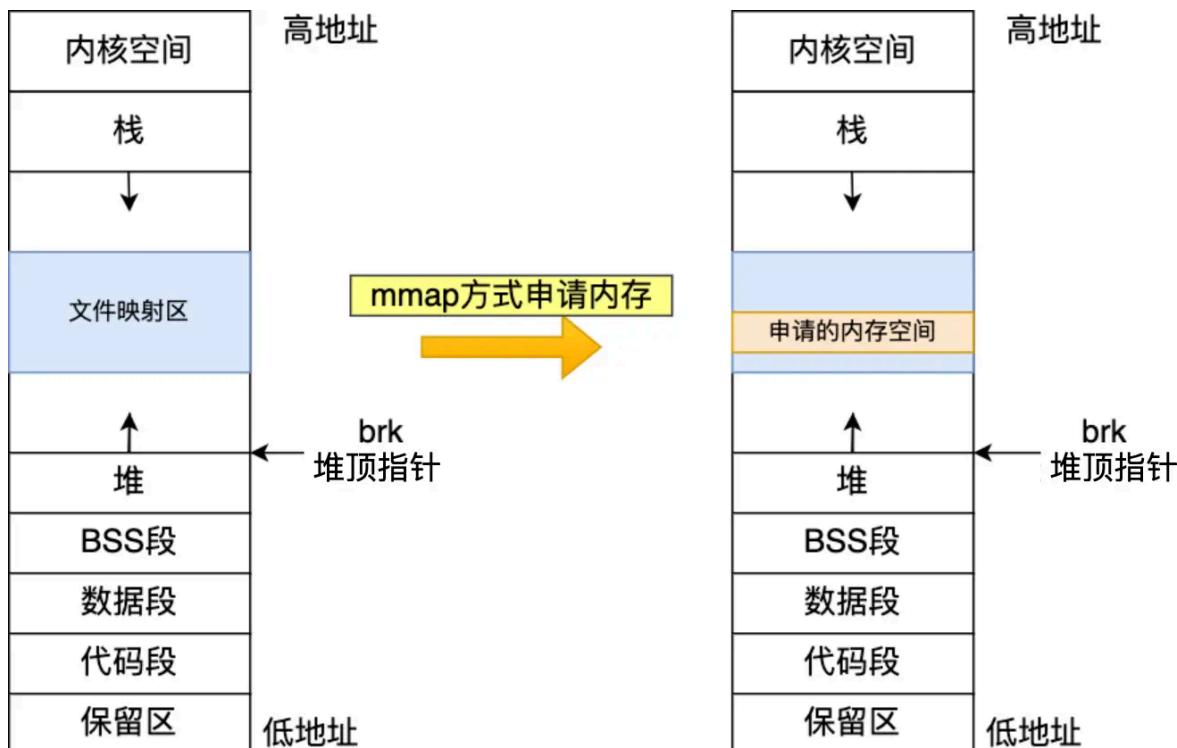
malloc非系统调用，是一个c库函数，用于动态分配内存，malloc根据需要分配的内存大小，使用两种系统调用方法分配内存（取决于glibc中的定义）：

- brk()系统调用从堆中分配内存（128KB以下）
- mmap()系统调用从文件映射区中分配内存（128KB以上）

brk()系统调用比较简单，通过移动堆顶指针获得分配的内存空间。



`mmap()`系统调用则是在文件映射段使用私有匿名映射（私有：这部分内存只能由该进程所访问，匿名则是该部分内存与任何文件系统中的文件无关，不是通过文件映射分配，而是操作系统直接分配）。



`malloc`分配的内存为虚拟内存，如果这部分内存未被访问，操作系统不会为其在物理内存中进行分配；当访问这部分内存时，通过查找页表发现没有存在于物理内存中，产生缺页中断，为其进行物理内存分配。

malloc(1)会分配多大的内存

`malloc(1)`从语义上来说是申请1字节的内存，但操作系统会分配更大的内存，具体取决于内存管理器。

- 内存对齐
- 内存管理的开销

- 内存碎片

内存对齐是将数据元素的地址安排在某些数值的整数倍，例如一个数据类型需要4字节对齐，其起止地址应该是4的倍数，也就是二进制中后两位为00
内存对齐可以提升性能和数据访问效率（时钟周期之类的）

```
struct Example {  
    char a;           // 占用1字节  
    int b;           // 通常占用4字节  
};
```

上述数据结构为了对齐4字节，a和b之间会插入一些填充字节，结构体最后的大小可能是8字节，而非5字节。

free释放完内存，会将内存还给操作系统吗

- 如果malloc是调用的brk()分配内存，free后这部分内存不会归还给操作系统，缓存在malloc的内存池中，留着下次用
- 如果malloc是调用的mmap()分配内存，free后这部分内存会直接归还给操作系统，内存得到释放

为什么不全部使用mmap来分配内存

mmap作为系统调用，用户在申请内存时需要陷入至内核态，状态转换需要开销。此外，mmap分配的内存free后直接归还给内存，因此第一次访问mmap分配的虚拟地址时，容易产生缺页中断。

而brk申请的内存free后不会归还给内存，因此下次申请时可以直接从内存池中取出使用，减少系统调用次数和缺页中断次数。

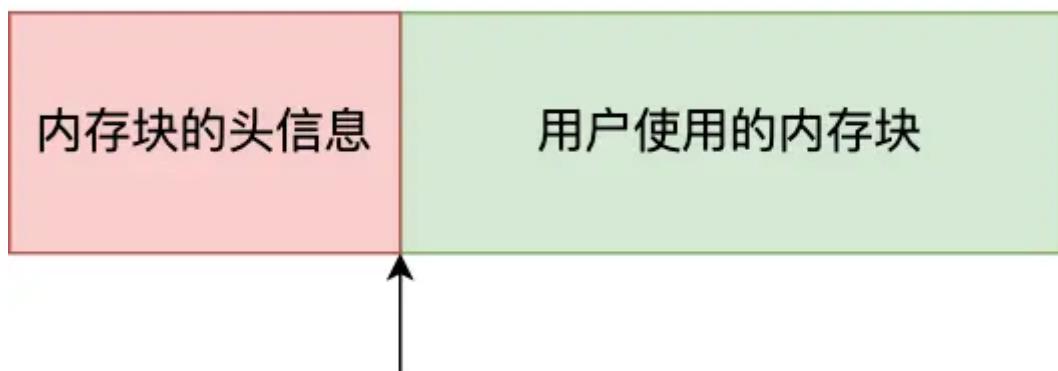
为什么不全部使用brk来分配内存

brk只能在堆顶进行处理，也就是只能线性的增加或减少内存地址，对于内存块大小不一样的应用和非连续的内存，brk容易产生内存碎片。

mmap则支持非连续的内存分配，因此两者结合起来用，大一点的分配用mmap，小一点的用brk。

free()的参数仅为一个地址，如何知道释放的内存大小

0x10



分配的内存块左边便宜16字节是内存块的描述信息，包含内存块大小。

在4GB内存物理内存的机器上申请8GB内存

32位系统

申请的内存虽然是虚拟内存，为每个进程分配独立完整的地址空间。但在第一次访问这些虚拟地址时，会产生缺页中断，操作系统为其分配物理内存。32位系统的机器，最多能支持寻址到4GB，但其中1GB用于内核地址空间，3GB为用户地址空间，因此在32位机器上，申请8GB内存会出现分配失败（申请4GB内存也会出现分配失败）。

64位系统

64位系统所支持寻址的地址大小是远大于8GB的，在64位系统，物理内存为4GB的机器上进行8GB内存申请，操作系统会为其申请虚拟内存，只要不读写这部分内存，就不会为其申请物理内存。

但需要对overcommit_memory这个参数进行修改，避免申请失败：

- 如果值为 0 (默认值) , 代表: Heuristic overcommit handling, 它允许overcommit, 但过于明目张胆的overcommit会被拒绝，比如malloc一次性申请的内存大小就超过了系统总内存。
Heuristic的意思是“试探式的”，内核利用某种算法猜测你的内存申请是否合理，大概可以理解为单次申请不能超过free memory + free swap + pagecache的大小 + SLAB中可回收的部分，超过了就会拒绝overcommit。
- 如果值为 1, 代表: Always overcommit. 允许overcommit, 对内存申请来者不拒。
- 如果值为 2, 代表: Don't overcommit. 禁止overcommit。

通过开启上述参数可以申请很大的内存，比如在64位系统中申请127t内存（不能申请128t（64位机器中内核空间和用户空间各占128t），进程的运行也需要虚拟内存）。但是如果沒有开启swap的话可能会导致操作系统使用OOM杀掉这个进程，虽然没有访问申请内存中的虚拟地址，但申请的过程中也需要用到物理内存，比如保存这些虚拟内存的元数据之类的。

swap

进行后台内存回收的守护进程kswapd。

Linux提供了两种不同的方法启用Swap，分别是Swap分区（Swap Partition）和Swap文件（Swapfile）：

- Swap分区是硬盘上的独立区域，该区域只会用于交换分区（换出的匿名内存），其他的文件不能存储在该区域上，我们可以使用swapon -s命令查看当前系统上的交换分区；
- Swap文件是文件系统中的特殊文件，它与文件系统中的其他文件也没有太多的区别；

Linux中虚拟地址空间分为用户地址空间和内核地址空间。



32位操作系统总的虚拟地址空间为 2^{32} , 共4G, 其中内核空间1G, 用户空间占3G。64位操作系统, 内核空间和用户空间分别位于最开始的和结束的128t, 中间的内存未定义。

一个进程可以创建多少个线程

- 进程的虚拟地址空间上限：创建线程要分配栈空间，线程越多，进程虚拟地址空间占用越大
- 系统参数限制：内核参数没有限定单个进程能创建多少个线程，但限制了整个系统的最大线程数

```
$ ulimit -a
-t: cpu time (seconds)          unlimited
-f: file size (blocks)          unlimited
-d: data seg size (kbytes)      unlimited
-s: stack size (kbytes)         8192
-c: core file size (blocks)     0
-m: resident set size (kbytes)  unlimited
-u: processes                   31761
-n: file descriptors           1024
-l: locked-in-memory size (kbytes) 65536
-v: address space (kbytes)      unlimited
-x: file locks                  unlimited
-i: pending signals             31761
-q: bytes in POSIX msg queues  819200
-e: max nice                   0
-r: max rt priority             0
-N 15:                           unlimited
```

可以通过上述命令查看堆栈空间，上述显示单个堆栈所用空间为8M，32位操作系统所支持的最大堆栈数为3G/8M，大概是300+个线程。

下面3个系统参数也会影响单个进程所能创建的最大线程数目：

```
# pyq @ 0x505951-Y7000P in ~ [23:09:45]
# 系统支持的最大线程数
$ cat /proc/sys/kernel/threads-max
63522

# pyq @ 0x505951-Y7000P in ~ [23:09:47]
# 可以分配的最大进程号
```

```
$ cat /proc/sys/kernel/pid_max  
4194304  
  
# pyq @ 0x505951-Y7000P in ~ [23:10:10]  
# 每个进程可以拥有的最大内存映射段数  
$ cat /proc/sys/vm/max_map_count  
65530
```

PID (Process ID): 这是分配给每个进程的唯一标识符。在Linux中，每个进程都有一个唯一的PID。

TID (Thread ID): 在Linux中，线程也被赋予一个唯一的标识符，称为线程ID或TID。对于多线程应用，每个线程都有其自己的TID，但它们共享相同的PID，即它们所属进程的PID。

为什么物理内存只有2G，进程的虚拟内存却可以使用25T呢？

因为虚拟内存并不是全部都映射到物理内存的，程序是有局部性的特性，也就是某一个时间只会执行部分代码，所以只需要映射这部分程序就好。

传统的LRU算法存在以下2个问题，导致缓存命中率下降：

- 预读失效：操作系统在读磁盘中的数据时通常会多读一些数据至缓存中，如果这部分数据最后没访问到则会造成预读失效（传统的LRU算法只有一个链表，并且将页面直接放置链表头部，可能会淘汰一些热点数据）。
- 缓存污染：批量读数据时把一些热点数据换出了（LRU是最近最少使用，比如我批量读入一波数据到缓存中，这部分数据可能只用一次，但此时缓存不够了，需要换出一部分数据，此时就可能把一些热点数据换出了）

Redis使用LFU算法（最不常使用）来进行缓存淘汰，从而减少缓存污染造成的影响（Redis没有预读机制）。

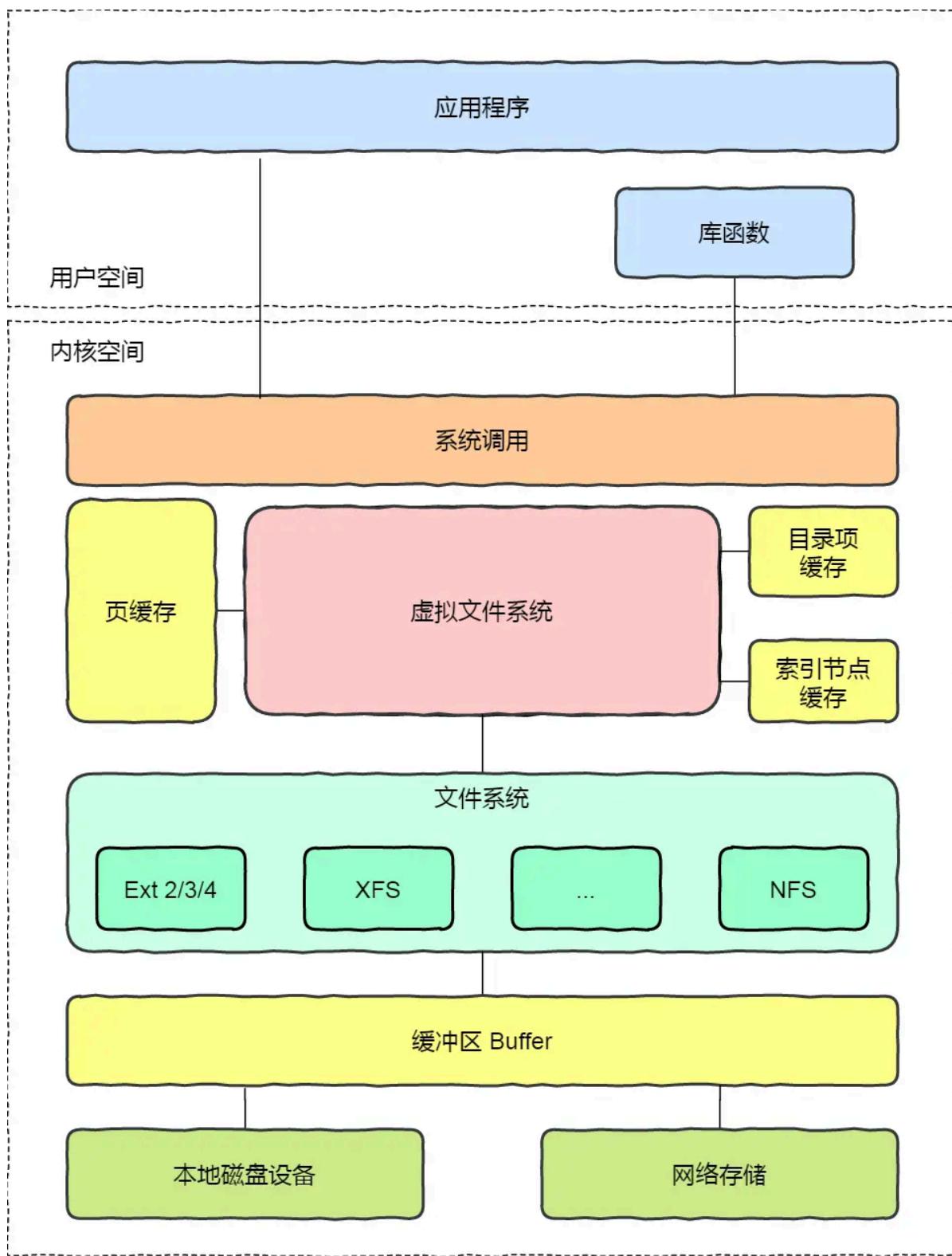
LFU相较于LRU，进行缓存淘汰时，会淘汰从加入缓存以来访问次数最低的页面。LFU的实现通过为每个页面维护一个计数器，可以通过一个或多个双向链表（每个链表保存不同计数的页面，当页面被访问时，将该页面转移到对应计数次数的链表）和哈希表（存储页面指针，方便快速查找）。每次淘汰时选择计数器最低的链表中的最旧项即可。
LFU也是有缺点的，比如实现会比较复杂，以及启动偏差（刚加入的页面虽然计数很少，但未来可能会变得热门）

MySQL和Linux则通过改进LRU算法来减少预读失效和缓存污染带来的影响。

Linux和MySQL中的缓存机制

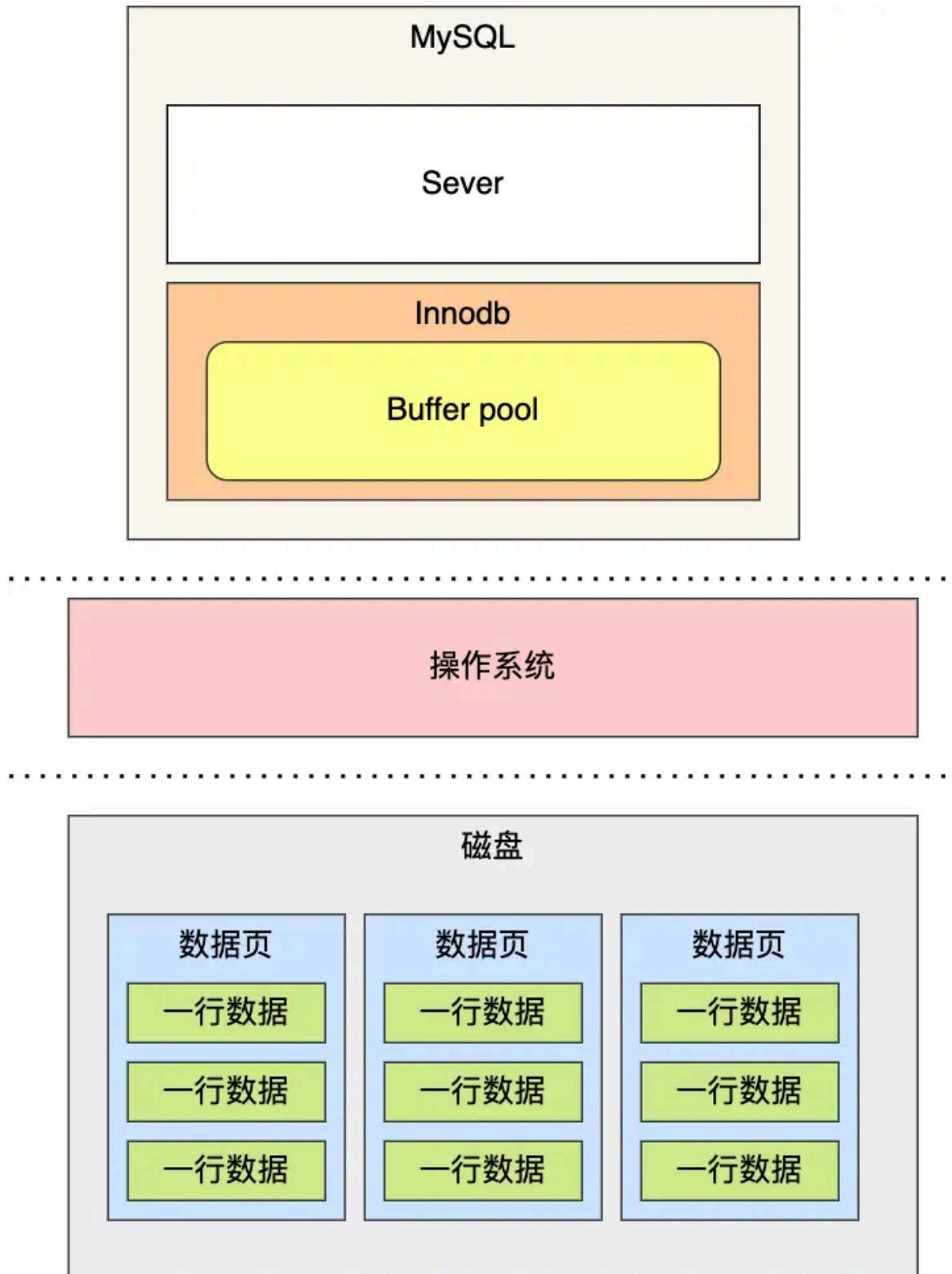
Linux

用户程序在第一次读取硬盘数据时，Linux系统会将硬盘数据进行缓存在系统的内存中（pagecache）。当用户程序再次读取这些数据时，先在缓存中查找，如果存在则直接返回，也就是缓存命中；否则对磁盘中的数据进行读取并进行缓存。



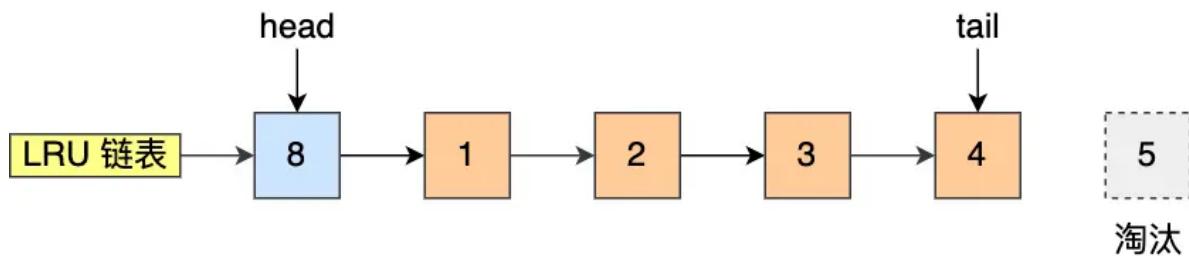
MySQL

MySQL的数据是位于磁盘中的，如果每次都去磁盘中读取数据会很慢，因此MySQL提供了innodb，innodb使用一个位于内存中的缓存池（buffer pool）存储数据和索引。当读数据时首先查找buffer pool，如果存在即缓存命中直接读取，否则从磁盘中进行加载。写数据采用延迟写的机制，即写数据先在buffer pool中进行操作，等到适当的时机再批量写入磁盘。bufferpool中被修改但未写入磁盘的数据称为脏数据，innodb会定期将这些脏数据刷新进磁盘，保持一致性和同步。



LRU

但PageCache和buffer pool都是有大小限制的，当容量满时就需要淘汰掉页，如何设计淘汰机制显得很关键。比较常用的算法是LRU算法，即淘汰最近最少使用的页。LRU通过维护一个链表，链表头存储最近访问过的页面，而表尾存储最近最不常访问的页面。当用户访问数据时，如果缓存命中，则将面中页移动至链表头。如果没有面中则需要将磁盘中的数据读入缓存，同时将该页面置于链表头，如果此时容量已经满了，还需要淘汰掉链表最后一个页面。



传统的LRU算法并未被MySQL和Linux所使用，因为传统的LRU算法会不可避免的出现预读失效和缓存污染两个问题。

预读失效

这里再详细的描述以下预读机制，预读机制是为了减少磁盘读写次数提高磁盘I/O吞吐量，比如当用户程序读取磁盘上一个block（block是磁盘读取的最小单位，4KB）时，由于局部性原理，操作系统往往会多读取额外的几个block进入缓存中。这样用户程序访问到这部分数据时可以直接从缓存中读取，而不用再去读取磁盘了。

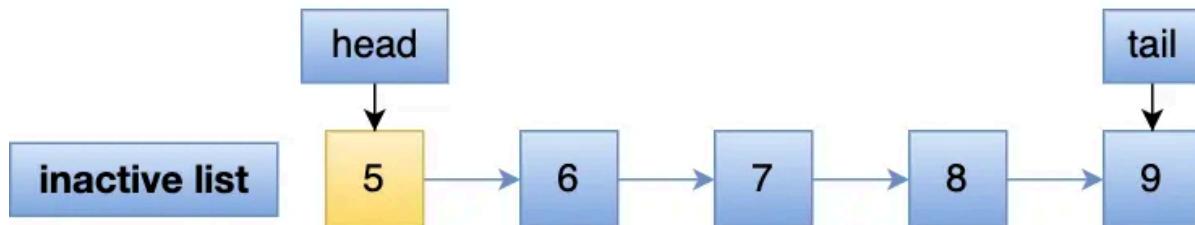
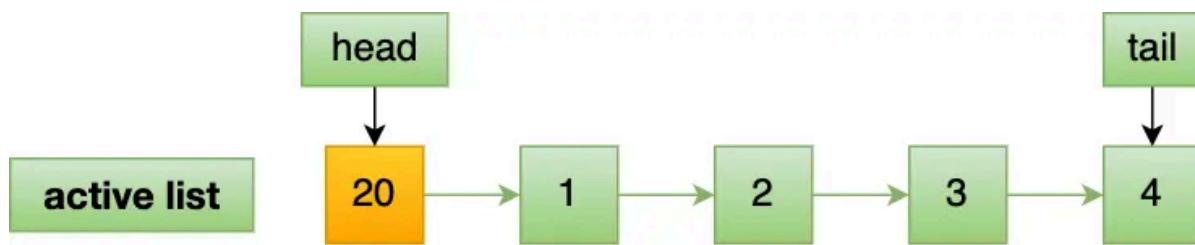
但预读机制会带来一些问题，传统的LRU算法中，预读的页会放入链表头部，如果这些页后续未被访问到，会大大降低缓存命中率。此外，如果缓存容量已满，预读的页还会淘汰链表尾部的旧页，如果这些旧页是热点数据，也会降低缓存命中率。

Linux中对LRU算法的改进

区别于传统LRU算法只使用一个链表来维护页面，Linux则使用2个链表（可以想成将之前的1个链表从中点分成了2个链表）：

- 活跃LRU链表：存储最近最常使用的页
- 不活跃LRU链表：存储最近最不常使用的页，淘汰页面选择该链表尾部的页面

当操作系统预读一些页面后，如果这些页面未被使用，操作系统并不会将这些页面放置到活跃LRU链表的头部，而是选择不活跃的LRU链表头部。当这些页面被使用时，再将其放置LRU链表的头部，如果活跃LRU链表容器已满，则会将尾部的页面放置不活跃LRU链表头部。这样即使预读的页一直未被读取，也会先热点数据淘汰。

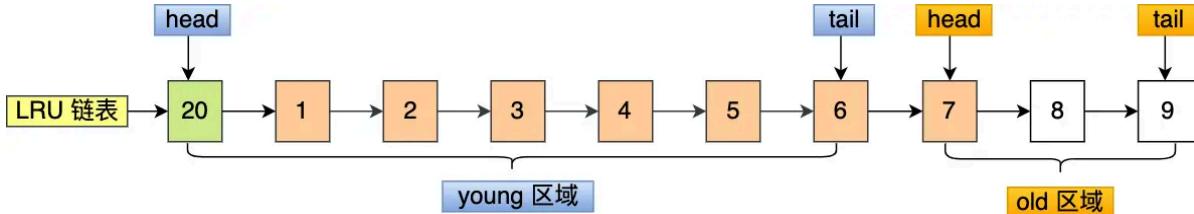


MySQL对LRU算法的改进

MySQL的改进原理和Linux类似，只不过MySQL仍采用一个链表，但将链表分成了2个区域，维护2对头尾指针。两个区域的比例默认是63：37：

- young区域：对应活跃LRU链表
- old区域：对应不活跃LRU链表

预读的页先放至old区域的头部，如果被访问，则升级到young区域头部。如果一直未被访问，也会先于young区域中的热点数据被淘汰。



缓存污染

缓存污染主要是针对批量读取数据而言的，当我们批量从磁盘中读取访问数据时（比如mysql从一个表中查询数据，即使结果集很小，但由于需要进行比对，也会读取访问大量数据），这些数据会被放置活跃LRU链表或young区域，如果此时缓存比较有限，就会导致热点数据被换出，后续访问这些热点数据时就会导致大量的磁盘I/O，性能下降。而缓存中的这些数据后续可能不会被用到，从而造成缓存污染。

如何避免缓存污染

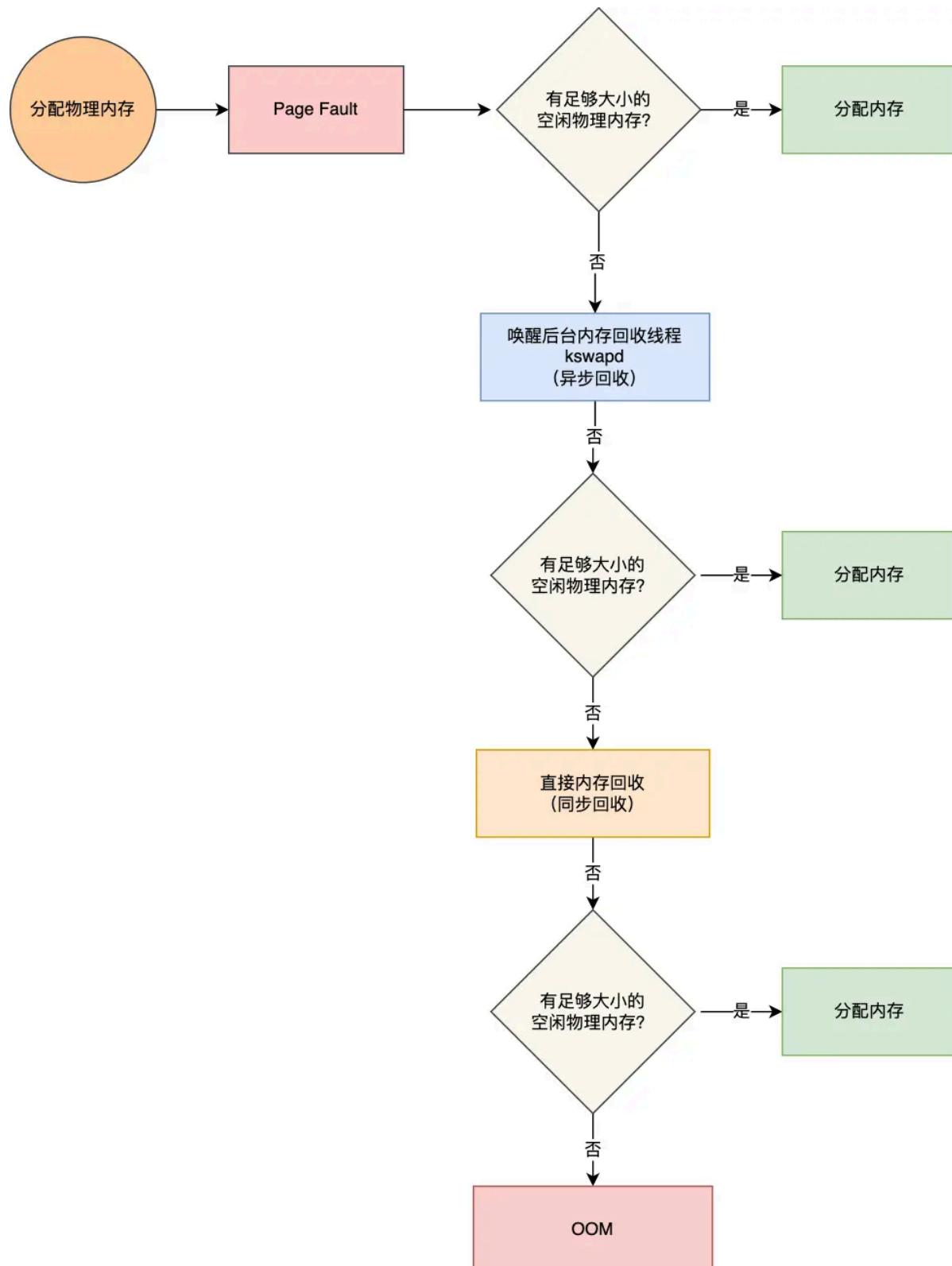
我们仔细分析一下导致缓存污染的根本原因是什么，当页面被访问一次，我们就将其放入了活跃LRU链表或者young区域，即使它们后续可能不被访问，也就是门槛太低了。

因此，只需要提高进入活跃LRU链表/young区域的门槛即可，Linux中当不活跃LRU链表中的页面被访问2次时，才将其升级。而MySQL不仅old区域的页面需要被访问2次才将其升级至young区域，还要两次访问间隔时间大于1s。通过提高升级门槛，如果这些数据只被访问1次，则不会升级，也就不会替换掉热点数据，减少缓存污染带来的影响。

物理内存满后，操作系统的一些方法：

1. 交换分区，LRU将一些长时间未使用的内存页面换出到硬盘中
2. 内存压缩，比如Windows任务管理器性能页面会有已压缩，通过压缩算法压缩包含大量重复信息和空闲空间的页面，将压缩的页面保存在内存的一个专有区域，这些页面被访问时再解压回来
3. 内存回收，回收缓存数据，分为后台内存回收和直接内存回收，一个是异步的，一个是同步的
4. 优化内存分配，比如延迟分配（也就是不到最后要用都不分配）
5. OOM

当进程访问虚拟地址，发生缺页中断，操作系统需要为其分配内存的流程如下：



当常规的内存回收策略比如后台内存回收和直接内存回收后，系统中仍然没有足够的内存进行分配时，操作系统会触发OOM killer来终止一些进程，防止系统因为内存耗尽而崩溃。OOM选择终止的进程需要考虑多个指标，比如内存使用量，运行时间和重要性等，需要最小化终止内存对系统的影响。

哪些内存可以被回收

主要有2种内存可以被回收：

1. 文件页：包括操作系统缓存的磁盘数据（buffer）和缓存的文件数据（cache），因为这些数据下次再次需要时可以从磁盘中读取，因此可以被回收，**非脏页可以直接回收，脏页（进行了修改但还没写入磁盘中）需要写入磁盘后再回收**
2. 匿名页：前面在malloc的时候提到使用mmap分配的页是私有且匿名的，所谓匿名就是没有具体的载体，不同于映射的磁盘文件，这部分内存可能还会被使用，因此回收时先swap置换到磁盘，再进行回收

回收文件页和匿名页都是通过LRU算法，LRU算法通过维护两个链表，一个是活跃内存页链表，一个是非活跃内存页链表，越靠近链尾越不活跃，优先被回收（也可以只通过一个链表实现）。

```
worker2@k8s-worker2:~$ cat /proc/meminfo | grep -i active | sort
Active:          777716 kB
Active(anon):    422500 kB
Active(file):    355216 kB
Inactive:        2122224 kB
Inactive(anon):  262524 kB
Inactive(file):  1859700 kB
```

内存回收带来的性能影响

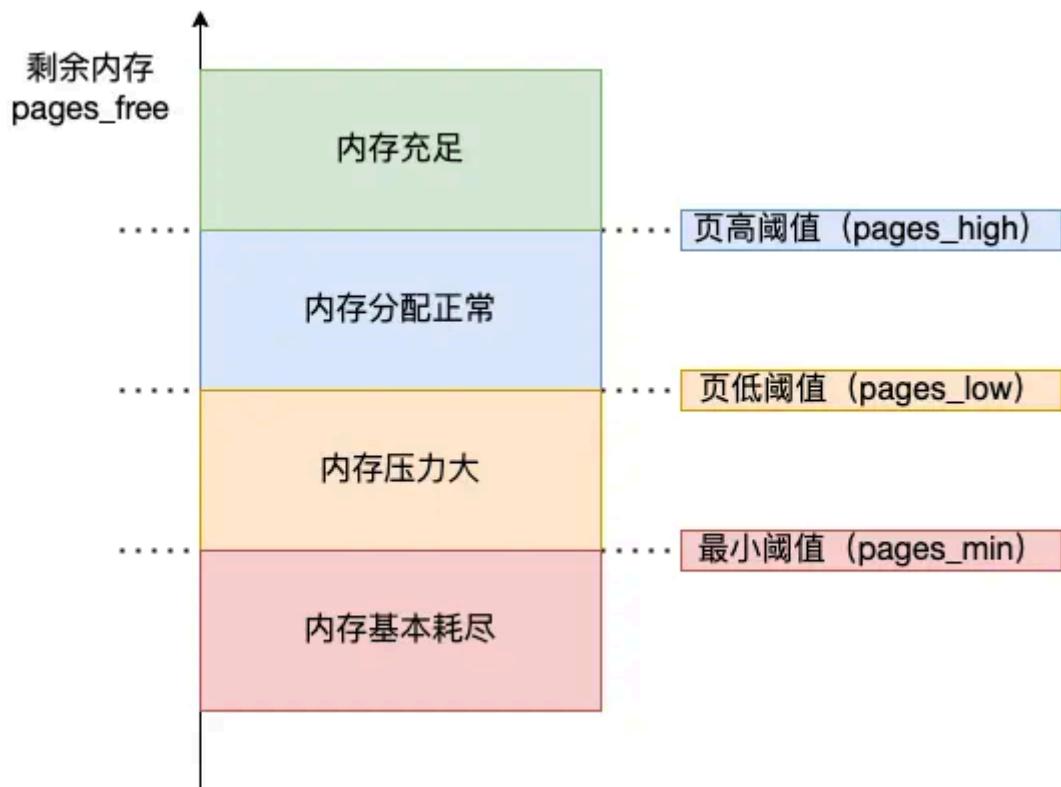
内存回收是会带来一些性能影响的，后台内存回收是异步的还好，直接内存回收是同步的，会导致阻塞，有性能损耗。此外，文件页回收中的脏页需要先写入磁盘，匿名页则是需要swap到磁盘，这两个都是有性能损耗的。

调整文件页和匿名页的回收倾向

匿名页肯定会写磁盘，而文件页中的干净页是可以直接回收的，下面的操作可以调整操作系统对这两种内存页的回收倾向。0~100，越大越倾向于回收匿名页。

```
worker2@k8s-worker2:~$ cat /proc/sys/vm/swappiness
60
```

尽早触发后台内存回收



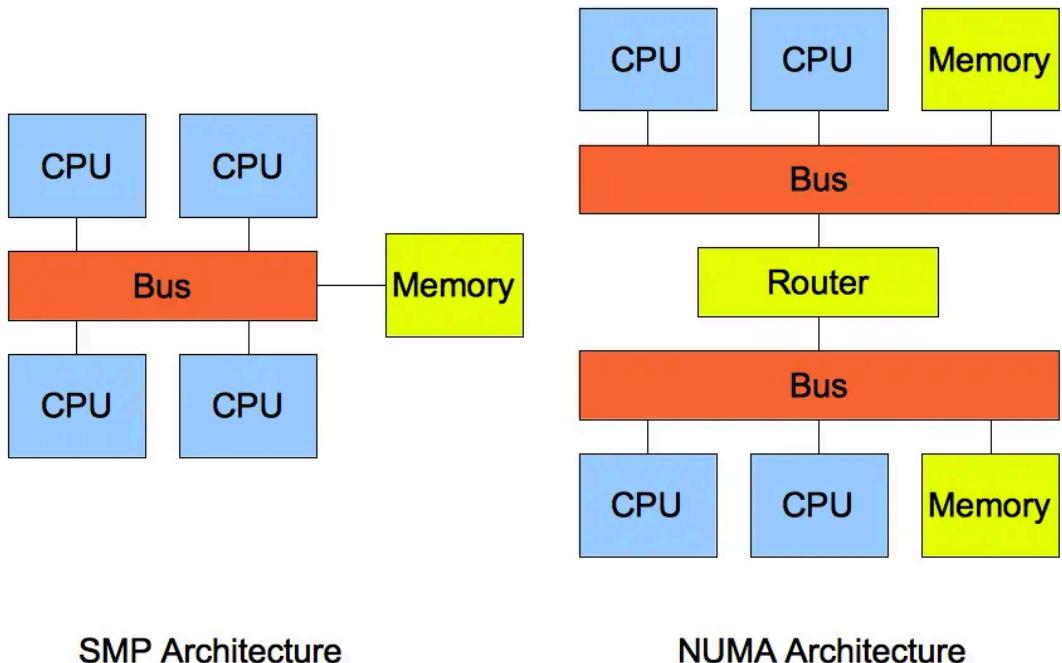
系统剩余内存处于内存压力大的区域，操作系统会开启后台内存回收；剩余内存处于内存基本耗尽的区域时，操作系统会开启直接内存回收。因此可以调整阈值。

NUMA价格下的内存回收策略

SMP架构指所有CPU都共用一个bus总线，比如访问内存，也称为一致性访问存储模型。

NUMA架构是将CPU进行分组，每个组称为一个node，包含多个CPU，每组有独立的资源，包括内存等。不同node通过互联模块总线进行通信，也就是说每个CPU都可以访问所有资源，但访问远端node的资源耗时更长。

在NUMA架构下，当某个Node内存不足时，系统可以从其他Node寻找空闲内存，也可以从本地内存中回收内存。



具体选哪种模式，可以通过`/proc/sys/vm/zone_reclaim_mode`来控制。它支持以下几个选项：

- 0（默认值）：在回收本地内存之前，在其他 Node 寻找空闲内存；
- 1：只回收本地内存；
- 2：只回收本地内存，在本地回收内存时，可以将文件页中的脏页写回硬盘，以回收内存。
- 4：只回收本地内存，在本地回收内存时，可以用 swap 方式回收内存。

如何保护一个进程不被OOM杀掉

当操作系统不得不使用OOM时，会通过算法计算出每个可以被杀掉的进程的分数，得分最高的会被杀掉。具体的计算过程就不说了，但可以设置进程的OOM校准值，`/proc/pid/oom_score_adj`，范围为-1000~1000，默认为0，也就是内存页面使用越多的越容易被杀掉。设置为-1000时计算所得的分数使得这个进程永远不会被杀死，例如一些系统程序sshd，而自己的业务程序不推荐这样设置。

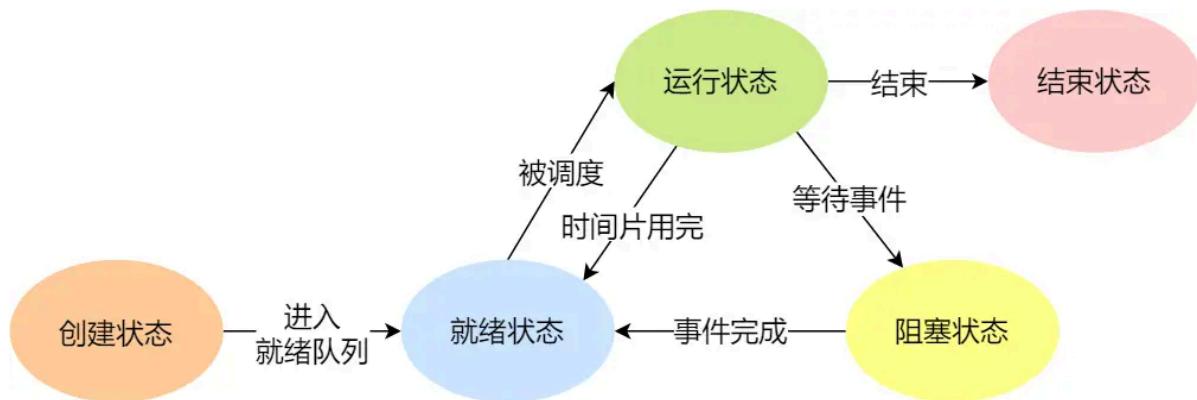
进程

进程的状态

一个进程在活动期间至少拥有3种基本状态：运行状态，就绪状态和阻塞状态：

1. 运行状态：该进程正在占用CPU，或者说CPU正在执行该进程
2. 就绪状态：该进程可以运行（比如数据已经准备完毕），等待操作系统分配时间片调度，也就是等待CPU执行
3. 阻塞状态：该进程正在等待某一事件（例如输入输出的完成）而暂时停止运行，此时赋予其CPU控制权，也无法运行

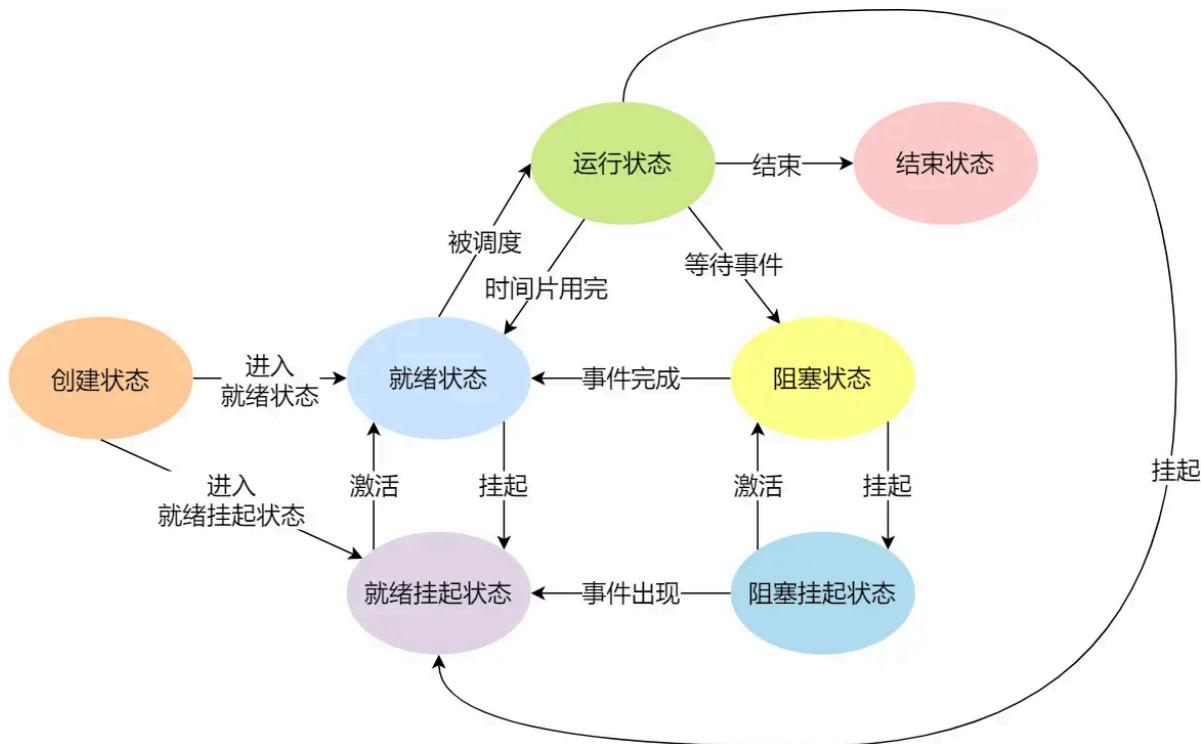
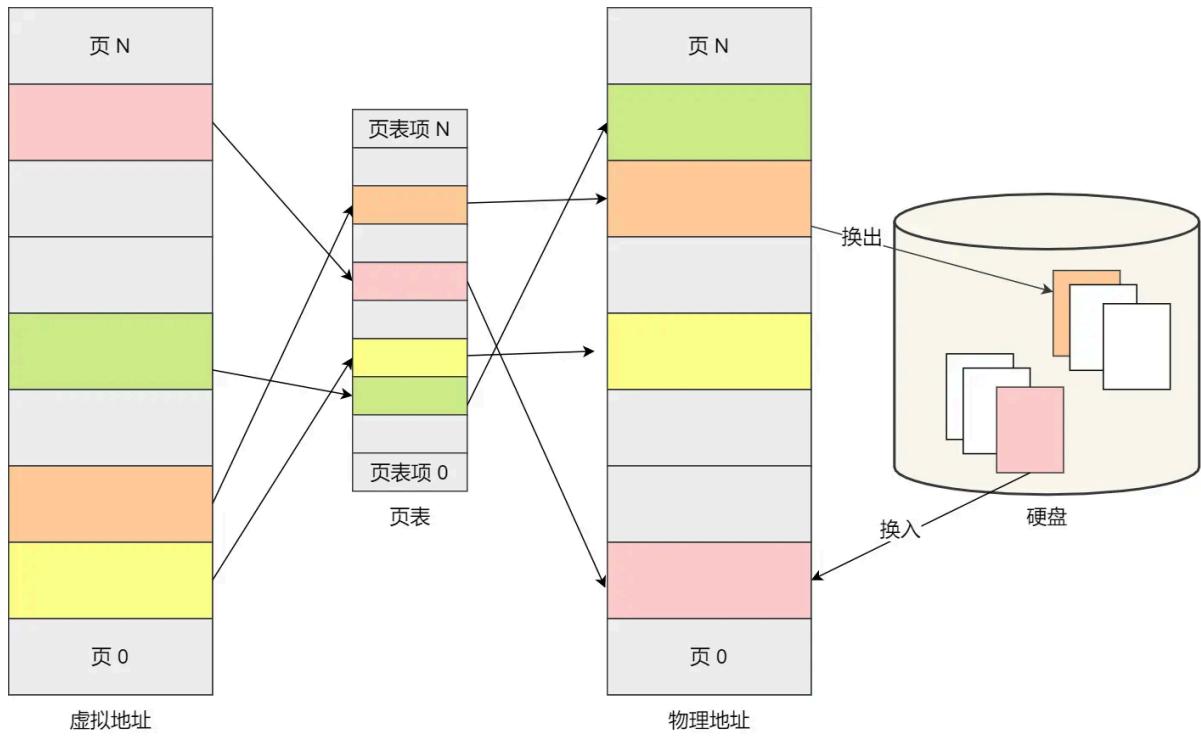
此外，进程还拥有另外2种基本状态：创建状态和结束状态，对应进程正在被创建的过程中以及运行完成或发生错误而结束的状态。



进程不同状态的变更：

- NULL -> 创建状态：一个新进程被创建时的第一个状态；
- 创建状态 -> 就绪状态：当进程被创建完成并初始化后，一切就绪准备运行时，变为就绪状态，这个过程是很快的；
- 就绪态 -> 运行状态：处于就绪状态的进程被操作系统的进程调度器选中后，就分配给CPU正式运行该进程；
- 运行状态 -> 结束状态：当进程已经运行完成或出错时，会被操作系统作结束状态处理；
- 运行状态 -> 就绪状态：处于运行状态的进程在运行过程中，由于分配给它的运行时间片用完，操作系统会把该进程变为就绪态，接着从就绪态选中另外一个进程运行；
- 运行状态 -> 阻塞状态：当进程请求某个事件且必须等待时，例如请求 I/O 事件；
- 阻塞状态 -> 就绪状态：当进程要等待的事件完成时，它从阻塞状态变到就绪状态；

处于阻塞状态的进程会占用物理内存资源，如果拥有大量被阻塞的进程，就会造成物理内存资源浪费。在虚拟内存管理的操作系统中，通常会把处于阻塞状态的进程的物理内存空间换出到硬盘，等到再次需要时，再换进物理内存空间。此时进程位于硬盘中，没有占用实际的物理内存空间，位于挂起状态。挂起状态可分为阻塞挂起状态和就绪挂起状态。



进程除了因为物理内存空间不够被换出到硬盘，从而处于挂起状态外，还可能因为sleep或ctrl+z等操作挂起一个进程。

进程的控制结构

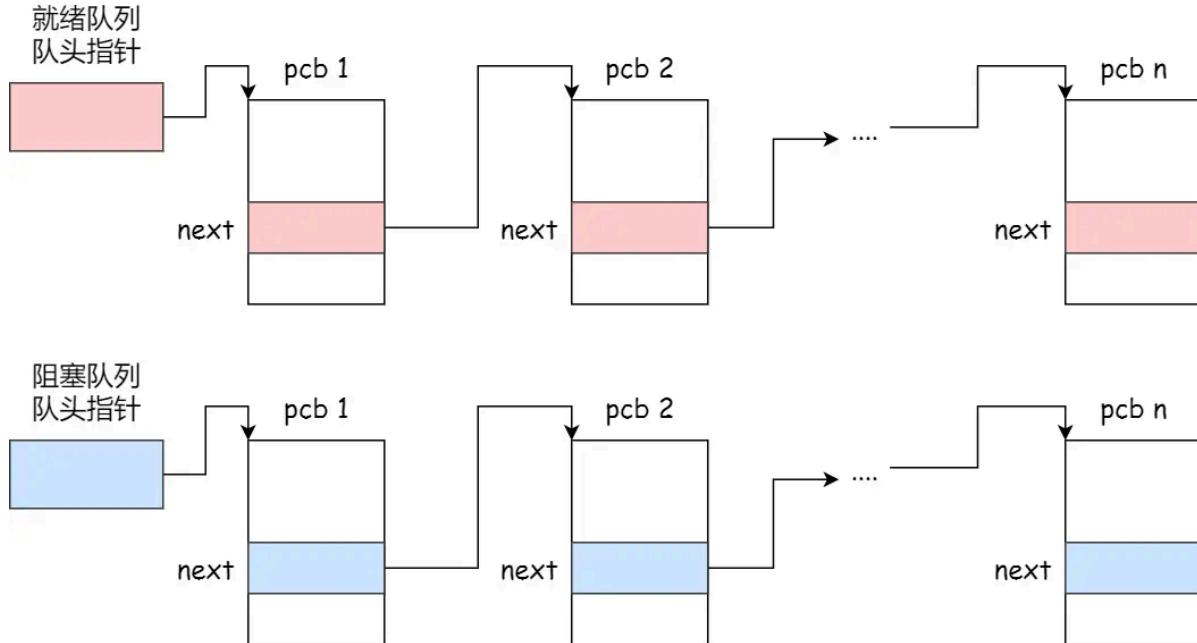
PCB (Process Control Block, 进程控制块) 是操作系统中用于存储有关进程的信息的数据结构，是操作系统进行进程管理和调度的核心数据结构，它包含了操作系统需要的所有信息，以便能够管理进程。task_struct是Linux操作系统中实现PCB功能的具体结构。PCB也是进程存在的唯一标识。

PCB中的内容包括：

- 进程状态：如运行、就绪、阻塞等。
- 进程标识符 (PID)：唯一标识一个进程。
- 程序计数器：指示进程下一条要执行的指令的位置。

- CPU 寄存器：存储进程的当前工作状态。
- CPU 调度信息：优先级、调度队列指针等。
- 内存管理信息：页表、段表等。
- 会计信息：CPU 使用时间、实际使用时间、进程创建和终止的时间等。
- I/O 状态信息：分配给进程的 I/O 设备列表、打开文件列表等。

操作系统通过链表的方式将处于相同状态的进程（PCB）链接在一起，形成不同的队列，例如阻塞队列，就绪队列等。单对于单核CPU而言，在某个时间只能运行一个进程。除了链表的方式，还可以通过索引，但链表创建和删除节点更加灵活。



进程的控制

进程的创建：进程可以创建另一个进程，子进程继承父进程所拥有的资源。

1. 申请一个空白PCB，并填入相关的信息，例如PID等
2. 为新进程分配运行所需的资源
3. 放入就绪队列，等待被os调度

进程的终止：进程终止大致可分为正常结束，异常结束和外界干预（信号kill）。子进程被终止时会将其资源归还给父进程，而父进程终止时，子进程将成为孤儿进程，交由1号进程（init进程，所有进程的祖先）处理（比如释放资源等）。

1. 查找需要被终止的PCB，若进程处于运行状态，立刻终止其执行，并将CPU资源分配给其他进程
2. 如果该进程有子进程，则将子进程交由1号进程处理
3. 释放资源
4. 从对应的队列删除

进程的阻塞：当进程需要等待某事件完成时，可以调用阻塞语句置为阻塞状态。当进程位于阻塞状态时，只能由另外一个进程唤醒。

1. 查找需要阻塞的PCB，如果当前进程位于运行状态，需要保存其上下文，并置为阻塞状态，停止运行
2. 添加至阻塞队列

进程的唤醒：位于阻塞状态的进程只能由其他进程唤醒，并且此时所等待的事件已经完成。

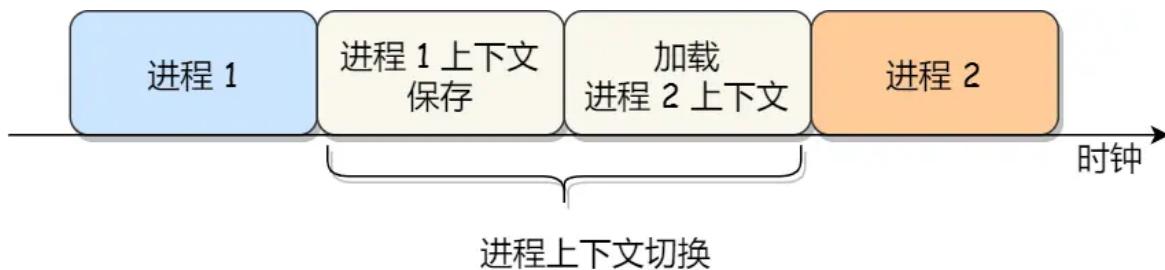
1. 在阻塞队列中查找需要唤醒的PCB
2. 将其状态置为就绪状态，并放入就绪队列

进程的上下文切换

一个进程切换到另外一个进程，称为进程的上下文切换。CPU运行进程时需要知道代码（指令）从哪里加载以及执行到哪里，也就是需要设置**CPU寄存器和程序计数器（PC）**，这两者都是CPU内部的硬件组件，被称为**CPU上下文**。CPU上下文切换也就是将前一个任务的CPU寄存器和程序计数器保存起来，然后加载新任务的上下文至CPU寄存器和程序计数器。

- CPU寄存器：CPU寄存器是小的、快速的存储设备，位于CPU内部，用于临时存储指令、数据和地址等信息。寄存器的存取速度非常快，远快于主内存。
- PC：程序计数器，也称为指令指针，是一个特殊的寄存器，用于存储下一条要执行的指令的内存地址。

进程的上下文切换包括用户空间的信息和内核空间的信息，当发生进程上下文切换时，内核会将当前进程的上下文信息保存在PCB中，然后从另一个进程的PCB中取出上下文信息，并恢复至CPU中，继续执行。



线程

为什么需要线程？

- 多进程之前的通信和数据共享
- 创建，终止，进程切换和维护进程开销大

进程的优点：

- 轻量
- 并发
- 一个进程可以多个线程
- 共享进程资源

进程的缺点：

- 当进程中的一个线程崩溃时，会导致其所属进程的所有线程崩溃（这里是针对 C/C++ 语言，Java 语言中的线程奔溃不会造成进程崩溃）

一些补充：

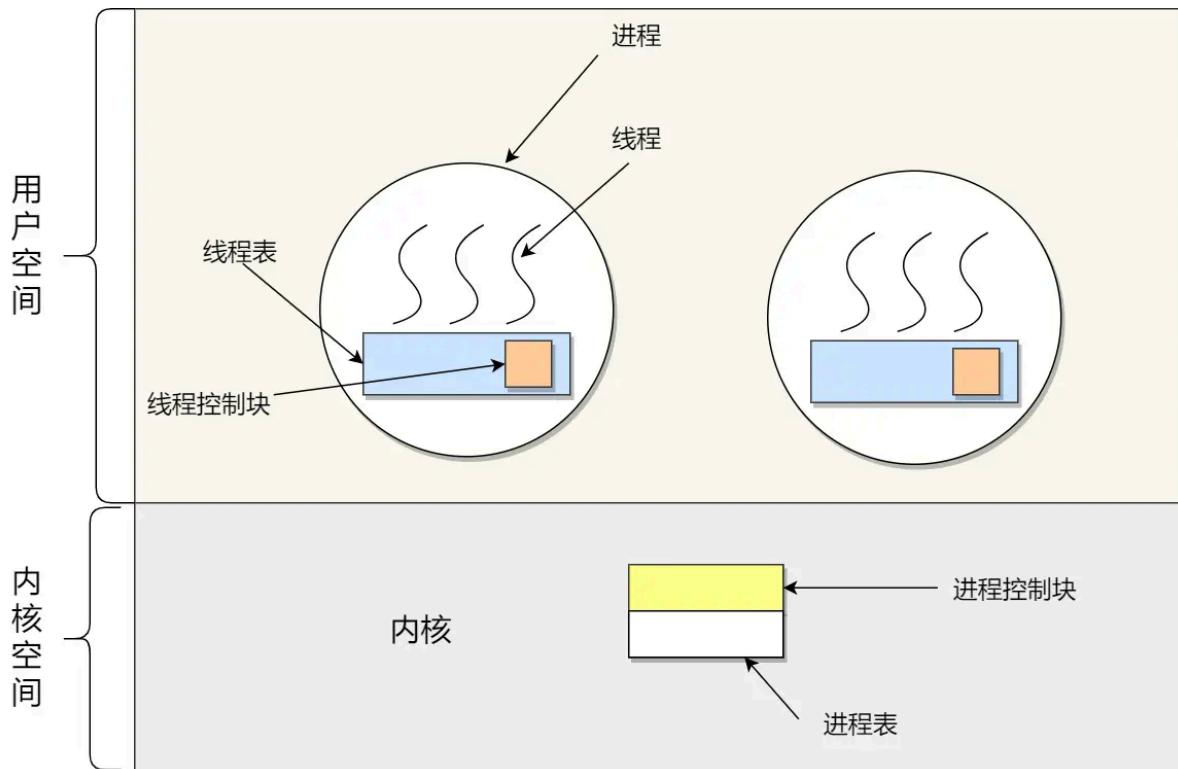
- 线程同样拥有运行、就绪和阻塞等状态
- 线程能减少并发执行的时空开销（位于同一地址空间，不需要换页表，只需要切换私有数据，例如寄存器和堆栈）
- **线程是os调度的基本单位**

线程的实现

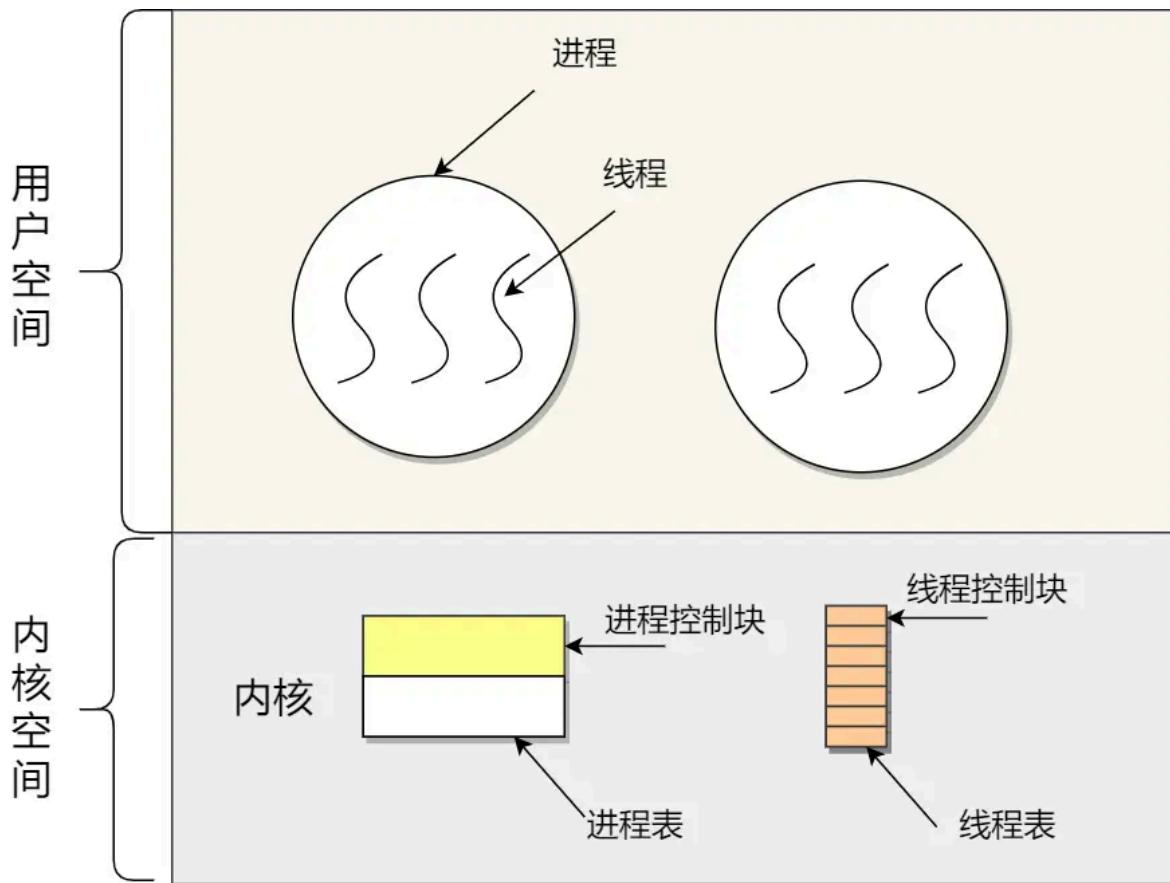
线程的实现方式可分为3种：

1. 用户线程：用户空间实现的线程，非内核管理，使用用户态的线程库创建
2. 内核线程：内核空间实现的线程，内核管理
3. 轻量级进程：在内核中来支持用户线程

用户线程是基于用户态的线程管理库来实现的，线程控制块（Thread Control Block, TCB）也是在库里面来实现的，对于操作系统而言是看不到这个TCB的，它只能看到整个进程的PCB。所以，用户线程的整个线程管理和调度，操作系统是不直接参与的，而是由用户级线程库函数来完成线程的管理，包括线程的创建、终止、同步和调度等。（多对一）



内核线程是由操作系统管理的，线程对应的TCB自然是放在操作系统里的，这样线程的创建、终止和管理都是由操作系统负责。（一对一）



轻量级进程 (Light-weight process, LWP) 是内核支持的用户线程，一个进程可有一个或多个LWP，每个LWP是跟内核线程一对一映射的，也就是LWP都是由一个内核线程支持，而且LWP是由内核管理并像普通进程一样被调度。在大多数系统中，LWP与普通进程的区别也在于它只有一个最小的执行上下文和调度程序所需的统计信息。一般来说，一个进程代表程序的一个实例，而LWP代表程序的执行线程，因为一个执行线程不像进程那样需要那么多状态信息，所以LWP也不带有这样的信息。

调度

详细内容看[博客——现代操作系统原理与实现-操作系统调度](#)。

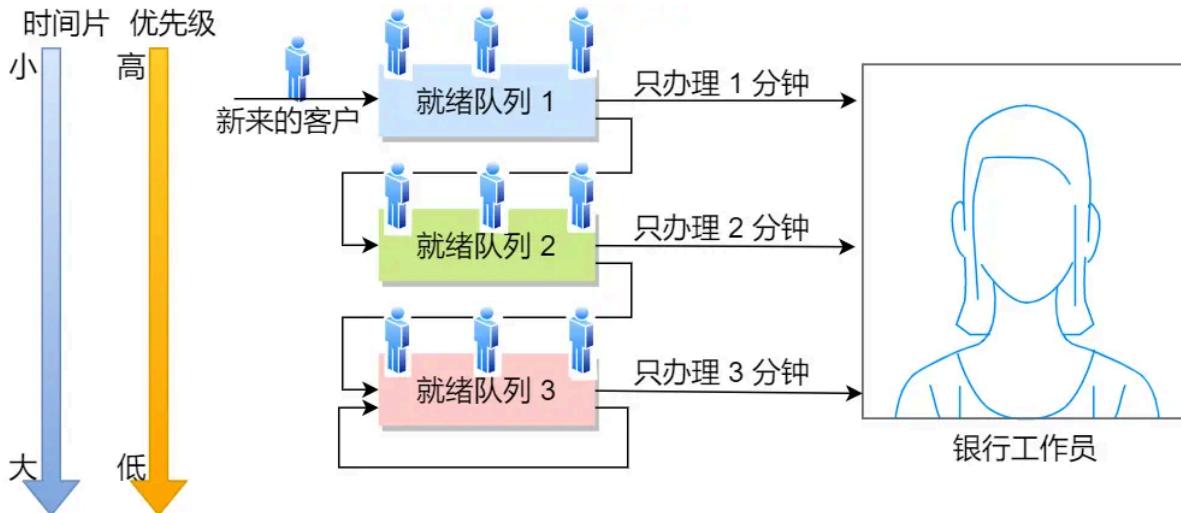
调度原则

- CPU 利用率：调度程序应确保 CPU 是始终忙碌的状态，这可提高 CPU 的利用率；
- 系统吞吐量：吞吐量表示的是单位时间内 CPU 完成进程的数量，长作业的进程会占用较长的 CPU 资源，因此会降低吞吐量，相反，短作业的进程会提升系统吞吐量；
- 周转时间：周转时间是进程运行+阻塞时间+等待时间的总和，一个进程的周转时间越小越好；
- 等待时间：这个等待时间不是阻塞状态的时间，而是进程处于就绪队列的时间，等待的时间越长，用户越不满意；
- 响应时间：用户提交请求到系统第一次产生响应所花费的时间，在交互式系统中，响应时间是衡量调度算法好坏的主要标准。

调度算法

最高优先级 (HPF) 调度算法：银行对客户分等级，分为普通客户、VIP 客户、SVIP 客户。只要高优先级的客户一来，就第一时间处理这个客户，这就是最高优先级 (HPF) 调度算法。但依然也会有极端的问题，万一当天来的全是高级客户，那普通客户不是没有被服务的机会，不把普通客户当人是吗？那我们把优先级改成动态的，如果客户办理业务时间增加，则降低其优先级，如果客户等待时间增加，则升高其优先级。

多级反馈队列 (MFQ) 调度算法：它是时间片轮转算法和优先级算法的综合和发展。



- 银行设置了多个排队（就绪）队列，每个队列都有不同的优先级，各个队列优先级从高到低，同时每个队列执行时间片的长度也不同，优先级越高的时间片越短。
- 新客户（进程）来了，先进入第一级队列的末尾，按先来先服务原则排队等待被叫号（运行）。如果时间片用完客户的业务还没办理完成，则让客户进入到下一级队列的末尾，以此类推，直至客户业务办理完成。
- 当第一级队列没人排队时，就会叫号二级队列的客户。如果客户办理业务过程中，有新的客户加入到较高优先级的队列，那么此时办理中的客户需要停止办理，回到原队列的末尾等待再次叫号，因为要把窗口让给刚进入较高优先级队列的客户。

可以发现，对于要办理短业务的客户来说，可以很快的轮到并解决。对于要办理长业务的客户，一下子解决不了，就可以放到下一个队列，虽然等待的时间稍微变长了，但是轮到自己的办理时间也变长了，也可以接受，不会造成极端的现象。

进程拥有独立的用户地址空间，共享内核地址空间，进程间通信必须通过内核，进程间通信的方式包括：

1. 管道
2. 消息队列
3. 共享内存
4. 信号量
5. 信号
6. socket

管道

Linux中的 | 就是管道，这种管道是单向数据传输的，称为匿名管道，用完即销毁。若想实现双向数据传输，则需要创建两个管道。

```
ps auxf | grep mysql
```

ps auxf: ps 是一个命令，用于报告当前进程的快照。

- a 告诉 ps 列出所有用户的进程。
- u 显示进程的用户/所有者。
- x 包括没有控制终端的进程，如守护进程。
- f 启用完整格式列表，包括进程树。

除了匿名管道，管道还有另外一种类型，即命名管道 FIFO，数据先进先出。

```
mkfifo pyqpipe
ls -l
# 可以看到文件类型显示为p, pipe
prw-r--r-- 1 pyq pyq 0 Apr 17 10:46 pyqpipe
# 使用标准输出重定向向管道中发送消息
echo "hello" > pyqpipe
# 读取管道中的内容
cat pyqpipe
```

I/O重定向：允许改变命令的标准输入，输出和错误的位置

标准输出重定向：> 和 >>，前者是替换文件内容，后者是追加到文件后面

标准错误重定向：2> 和 2>>

标准输入重定向：<

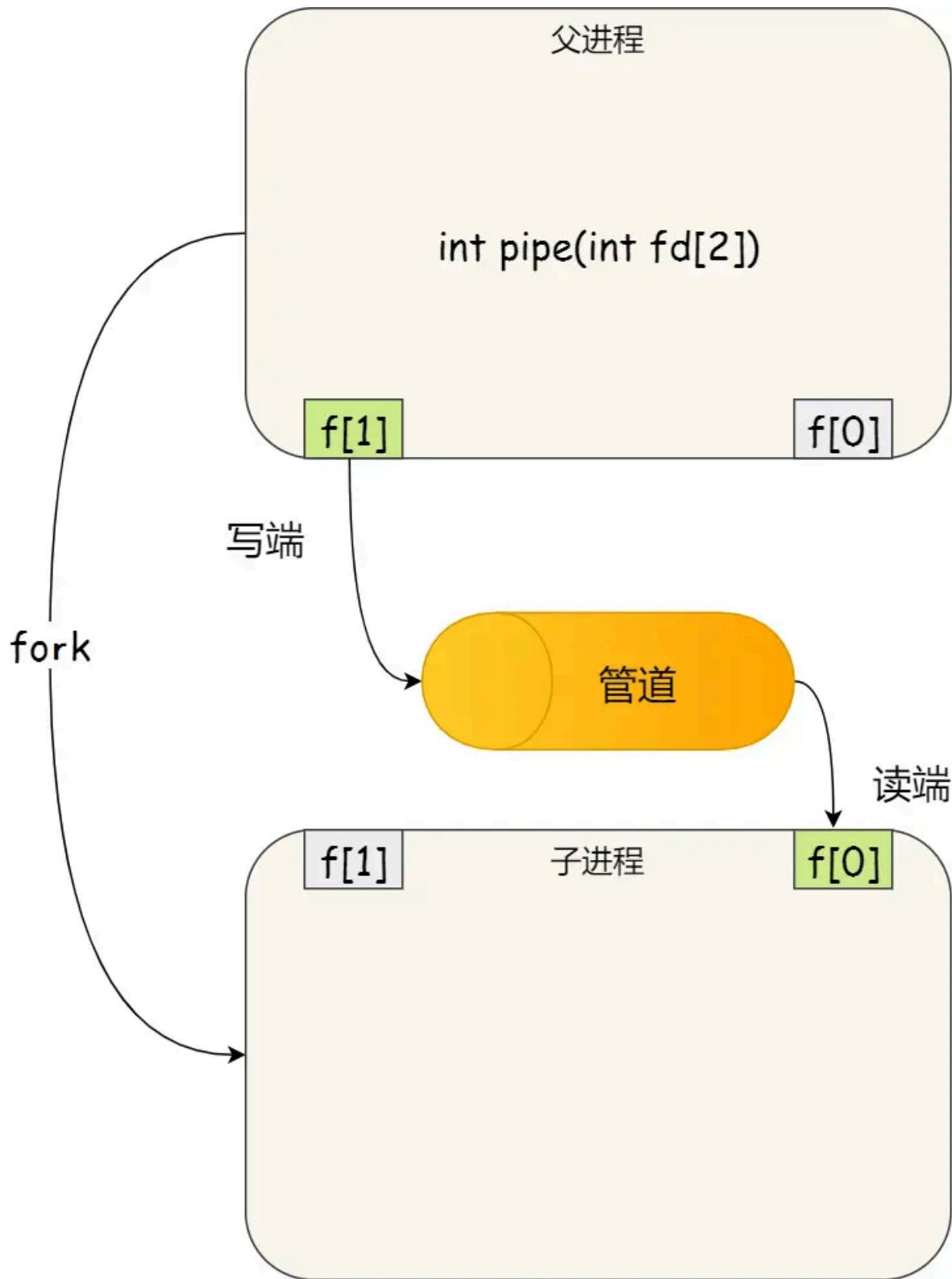
管道也是一种I/O重定向

如果管道中的内容没被读取，echo 命令不能正常退出。管道创建简单，很容易知道管道中的数据是否被另外一个进程所读取。缺点也很明显，效率低，不适合进程间频繁的数据交换。

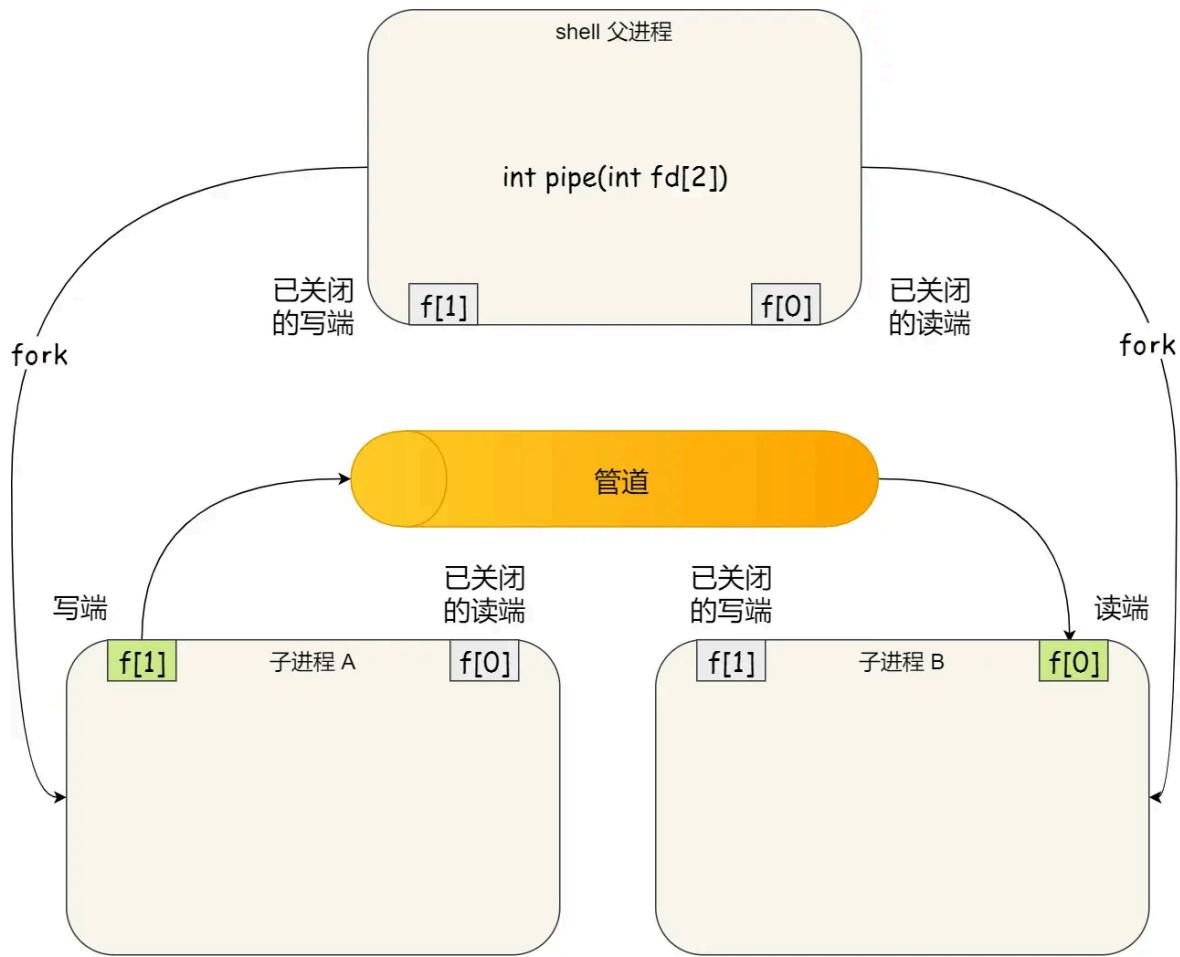
c中创建管道的方式：用于创建一个匿名管道，返回两个描述符，fd[0]为读端，fd[1]为写端。创建的管道仅位于内存中，本质是内核中的缓存

```
int pipe(int fd[2])
```

进程间使用管道通信的方式如下：注意同一时间只有一个读端或写端开启，可以看mit s6.081的实验。



使用 `ps auxf | grep mysql` 命令时，`ps` 和 `grep` 都是 shell 的子进程，过程如下：



匿名管道的通信范围是存在父子关系的进程，因为管道没有实体，只能通过fork父进程的fd来实现通信；而命名管道的适用范围更广些，因为存在实体；不管是匿名管道还是命名管道，通信数据都是FIFO的。

消息队列

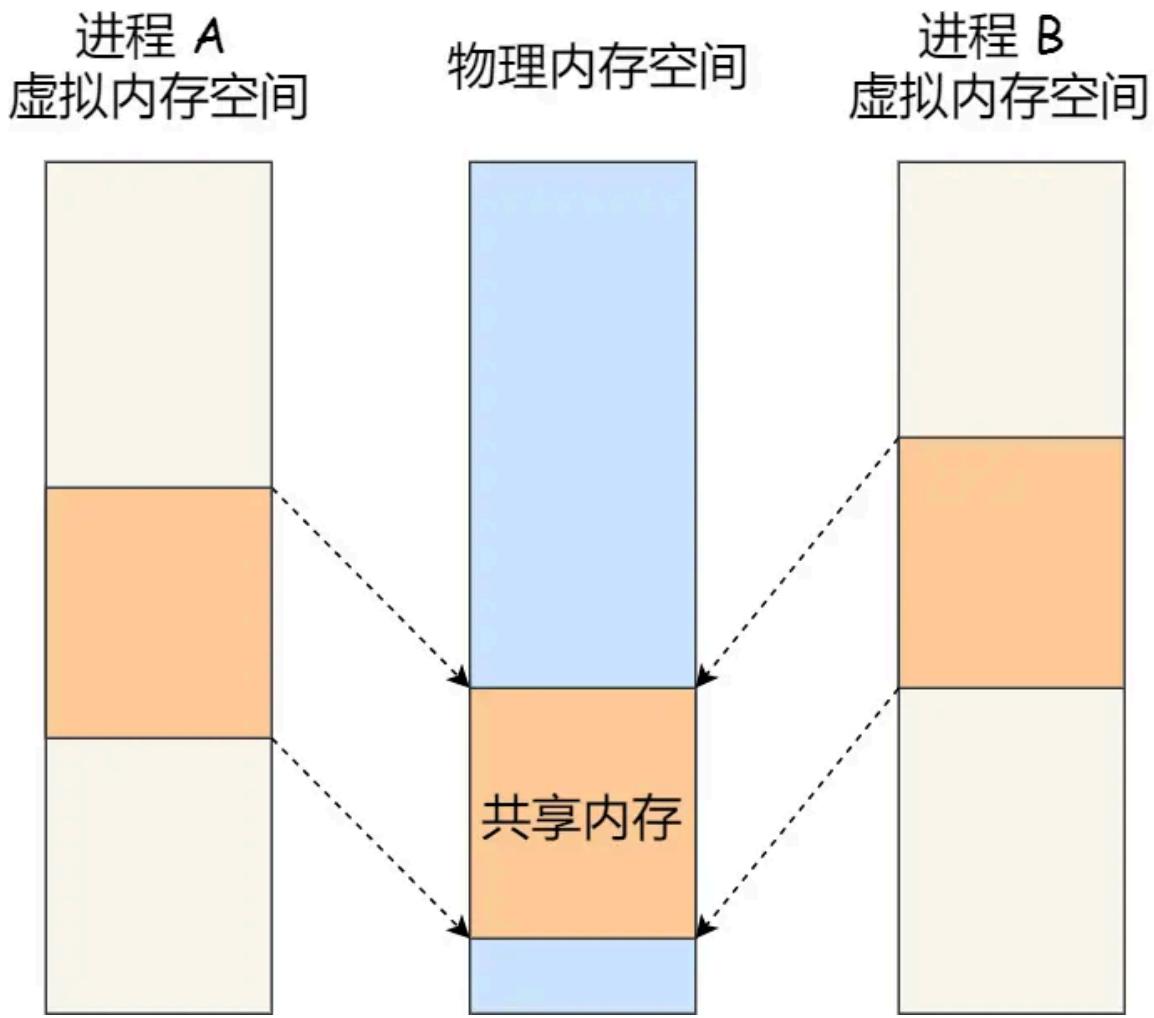
相较于管道，消息队列更适合进程间频繁通信。A进程需要给B进程发送消息，只需要把消息放置对应的消息队列即可返回（管道则是没有读取就不能返回），B再前往对应的消息队列读取即可。**消息队列本质是内核中的消息链表，发送数据格式是消息体（自定义的存储块，非管道中五格式的字节流数据）**。不主动释放消息队列，其生命周期随内核一起存在（区别于匿名管道）。

消息队列的缺点：一是通信不及时，二是附件也有大小限制。

- **消息队列不适合比较大数据的传输，因为在内核中每个消息体都有一个最大长度的限制，同时所有队列所包含的全部消息体的总长度也是有上限。**在Linux内核中，会有两个宏定义MSGMAX和MSGMNB，它们以字节为单位，分别定义了一条消息的最大长度和一个队列的最大长度。
- **消息队列通信过程中，存在用户态与内核态之间的数据拷贝开销**，因为进程写入数据到内核中的消息队列时，会发生从用户态拷贝数据到内核态的过程，同理另一进程读取内核中的消息数据时，会发生从内核态拷贝数据到用户态的过程。

共享内存

消息队列中数据的发送和读取需要用户态和内核态的拷贝，而**共享内存通过将两个进程的虚拟地址空间映射至同一个物理内存中**，一个进程进行写入，另外一个进程可直接读取，无需拷贝。



信号量

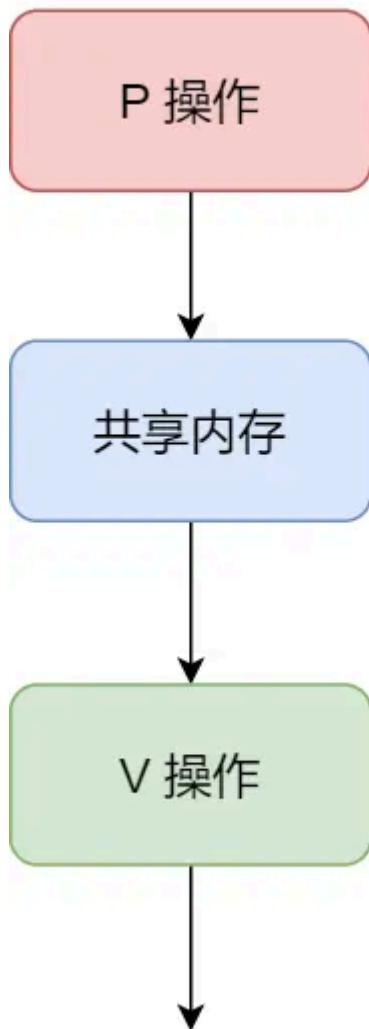
两个进程使用共享内存通信涉及到一个问题，也就是互斥和同步，如果两个进程同时写就可能发生冲突。**信号量是一个整形的计数器，用于进程间的互斥和同步。**

信号量表示资源的数量，控制信号量主要通过两种原子操作：

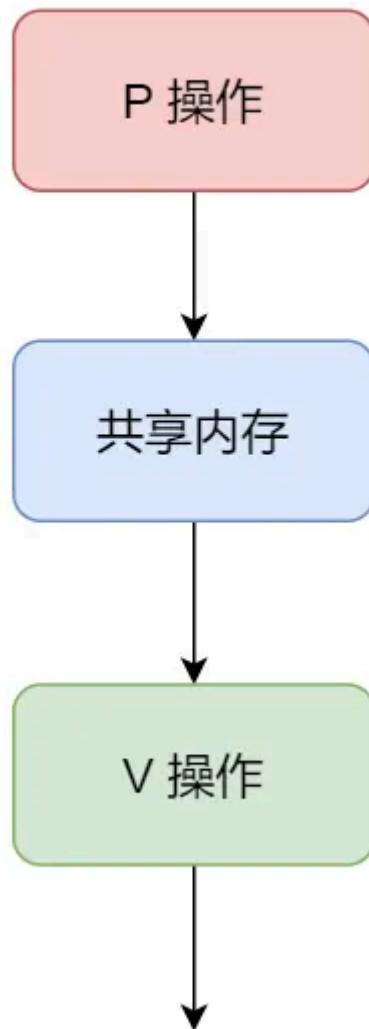
- P操作：使用资源前，将信号量-1；如果信号量在p操作后<0，表示当前已无可用资源，需阻塞等待；若信号量在p操作后 ≥ 0 ，表示进程可以使用资源。
- V操作：使用资源后，将信号量+1，如果信号量在v操作后 ≤ 0 ，表示当前有阻塞中的进程，需要将其唤醒；若信号量在v操作后>0，表示无阻塞中的进程。

将信号量设置为1，可以实现进程间互斥，保证任一时间只有1个进程在访问资源。

进程 A

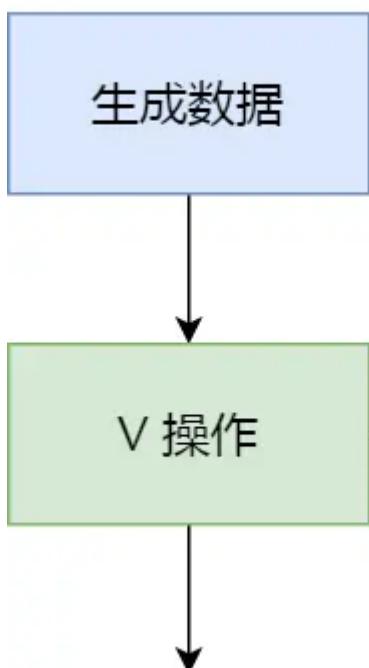


进程 B

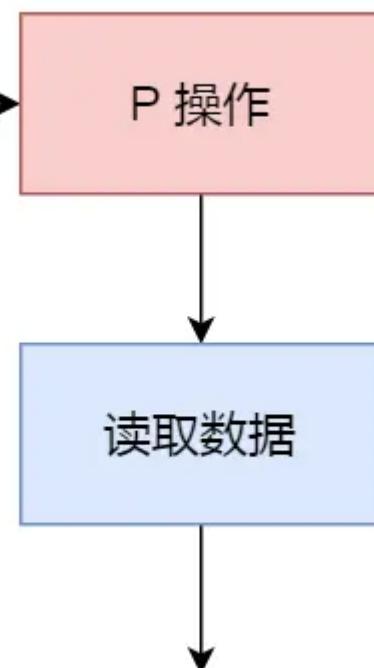


将信号量设置为0，可以实现进程间同步（被依赖的进程执行V操作，依赖的进程执行P操作），比如A进程写，B进程读，B进程需要在A进程写后来读。

进程 A



进程 B



- 假如B先执行，首先进行P操作，信号量变为-1，阻塞等待
- A执行，执行V操作，信号量变为0，唤醒B进程
- 假如A先执行，执行V操作，信号量变为1
- B执行，执行P操作，信号量变为0，可以执行

信号

前面所列举的进程间通信方式均是进程正常状态下的，而对于异常情况下的工作状态，需要使用信号的方式来通知进程，主要用于通知各种异常事件。信号是进程间唯一的异步通信，因为它们可以在任何时候发送到一个进程，通常与进程的正常控制流无关。这意味着信号可以在程序执行的任何点被接收和处理，而不需要进程显式地在某个时刻等待或检查信号的到来。

Linux中的信号包括：Ctrl+C产生SIGINT信号，表示终止该进程；Ctrl+Z产生SIGTSTP信号，表示停止该进程，但还未结束；kill也是一种信号；

```
$ kill -1
 1) SIGHUP      2) SIGINT      3) SIGQUIT      4) SIGILL      5) SIGTRAP
 6) SIGABRT     7) SIGBUS      8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGNALRM    15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU    25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

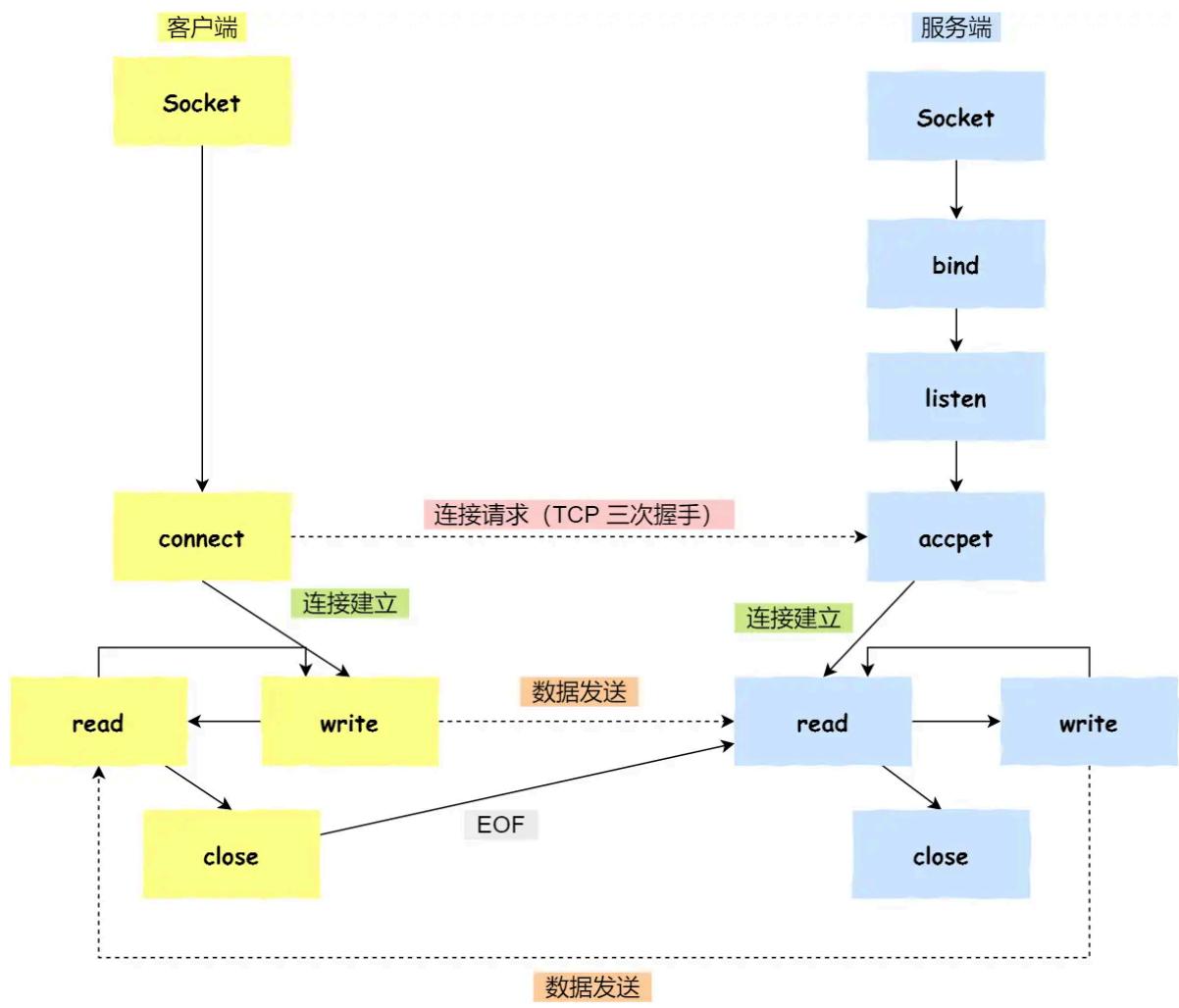
用户进程对信号的处理方式。

1. 执行默认操作。Linux对每种信号都规定了默认操作，例如，上面列表中的SIGTERM信号，就是终止进程的意思。
2. 捕捉信号。我们可以为信号定义一个信号处理函数。当信号发生时，我们就执行相应的信号处理函数。
3. 忽略信号。当我们不希望处理某些信号的时候，就可以忽略该信号，不做任何处理。有两个信号是应用进程无法捕捉和忽略的，即SIGKILL和SEGSTOP，它们用于在任何时候中断或结束某一进程。

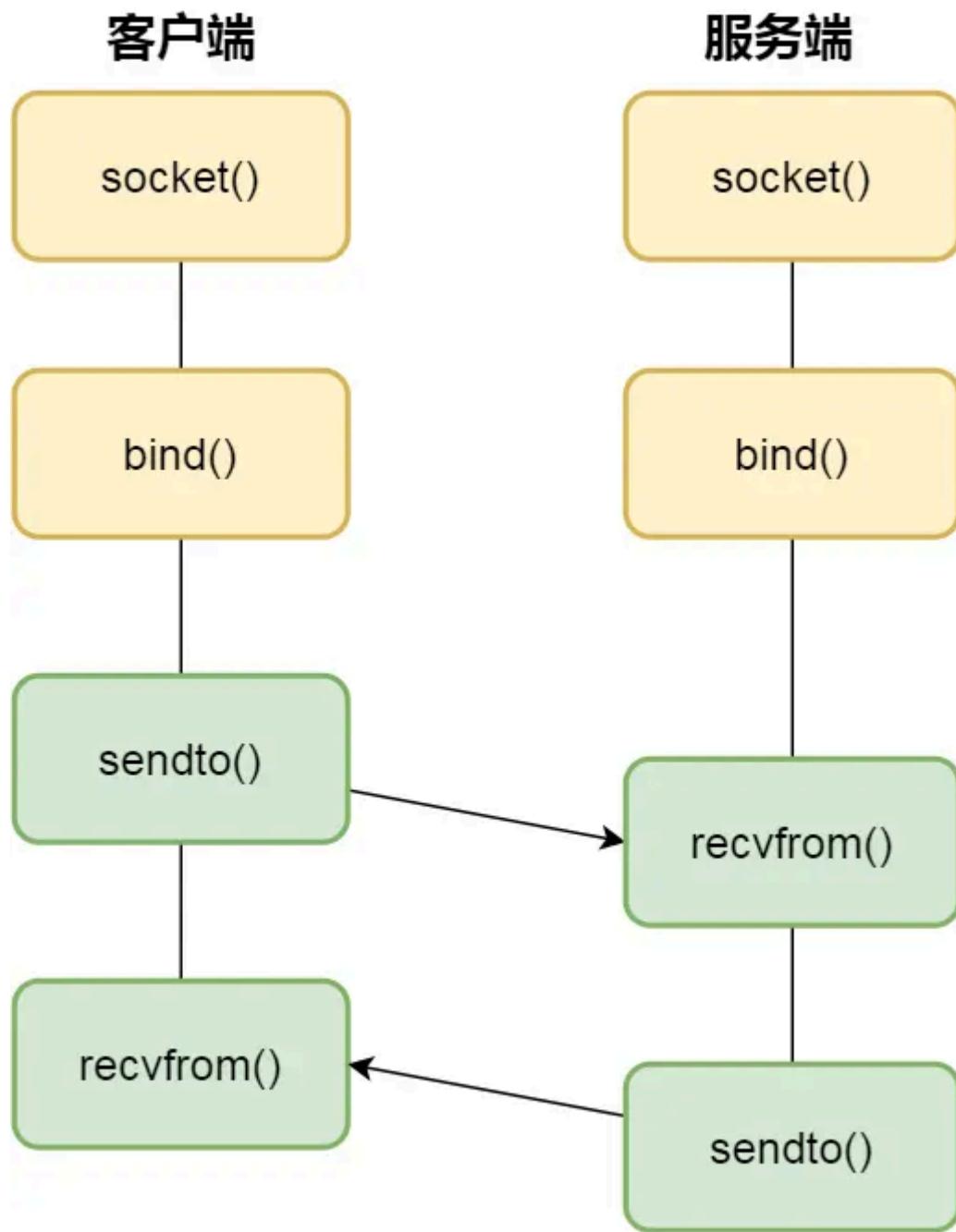
中断并不是进程间通信，是操作系统用来处理硬件信号的一种机制，允许硬件设备通知CPU有即时处理需求的事件，或者用于操作系统内核管理软件事件。这种机制使得CPU可以响应和处理外部或内部的事件，而不必持续检查事件的状态。中断也是异步的。

socket

TCP (字节流 SOCK_STREAM)



UDP (数据报 SOCK_DGRAM)



本地进程通信：本地字节流socket和本地数据报socket在bind的时候，不像TCP和UDP要绑定IP地址和端口，而是**绑定一个本地文件**，这也就是它们之间的最大区别。

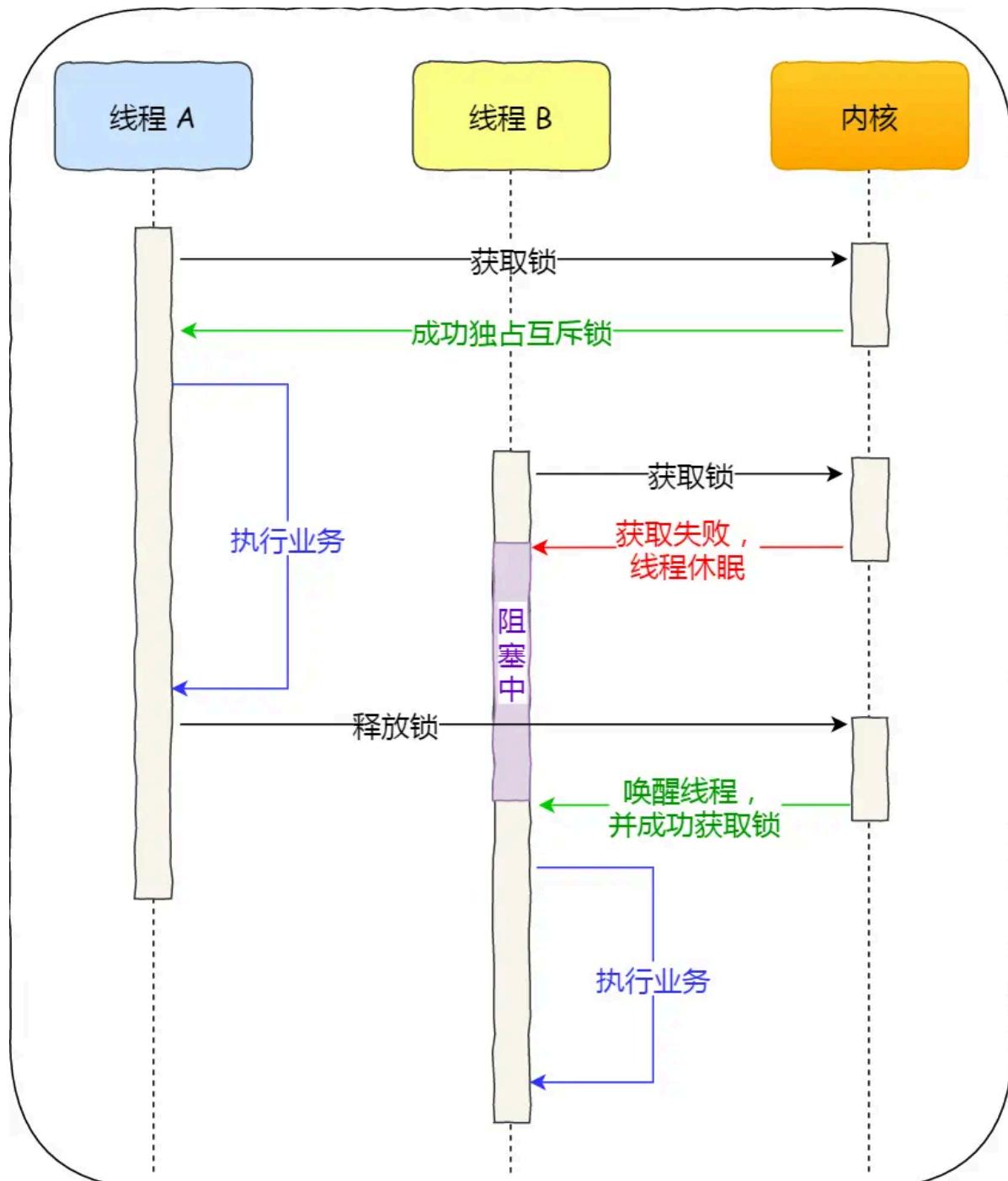
使用锁的目的是为了保证共享资源在任意时刻只有一个线程访问。常见的锁包括：

- 互斥锁
- 自旋锁
- 读写锁
- 乐观锁
- 悲观锁

互斥锁和自旋锁

互斥锁和自旋锁是两种偏底层的锁，很多高级的锁都是由这两种锁实现的：

- 互斥锁：加锁失败后，会立即释放CPU，进入阻塞状态（例如c++11提供的mutex）
- 自旋锁：加锁失败后，会进入忙等状态，直到获取到锁



互斥锁中，进程尝试获取锁失败后会陷入内核态变为阻塞状态，等待锁可用时再由内核唤醒，从阻塞态变为就绪态，**会发生2次线程上下文切换**（线程的上下文切换主要是线程的私有资源，比如寄存器和私有数据等）。

自旋锁是通过CPU提供的CAS函数（Compare And Swap）（之前提到的test and set也差不多，都是原子指令/操作），**在用户态完成加锁和解锁操作，不会主动产生线程上下文切换，所以相比互斥锁来说，会快一些，开销也小一些。**

CAS操作通常包含三个参数：

内存位置（Memory Location）：要操作的数据的内存地址。

预期值（Expected Value）：预期内存位置中的值。

新值（New Value）：如果内存位置的当前值与预期值匹配，则将其更新为此新值。

操作的步骤如下：

首先，CAS会检查目标内存位置的当前值是否与提供的预期值相匹配。

如果它们匹配，那么将内存位置的值更新为新值。

如果它们不匹配，操作失败，通常返回内存位置的当前值。

CAS实现自旋锁：设锁为变量lock，整数0表示锁是空闲状态，整数pid表示线程ID，那么CAS(lock, 0, pid)就表示自旋锁的加锁操作，CAS(lock, pid, 0)则表示解锁操作。忙等待状态可通过while实现。

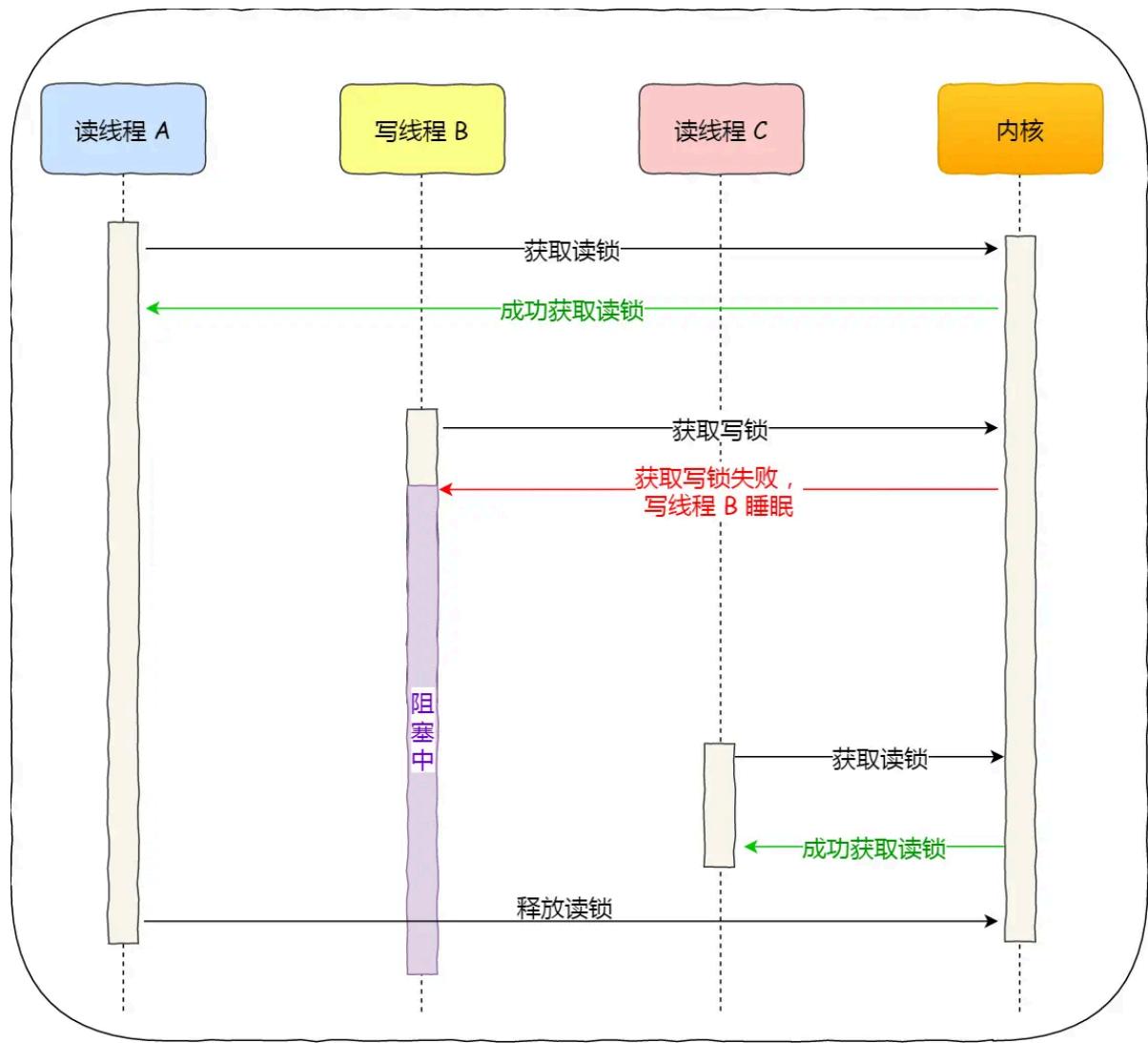
在单核CPU上，需要抢占式的调度器（即不断通过时钟中断一个线程，运行其他线程）。否则，自旋锁在单CPU上无法使用，因为一个自旋的线程永远不会放弃CPU。

- 在多核环境中，自旋锁可能更高效，因为它避免了频繁的线程上下文切换。但在单核环境中，频繁的自旋会导致CPU资源浪费，因为自旋的线程实际上没有做任何有用的工作。
- 互斥锁在几乎所有情况下都更适合单核处理器，因为它们通过使线程休眠来更好地管理有限的CPU资源。

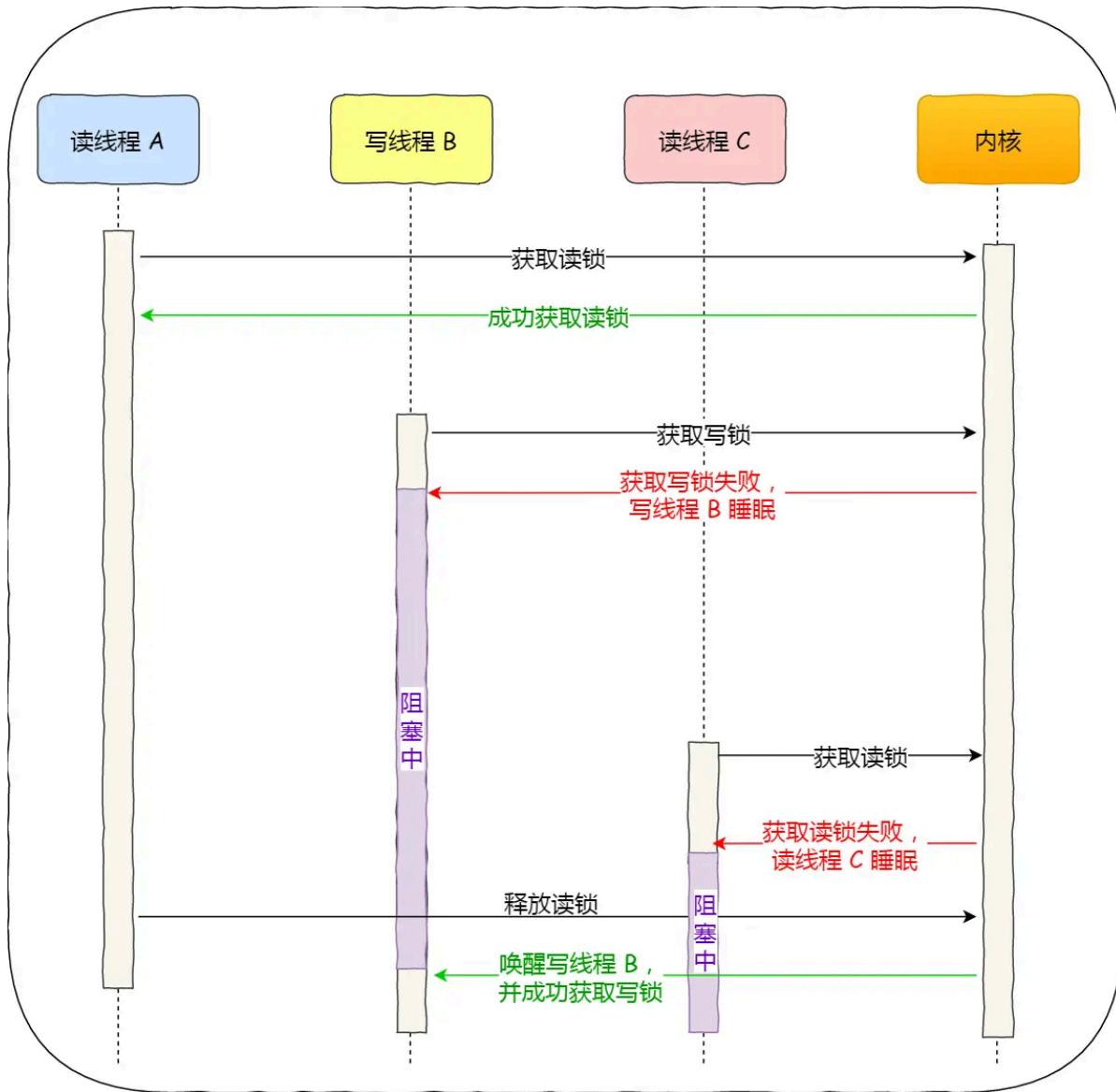
读写锁

读写锁和读者-写者问题相关，读锁是共享锁，允许多个读者并发拥有读锁；写锁是独占锁，同一时刻只允许一个写者拥有，并且不允许写锁和读锁同时被拥有。**读写锁适合读多写少的场景**，读写锁可分为读优先锁和写优先锁。读优先锁和写优先锁都会产生饥饿问题。

读优先锁：



写优先锁：



公平读写锁比较简单的一种方式是：用队列把获取锁的线程排队，不管是写线程还是读线程都按照先进先出的原则加锁即可，这样读线程仍然可以并发，也不会出现饥饿的现象。

读写锁可以选择互斥锁或自旋锁实现。

悲观锁和乐观锁

悲观锁适合发生冲突概率很高的场景，悲观锁在获取共享资源前需要加锁，再对共享资源进行操作。互斥锁、自旋锁和读写锁都属于悲观锁。

乐观锁适合发生冲突概率较低的场景，乐观锁在获取共享资源前不需要加锁，直接对共享资源进行操作，操作完成验证这段时间是否发生冲突，如果发生冲突则放弃修改，否则修改完成。CAS可以看作一种乐观锁。

乐观锁的例子：在线文档

我们都知道在线文档可以同时多人编辑的，如果使用了悲观锁，那么只要有一个用户正在编辑文档，此时其他用户就无法打开相同的文档了，这用户体验当然不好了。那实现多人同时编辑，实际上是用了乐观锁，它允许多个用户打开同一个文档进行编辑，编辑完提交之后才验证修改的内容是否有冲突。怎么样才算发生冲突？这里举个例子，比如用户A先在浏览器编辑文档，之后用户B在浏览器也打开了相同的文档进行编辑，但是用户B比用户A提交早，这一过程用户A是不知道的，当A提交修改完的内容时，那么A和B之间并行修改的地方就会发生冲突。

服务端要怎么验证是否冲突了呢？通常方案如下：

- 由于发生冲突的概率比较低，所以先让用户编辑文档，但是浏览器在下载文档时会记录下服务端返回的文档版本号；
- 当用户提交修改时，发给服务端的请求会带上原始文档版本号，服务器收到后将它与当前版本号进行比较，如果版本号不一致则提交失败，如果版本号一致则修改成功，然后服务端版本号更新到最新的版本号。
- 实际上，我们常见的SVN和Git也是用了乐观锁的思想，先让用户编辑代码，然后提交的时候，通过版本号来判断是否产生了冲突，发生了冲突的地方，需要我们自己修改后，再重新提交。

乐观锁虽然去除了加锁解锁的操作，但是一旦发生冲突，重试的成本非常高，所以**只有在冲突概率非常低，且加锁成本非常高的场景时，才考虑使用乐观锁。**

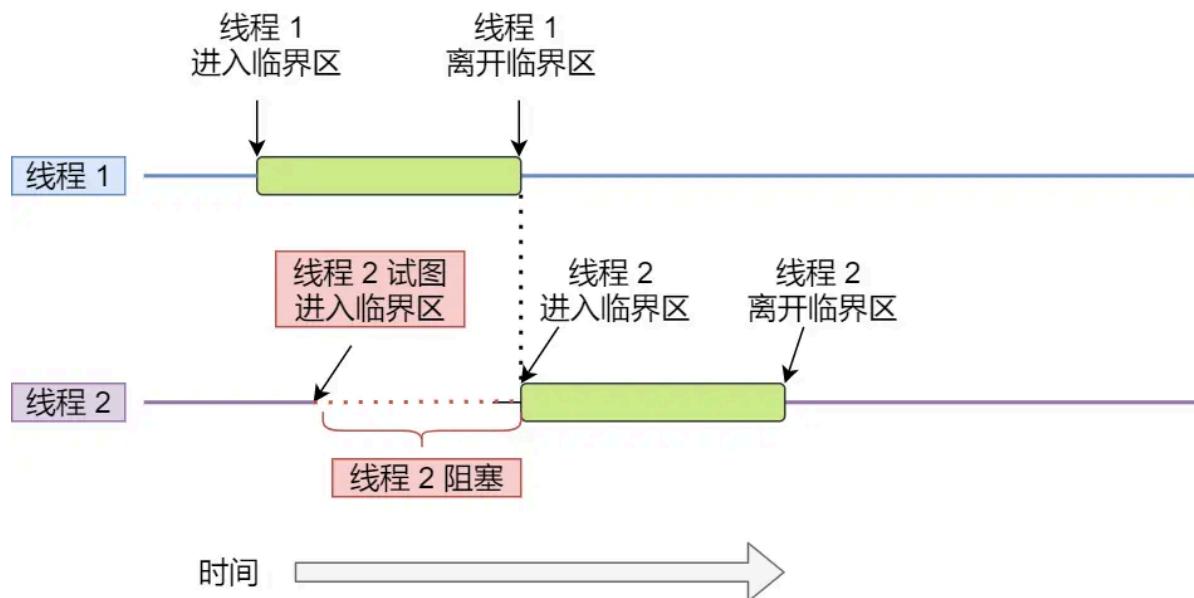
不管是什锁，加锁的粒度应该小，这样执行快。

竞争与协作

互斥

竞争条件：在多进程/多线程情景，未对共享资源做适当同步，从而导致不确定性。

临界区：访问共享资源（内存，文件）的程序片段，为了避免不确定性，需要保证同一时刻只能有一个线程/进程访问该区域，也就是互斥。



同步

多进程/线程场景下，除了互斥，还有同步。同步指多线程/进程之间存在某种依赖关系，需要互通消息，例如A进程是生产者，B进程是消费者，B必须等A生产出来才能消费。

同步就好比：操作A应在操作B之前执行，操作C必须在操作A和操作B都完成之后才能执行等；

互斥就好比：操作A和操作B不能在同一时刻执行；

互斥和同步的实现和使用

实现多线程/进程之间的协作，主要可通过锁和信号量实现（信号量相比锁更强大，可以方便的实现同步）：

- 锁：加锁和解锁
- 信号量：P, V操作

锁

锁可分为忙等待锁和无忙等待锁。

这里简单描述一下自旋锁的伪代码，定义一个锁：包含一个flag成员，lock方法和unlock方法（当然还有构造方法之类的）

lock和unlock都是原子操作，比如test and set，lock通过一个while循环判断锁是否有其他线程持有，如果没有则立刻退出while循环，并设置flag值使得其他线程尝试获取锁一直忙等（while）；unlock则是将flag设置为初始值

需要注意的是这种简单的是实现方式可能会出现A线程持有锁，而B线程释放锁的过程；一般来说锁的释放只能持有锁的线程进行，需要程序员的编程规范，或者记录持有锁的线程

- 忙等待锁：也称为自旋锁，如果获取不到锁，就一直等待。忙等待锁的简易实现可见代码，需要使用原子操作（要么都做，要么不做），避免多个线程同时改变锁状态。
- 无忙等待锁：获取不到锁不用自旋。

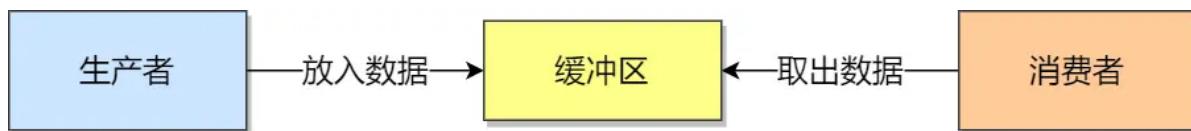
无忙等待锁，增加了一个队列用于存储阻塞的需要获取锁的线程，lock时若获取不到锁就将该线程添加至队列中，unlock再把队头的队列移出

信号量

信号量的伪代码实现和无忙等待锁有一些类似，一个成员用于表示资源数量（也就是支持多少个线程同时使用），一个队列用于存储阻塞的需要使用资源的线程，p操作将信号量-1，如果<0就将该线程移入队列；v操作将信号量+1，如果<=0，从队列中移出一个线程，将其唤醒，并将资源转交给它

1来互斥，0来同步就不再赘述了。

生产者和消费者问题



生产者/消费者问题描述：

1. 缓冲区大小是有限的
2. 生产者生产数据放入缓冲区中
3. 消费者从缓冲区中取出数据进行处理
4. 同一时刻生产者和消费者只能有一个访问缓冲区

从上述描述看，需要进行互斥和同步：互斥则是对临界区缓冲区的访问控制；同步则是缓冲区为空时，消费者需要等待生产者生产，相反若缓冲区满，生产者需要等消费者消费才能继续生产。

代码中是使用锁和条件变量来解决生产者和消费者问题，如果纯用信号量来解决描述如下：

定义3个信号量，一个用于访问缓冲区，初始值为1；一个用于消费者消耗资源，初始值为0；一个用于生产者查询缓冲区空位，初始值为缓冲区大小

生产者步骤为：首先对访问缓冲区的信号量进行P操作实现互斥，再对缓冲区空位的信号量进行P操作，再对消耗资源的信号量进行V操作，生产完一个就对访问缓冲区的信号量V操作，让消费者可以消费

消费者步骤为：首先对访问缓冲区的信号量进行P操作实现互斥，对消耗资源的信号量进行P操作，再对缓冲区空位的信号量进行V操作，生产完一个就对访问缓冲区的信号量V操作，让消费者可以消费

经典同步问题

哲学家就餐问题

哲学家就餐问题由迪杰斯特拉提出：

- 环境设置：五位哲学家围坐在一张圆桌周围，每位哲学家在桌子上有一盘意面。桌子中间有一碗无限供应的意大利面。
- 资源：每位哲学家的左右两边各放有一支筷子（总共五支筷子）。

- 行为：哲学家有两种行为，一是思考，二是就餐。思考时不需要任何资源，但就餐时需要同时拿起左右两边的筷子。
- 挑战：每位哲学家拿筷子的顺序不一定，可能会导致所有哲学家都拿着左边的筷子等待右边的筷子被释放，从而造成死锁。

方案1：由于每根筷子同一时刻只能被一个哲学家拿起，因此是互斥量，将每根筷子设置为信号量，初始值为1。哲学家在就餐之前需要拿起左边的和右边的筷子，也就是进行P操作，就餐完再V操作释放筷子。

但此方案会出现死锁，当所有哲学家都拿着左边的筷子或者拿着右边的筷子，等待另一边的筷子时。

方案2：上述现象出现主要是因为拿起左右两根筷子的动作不是原子操作，因此方案2选择在拿起左右两根筷子时加锁（或者使用信号量）。

但此方案同一时间只能有1个哲学家就餐，因为锁只有1个。

方案3：避免使用锁同时避免所有哲学家都先拿起同一边的筷子，让偶数位的哲学家先拿左边筷子再拿右边筷子；奇数位哲学家先拿右边筷子再拿左边筷子。

此方案可以满足同一时刻2位哲学家就餐，并且不会死锁。

方案4：把哲学家设置为信号量（初始值为0，用于同步），此外，同一时刻只能对一个哲学家进行操作，因此需要使用一个互斥量。定义3个状态（思考，饥饿，就餐），只有当左右两边的哲学家都没就餐并且当前哲学家为饥饿状态时，设置其状态为进餐，并使用V操作唤醒该哲学家表示可以进餐了。哲学家需要经历思考，拿起筷子，就餐，放下筷子四个步骤。拿起筷子的过程中，如果成功获取两边筷子，会使用V操作将该哲学家信号量+1，然后再进行P操作-1，执行后续步骤，否则在P操作时被阻塞。放下筷子时，需要通知左右两边的哲学家可以进餐。比较复杂。

```
#define N 5 // 哲学家个数
#define LEFT (i + N - 1) % N // i 的左边邻居编号
#define RIGHT (i + 1) % N // i 的右边邻居编号

#define THINKING 0 // 思考状态
#define HUNGRY 1 // 饥饿状态
#define EATING 2 // 进餐状态

int state[N]; // 数组记录每个哲学家的状态

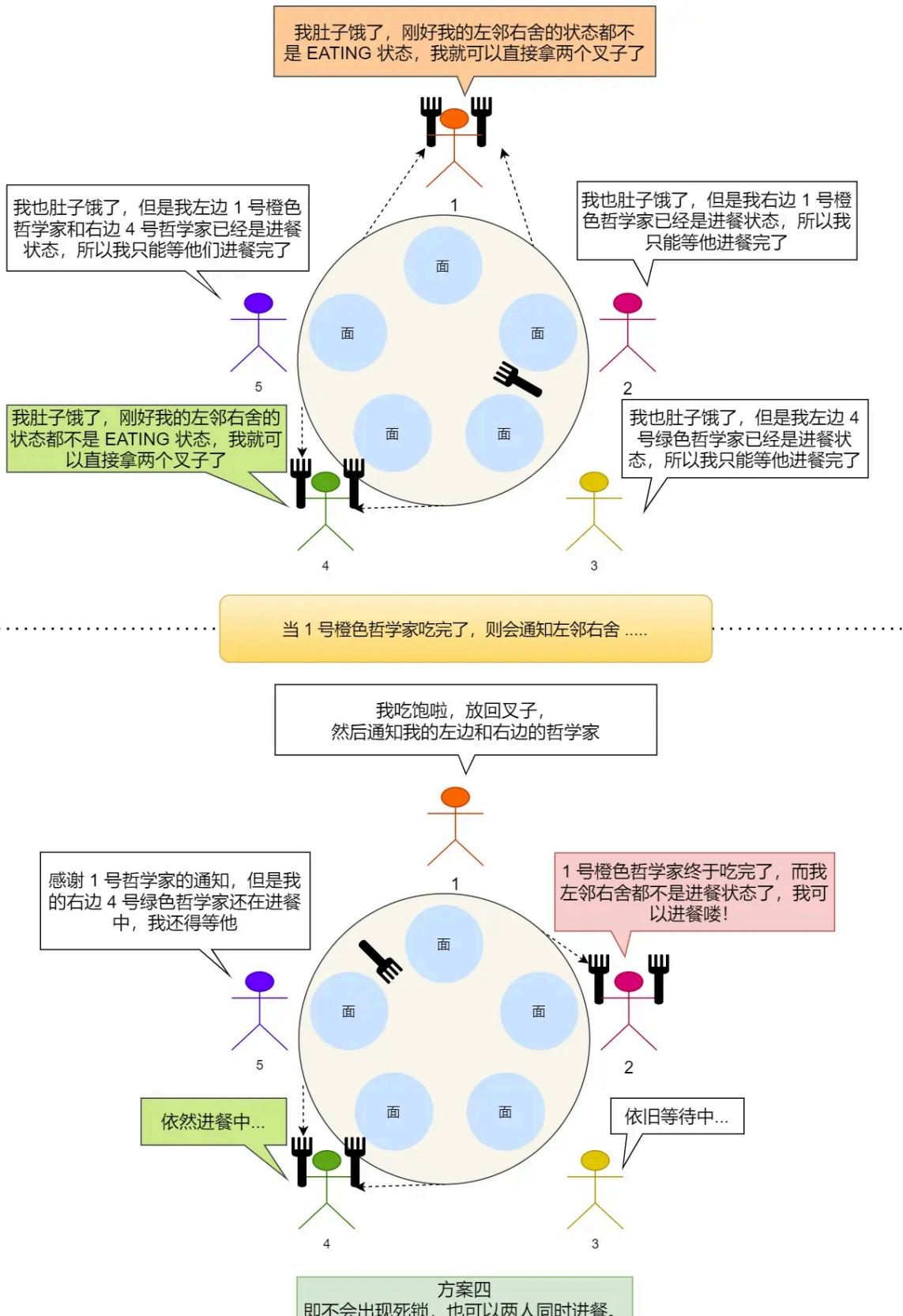
semaphore s[5]; // 每个哲学家一个信号量，初值 0
semaphore mutex; // 互斥信号量，初值为 1

void test(int i) // i 为哲学家编号 0-4
{
    // 如果 i 号的左边右边哲学家都不是进餐状态，把 i 号哲学家标记为进餐状态
    if (state[i] == HUNGRY &&
        state[LEFT] != EATING &&
        state[RIGHT] != EATING )
    {
        state[i] = EATING // 两把叉子到手，进餐状态
        V(s[i]); // 通知第 i 哲学家可以进餐了
    }
}

// 功能：要么拿到两把叉子，要么被阻塞起来
void take_forks(int i) // i 为哲学家编号 0-4
{
    P(mutex); // 进入临界区
    state[i] = HUNGRY; // 标记哲学家处于饥饿状态
    test(i); // 尝试获取 2 支叉子
    V(mutex); // 离开临界区
    P(s[i]); // 没有叉子则阻塞，有叉子则继续正常执行
}

// 功能：把两把叉子放回原处，并在需要的时候，去唤醒左邻右舍
void put_forks(int i) // i 为哲学家编号 0-4
{
    P(mutex); // 进入临界区
    state[i] = THINKING; // 吃完饭了，交出叉子，标记思考状态
    test(LEFT); // 检查左边的左邻右舍是否在进餐，没则唤醒
    test(RIGHT); // 检查右边的左邻右舍是否在进餐，没则唤醒
    V(mutex); // 离开临界区
}

// 哲学家主代码
void smart_person(int i) // i 为哲学家编号 0-4
{
    while(TRUE)
    {
        think(); // 思考
        take_forks(i); // 准备拿去叉子吃饭
        eat(); // 就餐
        put_forks(i); // 吃完放回叉子
    }
}
```



「哲学家进餐问题」对于互斥访问有限的竞争问题（如 I/O 设备）一类的建模过程十分有用。

读者-写者问题

读者-写者问题为数据库访问构建了一个模型：

- 读读：允许
- 读写：互斥，只有在没人写的情况下才能读
- 写写：互斥，不能同时写

方案1：读者优先方案，有读者在读，后续读者可以进入，而写者只有在没有读者的情况下写。并且只要有读者读就会立刻阻塞写者

- 1个记录当前读者数量的全局变量，初始为0
- 1个保证修改读者数量的全局变量互斥的信号量，初始为1
- 写者的互斥量
- 有读者读，并且此时全局变量为0时（避免重复P操作，仅需一次即可避免写者写），对写者互斥量进行P操作，阻塞写者
- 当前没有读者读时，对写者全局变量进行V操作，写者可写

方案2：写者优先方案，只要有写者写（第一个写者写需要等所有读者都读完了才能开始写），后续写者可以进入，但同一时刻只能一个写者写，而读者只有在没有写者的情况下写；读者操作和前面方案一样

（我怎么感觉这个也不算写者优先吧，毕竟读者一直读的话，写者也会饿死）

- 增加了一个记录写者数量的全局变量和一个互斥量来实现操作这个全局变量互斥



```
semaphore rCountMutex; // 控制对 Rcount 的互斥修改, 初始值为 1
semaphore rMutex; // 控制读者进入的互斥信号量者, 初始值为 1

semaphore wCountMutex // 控制 wCount 互斥修改, 初始值为 1
semaphore wDataMutex // 控制写者写操作的互斥信号量, 初始值为 1

int rCount = 0; // 正在进行读操作的读者个数, 初始化为 0
int wCount = 0; // 正在进行读操作的写者个数, 初始化为 0

// 写者进程/线程执行的函数
void writer()
{
    while(TRUE)
    {
        P(wCountMutex); // 进入临界区
        if ( wCount == 0 )
        {
            P(rMutex); // 当第一个写者进入, 如果有读者则阻塞读者
        }
        wCount++; // 写者计数 + 1
        V(wCountMutex); // 离开临界区

        P(wDataMutex); // 写者写操作之间互斥, 进入临界区
        write(); // 写数据
        V(wDataMutex); // 离开临界区

        P(wCountMutex); // 进入临界区
        wCount--; // 写完数据, 准备离开
        if ( wCount == 0 )
        {
            V(rMutex); // 最后一个写者离开了, 则唤醒读者
        }
        V(wCountMutex); // 离开临界区
    }
}

// 读者进程/线程执行的函数
void reader()
{
    while(TRUE)
    {
        P(rMutex);
        P(rCountMutex); // 进入临界区
        if ( rCount == 0 )
        {
            P(wDataMutex); // 当第一个读者进入, 如果有写者则阻塞写者写操作
        }
        rCount++;
        V(rCountMutex); // 离开临界区
        V(rMutex);

        read(); // 读数据

        P(rCountMutex); // 进入临界区
        rCount--;
        if ( rCount == 0 )
        {
            V(wDataMutex); // 当没有读者了, 则唤醒阻塞中写者的写操作
        }
        V(rCountMutex); // 离开临界区
    }
}
```

```
}
```

方案3：公平策略，相较于方案1，只需增加一个flag信号量实现互斥，避免只要有读者读就一直进入的权限。



```
semaphore rCountMutex; // 控制对 Rcount 的互斥修改，初始值为 1
semaphore wDataMutex; // 控制写者写操作的互斥信号量，初始值为 1
semaphore flag; // 用于实现公平竞争，初始值为 1
int rCount = 0; // 正在进行读操作的读者个数，初始化为 0

// 写者进程/线程执行的函数
void writer()
{
    while(TRUE)
    {
        P(flag);
        P(wDataMutex); // 写者写操作之间互斥，进入临界区
        write(); // 写数据
        V(wDataMutex); // 离开临界区
        V(flag);
    }
}

// 读者进程/线程执行的函数
void reader()
{
    while(TRUE)
    {
        P(flag);
        P(rCountMutex); // 进入临界区
        if ( rCount == 0 )
        {
            P(wDataMutex); // 当第一个读者进入，如果有写者则阻塞写者写操作
        }
        rCount++;
        V(rCountMutex); // 离开临界区
        V(flag);

        read(); // 读数据

        P(rCountMutex); // 进入临界区
        rCount--;
        if ( rCount == 0 )
        {
            V(wDataMutex); // 当没有读者了，则唤醒阻塞中写者的写操作
        }
        V(rCountMutex); // 离开临界区
    }
}
```

死锁

构成死锁需同时满足的4个条件：

- 互斥条件
- 持有并等待
- 不可剥夺
- 环路等待

通过一个例子来说明上述4个条件，进程A需要资源1和资源2才能完成任务，而进程B也需要资源1和资源2才能完成任务；若此时进程A持有资源1，需要资源2，进程B持有资源2，需要资源1，A和B构成死锁。

互斥条件：资源1和资源2在同一时刻都只能由一个进程获取

持有并等待：进程A持有资源1，等待资源2，不完成任务前不退出；进程B同理

不可剥夺：资源1在进程A完成前不可剥夺

环路等待：进程A和进程B构成环路

利用工具排查死锁问题

排查死锁可以通过代码审查，C/C++可以使用GDB的调试功能，你可以在运行时检查线程的状态和锁的持有情况。虽然它不自动检测死锁，但你可以手动查看锁的持有者和等待这些锁的线程。使用info threads查看所有线程，thread 切换到特定线程，info lock查看锁的状态。

避免死锁问题的产生

避免死锁问题的产生，只需要破坏上面的4个条件之一即可，最常用的是使用**资源有序分配来破坏环路等待条件**。也就是进程A和进程B都先获取资源1再获取资源2。

银行家算法

银行家算法（Banker's Algorithm）是一种避免死锁和确保系统资源分配安全性的算法，由Dijkstra提出。这种算法通过模拟银行家发放贷款的方式来处理多个进程请求有限资源的问题。银行家在贷款时需要确保即使在最坏的情况下也能够收回贷款，类似地，银行家算法确保即使在最坏的情况下系统也不会进入死锁状态。

银行家算法的基本概念：

- 最大需求（Max）：每个进程可能请求的最大资源数量。
- 分配（Allocation）：当前每个进程已被分配的资源数量。
- 需求（Need）：每个进程还需要多少资源才能完成。计算公式为 $Need[i] = Max[i] - Allocation[i]$ 。
- 可用（Available）：系统中当前可用的资源数量。

银行家算法的工作流程：

1. 初始化资源：系统启动时，定义每种资源的总量。
2. 进程请求资源：进程提出资源请求。
3. 资源分配尝试：系统尝试预分配资源给请求进程，并进入安全性算法检查。
4. 安全性算法：系统检查是否存在至少一条使所有进程顺序结束的安全序列。如果存在，预分配资源变为正式分配。
5. 进程释放资源：进程完成后释放资源，系统将这些资源标记为可用。

6. 重复以上步骤：继续响应新的资源请求。

银行家算法的例子：

假设有两种资源类型A和B，可用数量分别为10和5。有三个进程P1、P2、P3，各自的最大需求和当前分配如下表：

进程	最大需求A	最大需求B	已分配A	已分配B
P1	7	5	3	2
P2	3	2	2	1
P3	9	0	2	0

计算需求和可用资源：

1. 需求P1 = (7-3, 5-2) = (4, 3)
2. 需求P2 = (3-2, 2-1) = (1, 1)
3. 需求P3 = (9-2, 0-0) = (7, 0)
4. 可用资源 = 总资源 - 已分配资源 = (10-7, 5-3) = (3, 2)

安全性检查：

假设P2请求(1, 1)，系统尝试分配后：

1. 分配给P2 = (3, 2)
2. 可用资源 = (3-1, 2-1) = (2, 1)
3. 此时，系统检查安全性。P2的需求已被满足，可以完成执行并释放资源。释放后系统可用资源为(5, 3)。这足以满足P1或P3的剩余需求。因此，存在安全序列[P2, P1, P3]，这次资源请求是安全的。

银行家算法通过预防性地检查资源分配后是否存在一个安全的完成序列，从而避免系统进入死锁状态。这种方法虽然可以保证系统的安全性，但在实际操作中可能因为其算法复杂度和资源使用效率较低而不被广泛使用。

线程崩溃会导致进程崩溃吗？

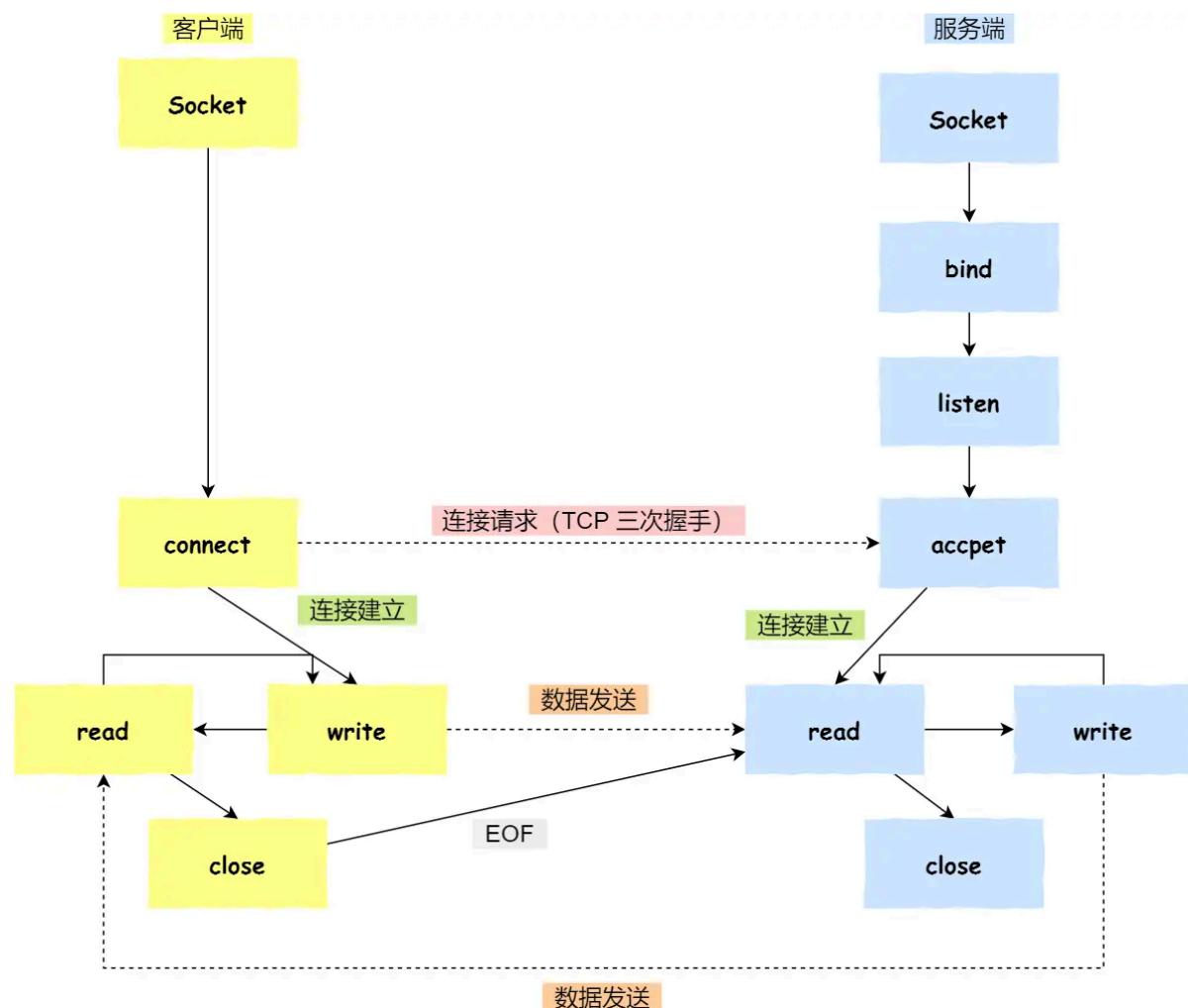
取决于线程崩溃的原因和如何处理这种情况。由于进程中的线程共享地址空间，一个线程的行为可能会影响整个进程。如果一个线程因为未捕获的异常（如访问非法内存、除零错误等）而崩溃，那么通常这会导致整个进程终止。这是因为这类异常通常会引发信号（如SIGSEGV、SIGABRT等），**如果这些信号没有被适当处理，操作系统默认的行为是终止整个进程。**

比如当我们使用kill命令终止一个进程时（普通用户只能kill用户为（UGO）自己的进程，程序的U为创建它的用户），其背后的机制如下

- CPU 执行正常的进程指令
- 调用 kill 系统调用向进程发送信号
- 进程收到操作系统发的信号，CPU 暂停当前程序运行，并将控制权转交给操作系统
- 调用 kill 系统调用向进程发送信号（假设为 11，即 SIGSEGV，一般非法访问内存报的都是这个错误）
- 操作系统根据情况执行相应的信号处理程序（函数），一般执行完信号处理程序逻辑后会让进程退出

如果进程没有注册自己的信号处理函数，那么操作系统会执行默认的信号处理程序（一般最后会让进程退出），但如果注册了，则会执行自己的信号处理函数，这样的话就给了进程一个垂死挣扎的机会，它收到 kill 信号后，可以调用 exit() 来退出，但也可以使用 sigsetjmp, siglongjmp 这两个函数来恢复进程的执行。也就是说虽然给进程发送了 kill 信号，但如果进程自己定义了信号处理函数或者无视信号就有机会逃出生天，当然了 kill -9 命令例外，不管进程是否定义了信号处理函数，都会马上被干掉。

针对 TCP 应该如何 Socket 编程？



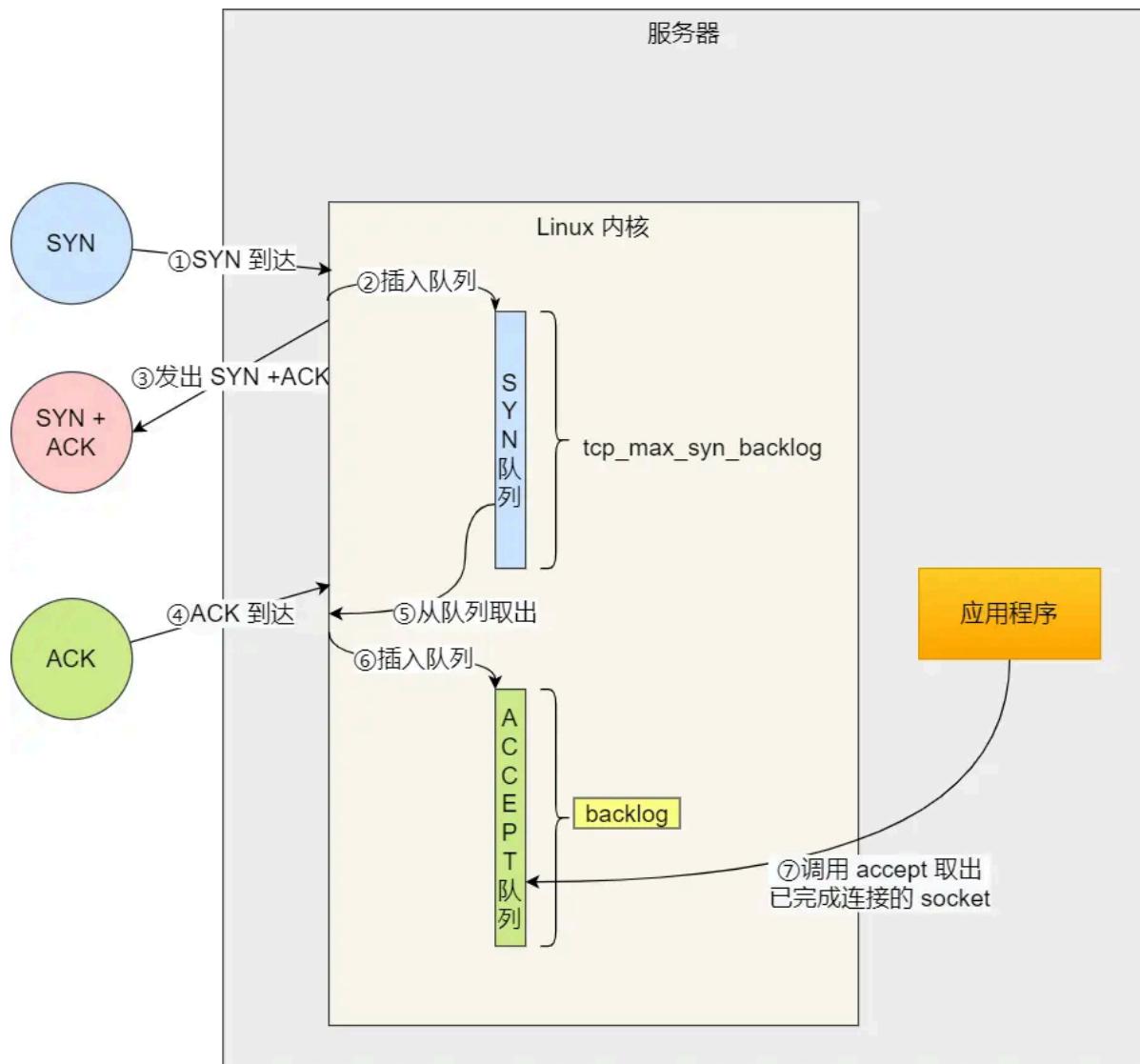
- 服务端和客户端初始化 `socket`，得到文件描述符；
- 服务端调用 `bind`，将 socket 绑定在指定的 IP 地址和端口；
- 服务端调用 `listen`，进行监听；
- 服务端调用 `accept`，等待客户端连接；
- 客户端调用 `connect`，向服务端的地址和端口发起连接请求；
- 服务端 `accept` 返回用于传输的 `socket` 的文件描述符；
- 客户端调用 `write` 写入数据；服务端调用 `read` 读取数据；
- **客户端断开连接时，会调用 `close`，那么服务端 `read` 读取数据的时候，就会读取到了 `EOF`，待处理完数据后，服务端调用 `close`，表示连接关闭。**

这里需要注意的是，服务端调用 `accept` 时，连接成功了会返回一个已完成连接的 socket，后续用来传输数据。所以，监听的 socket 和真正用来传送数据的 socket，是「两个」 socket，一个叫作**监听 socket**，一个叫作**已完成连接 socket**。成功连接建立之后，双方开始通过 `read` 和 `write` 函数来读写数据，就像往一个文件流里面写东西一样。

listen 时候参数 backlog 的意义？

Linux内核中会维护两个队列：

- 半连接队列（SYN 队列）：接收到一个 SYN 建立连接请求，处于 `SYN_RECV` 状态；
- 全连接队列（Accept 队列）：已完成 TCP 三次握手过程，处于 `ESTABLISHED` 状态；



```
int listen (int sockfd, int backlog)
```

- 参数一 sockfd 为 sockfd 文件描述符
- 参数二 backlog，这参数在历史版本有一定的变化

在早期 Linux 内核 backlog 是 SYN 队列大小，也就是未完成的队列大小。在 Linux 内核 2.2 之后，backlog 变成 accept 队列，也就是已完成连接建立的队列长度，所以现在通常认为 backlog 是 accept 队列。**但是上限值是内核参数 somaxconn 的大小，也就是说 accpet 队列长度 = min(backlog, somaxconn)。**

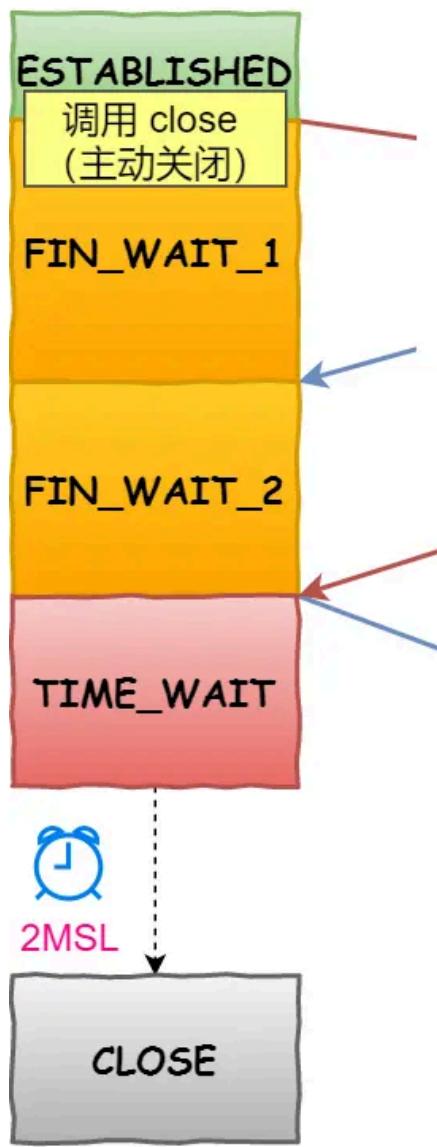
accept 发生在三次握手的哪一步？

客户端 connect 成功返回是在第二次握手，服务端 accept 成功返回是在三次握手成功之后。

客户端调用 close 了，连接是断开的流程是什么？

我们看看客户端主动调用了 `close`，会发生什么？

客户端



服务端

FIN

ACK

FIN

ACK

CLOSE

调用 close

LAST_ACK

CLOSE

- 客户端调用 `close`，表明客户端没有数据需要发送了，则此时会向服务端发送 FIN 报文，进入 FIN_WAIT_1 状态；
- 服务端接收到了 FIN 报文，TCP 协议栈会为 FIN 包插入一个文件结束符 `EOF` 到接收缓冲区中，应用程序可以通过 `read` 调用来感知这个 FIN 包。这个 `EOF` 会被放在已排队等候的其他已接收的数据之后，这就意味着服务端需要处理这种异常情况，因为 `EOF` 表示在该连接上再无额外数据到达。此时，服务端进入 CLOSED_WAIT 状态；
- 接着，当处理完数据后，自然就会读到 `EOF`，于是也调用 `close` 关闭它的套接字，这使得服务端发出一个 FIN 包，之后处于 LAST_ACK 状态；
- 客户端接收到服务端的 FIN 包，并发送 ACK 确认包给服务端，此时客户端将进入 TIME_WAIT 状态；
- 服务端收到 ACK 确认包后，就进入了最后的 CLOSE 状态；
- 客户端经过 2MSL 时间之后，也进入 CLOSE 状态；

没有 accept，能建立 TCP 连接吗？

答案：**可以的。** accpet 系统调用并不参与 TCP 三次握手过程，它只是负责从 TCP 全连接队列取出一个已经建立连接的 socket，用户层通过 accpet 系统调用拿到了已经建立连接的 socket，就可以对该 socket 进行读写操作了。

没有 listen，能建立 TCP 连接吗？

答案：**可以的。** 客户端是可以自己连自己的形成连接（TCP自连接），也可以两个客户端同时向对方发出请求建立连接（TCP同时打开），这两个情况都有个共同点，就是**没有服务端参与，也就是没有 listen，就能 TCP 建立连接。**

最基本的socket模型

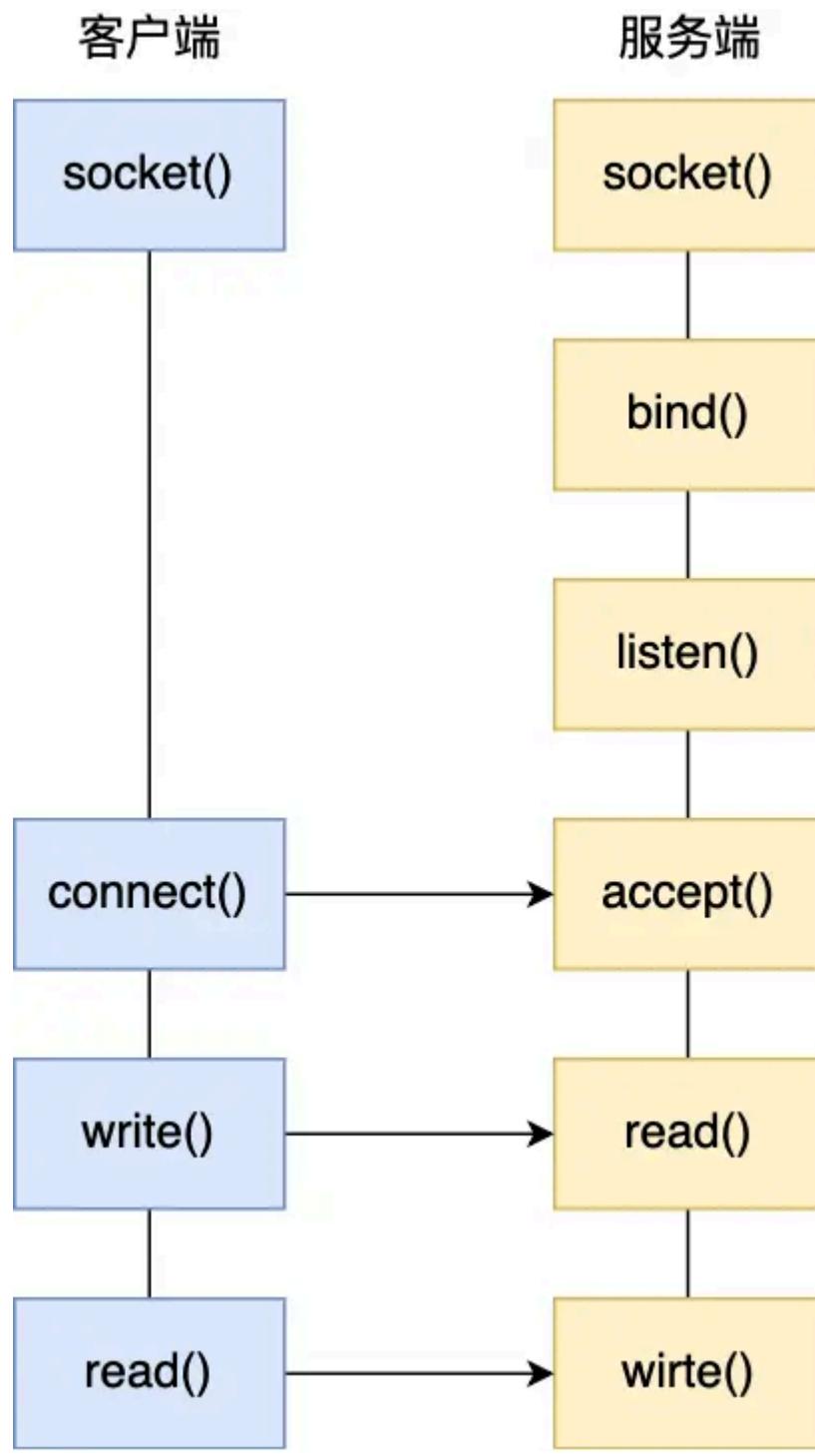
通过socket可以在客户端和服务端中通信，实现进程间的跨主机通信。双方进行通信前需要各自创建一个socket，用于读取和发送数据。创建 socket的时候，可以指定网络层使用的是IPv4还是IPv6，传输层使用的是TCP还是UDP。相较于TCP，UDP的socket编程更加简单。

服务端和客户端的socket编程：

- 首先通过socket()创建一个基于IPV4/TCP的socket，调用bind()给这个socket绑定IP:Port。
 - 绑定端口的目的：内核收到tcp报文后，会检查里面的端口号，通过这个端口号把数据转发到对应的应用程序
 - 绑定IP地址的目的：一台计算机可能有多个网卡，每个网卡的都有对应的IP地址，只有绑定了网卡，内核才会把对应网卡的数据转发至应用程序
- 绑定成功后，服务端通过listen()进行监听
- 进入监听状态后，调用accept()函数等待客户端的连接，如果没有客户端的连接则会阻塞
- 而客户端在创建好socket后，则通过connect()发起连接，该函数需要指明服务端的IP:Port，然后就是TCP的三次握手
- 在三次握手的过程中，服务器的内核为每个socket维护了两个队列：TCP半连接队列（没有完成三次握手，服务端处于syn_rcvd）和TCP全连接队列（完成三次握手，服务端处于established）
- 当全连接队列不为空时，accept()从其中取出一个已完成连接的socket返回应用程序，后续和客户端的数据传输都使用这个socket
- 建立连接后，双方可通过read()和write()来读写数据（或recv()和send()）
- 通信结束后，双方应通过close()函数关闭socket，以释放系统资源

从上述流程中可以看出，客户端和服务端的通信需要2个socket：

1. 用于监听的socket
2. 已连接的socket（用于读写数据）



得益于“Linux一切皆文件”的理念，我们对socket进行读写，本质是在对文件进行读写，每个socket都有对应的文件描述符。

task_struct, 文件描述符和socket_struct

在Linux操作系统中，`task_struct`是一个非常重要的结构体，用于描述一个进程的状态。这个结构体包含了操作系统需要的几乎所有信息来管理进程，每个进程在内核中都有一个对应的`task_struct`实例。`task_struct`结构体的字段非常多，以下是一些主要的字段：

- `state`: 表示进程的状态（例如，运行、可中断睡眠、不可中断睡眠、停止、僵尸等）。
- `pid`: 进程的唯一标识符。
- `priority`: 进程的动态优先级。
- `static_prio`: 进程的静态优先级。
- `normal_prio`: 进程的普通优先级。

- parent: 指向父进程的指针。
- children: 进程的子进程列表。
- sibling: 同一个父进程的其他子进程链表。
- mm: 指向内存描述符mm_struct的指针，包含进程的所有内存信息。
- files: 指向打开文件描述符的结构体。
- fs: 用于文件系统导航的信息。
- signal: 信号相关的信息。
- sighand: 信号处理的结构体。
- thread: 线程特定的信息，如寄存器状态等。
- task_list: 用于将进程链接到全局进程列表的链表节点。

task_struct中有一个指向「文件描述符数组」的成员指针，该数组里列出这个进程打开的所有文件的文件描述符。数组的下标是文件描述符，是一个整数，而数组的内容是一个指针，指向内核中所有打开的文件的列表，内核可以通过文件描述符找到对应打开的文件。

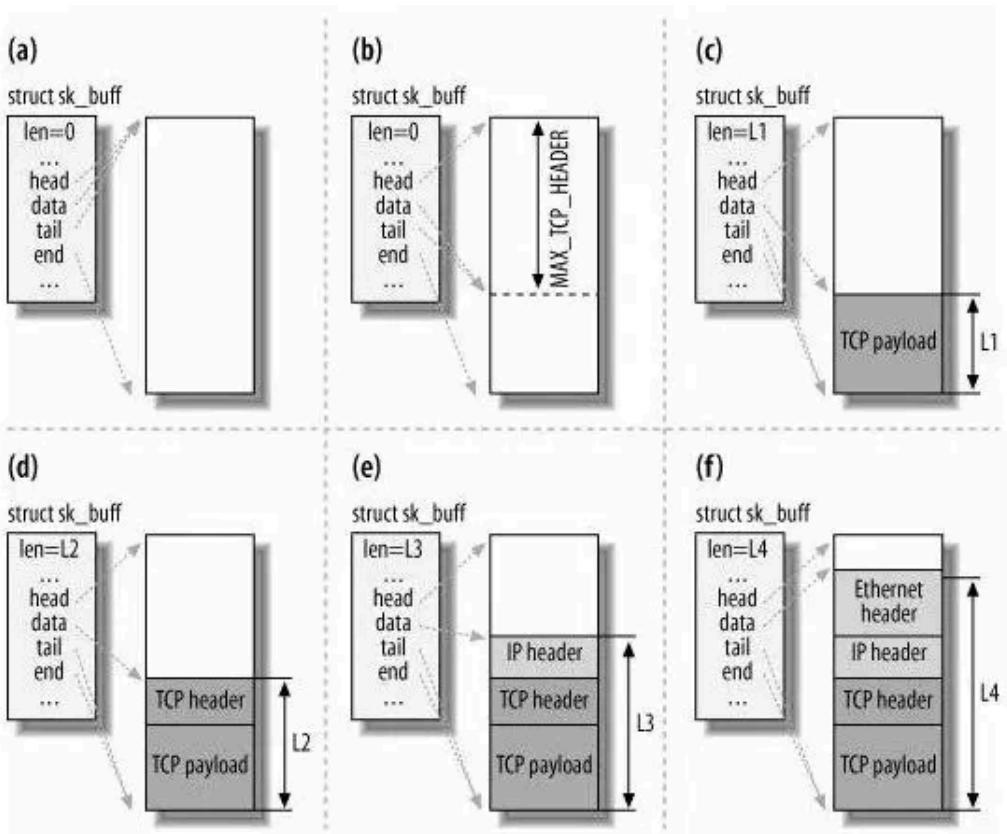
每个进程都有自己独立的一个从零开始的文件描述符空间，所以如果运行了两个不同的程序，对应两个不同的进程，如果它们都打开一个文件，它们或许可以得到相同数字的文件描述符，但是因为内核为每个进程都维护了一个独立的文件描述符空间，这里相同数字的文件描述符可能会对应到不同的文件。

Linux中每个文件/目录都有一个inode (inode包含了关于文件的大部分信息，但不包括文件名和文件数据)，Socket文件的inode指向了内核中的Socket结构，在这个结构体里有两个队列，分别是发送队列和接收队列，这个两个队列里面保存的是一个个**struct sk_buff**，用链表的组织形式串起来。**sk_buff**=可以表示各个层的数据包，在应用层数据包叫data，在TCP层我们称为segment，在IP层我们叫packet，在数据链路层称为frame。

为什么全部数据包只用一个结构体来描述呢？协议栈采用的是分层结构，上层向下层传递数据时需要增加包头，下层向上层数据时又需要去掉包头，**如果每一层都用一个结构体，那在层之间传递数据的时候，就要发生多次拷贝，这将大大降低CPU效率**。于是，为了在层级之间传递数据时，不发生拷贝，通过调整**sk_buff**中**data**的指针（指向当前正在处理的数据的起始位置），从而实现一个结构体来描述所有的网络包。比如：

- 当接收报文时，从网卡驱动开始，通过协议栈层层往上传送数据报，通过增加**skb->data**的值，来逐步剥离协议首部。
- 当要发送报文时，创建**sk_buff**结构体，数据缓存区的头部预留足够的空间，用来填充各层首部，在经过各下层协议时，通过减少**skb->data**的值来增加协议首部。

当然不只是通过**data**，其他的指针诸如**len**和**tail**也是在变化的，这里的增加和减少是相对于顶部而言



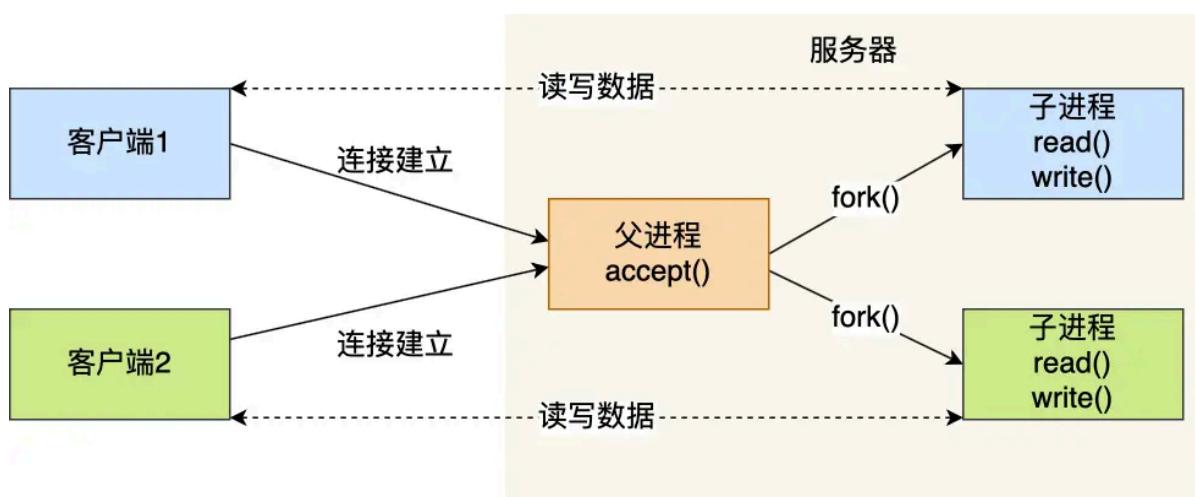
但上述模型采用同步阻塞的方式使得同一时间只能服务一个用户。此外根据TCP连接的四元组：本机IP，本机端口，对端IP，对端端口，理论上服务端单机最大TCP连接数约为 $2^{32} \times 16$ （ $2^{32} \times 16$ ，目标IP和目标端口号进行组合），但现实受限于以下两个条件

- 单个进程打开的文件描述符是有限的，默认是1024
- 每个连接需要消耗一定的系统内存和资源

多进程模型

相较于上述同步阻塞型socket模型，如果服务器要同时支持多个客户端，最传统的方法则是多进程模型，为每个客户端分配一个进程来处理请求。

多进程模型中，主进程通过`accept()`监听客户端的连接，一旦连接完成，`accpet()`将会返回一个已连接socket。主进程通过`fork()`创建一个子进程来处理对该客户端的请求。`fork()`会把父进程的东西都复制一遍，例如文件描述符、代码和地址内存空间等。因此子进程此时可以直接与客户端通信并处理请求。

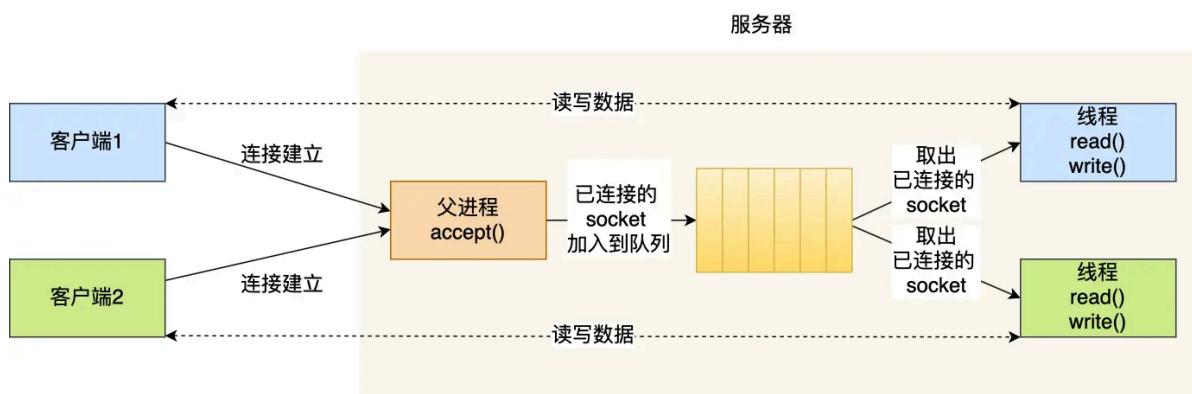


子进程退出时，主进程需要及时对其资源进行回收（可以通过wait和waitpid），否则将会成为僵尸进程，从而消耗系统资源。多进程模型只能应对少量的客户端连接，因为进程上下文的切换是需要代价的。

多线程模型

相较于多进程模型，我们可以在单进程中可以运行多个线程，同进程里的线程可以共享进程的部分资源，比如文件描述符列表、进程空间、代码、全局数据、堆、共享库等，这些共享些资源在上下文切换时不需要切换，而只需要切换线程的私有数据、寄存器等不共享的数据，因此同一个进程下的线程上下文切换的开销要比进程小得多。

服务器通过TCP和客户端连接后，通过pthread_create()函数创建线程，然后将已连接Socket的文件描述符传递给线程函数，接着在线程里和客户端进行通信，从而达到并发处理的目的。为了避免线程的频繁创建和销毁，可以提前创建若干个线程也就是线程池，当与客户端建立连接后，将已连接socket放入队列，线程池中的线程从队列中取出socket进行处理。当然由于这个队列是全局的，还需要进行并发控制，比如信号量和锁等内容。



不管是多进程模型还是多线程模型，每建立一个TCP连接就需要一个进程或线程进行处理，对于C10K问题，则需要维护1万个进程或线程，这是不可接受的。

I/O多路复用

解决上述问题的办法则是通过I/O多路复用，也就是使用一个进程来维护多个socket。CPU虽然一次只能处理一个请求，但处理一次请求的时间很短，1s内可以处理上千个请求，把时间拉长来看相当于多个请求复用了一个进程，和os的进程并发类似，也叫时分多路复用。内核提供的I/O多路复用包括select/poll和epoll，可以通过一个系统调用从内核中获取多个事件，通过将所有连接（文件描述符）传给内核，再由内核返回产生的事件的连接，在用户态进行处理。

(I/O多路复用是同步阻塞的，其内部是非阻塞的)

select/poll

select实现多路复用的过程可以简单阐述如下：select将所有已连接的socket放入一个文件描述符集合，再通过select将文件描述符集合拷贝至内核，内核对这个集合的所有文件描述符进行遍历，发现有事件产生则将对应的socket标记为可读或可写。再将文件描述符集合拷贝至用户态，用户态对这个集合进行再次遍历找到可读或可写的socket进行处理。select的过程中涉及2次对文件描述符集合的遍历和拷贝。此外，select使用固定长度的BitsMap，表示文件描述符集合，而且所支持的文件描述符的个数是有限制的，在Linux系统中，由内核中的FD_SETSIZE限制，默认最大值为1024，只能监听0~1023的文件描述符。

相较于select, poll使用动态数组，以链表的形式来组织，突破了位图固定长度的限制，只受限于系统资源和内存（但受限于系统描述符总数的限制）。poll和select并没有太大的本质区别，都是使用线性结构存储进程关注的socket集合（只不过一个数量有限，一个“无限”），都需要遍历文件描述符集合来找到可读或可写的socket，时间复杂度为O(n)，而且也需要在用户态与内核态之间拷贝文件描述符集合，这种方式随着并发数上来，性能的损耗会呈指数级增长。

epoll

epoll是对poll的进一步改进，特别是在处理大量文件描述符时。epoll使用一种更为高效的机制，它只关注活跃的文件描述符，而不是遍历整个文件描述符集：

- epoll使用事件通知方式，只返回活动的文件描述符，减少了不必要的检查和内存拷贝。
- epoll适合处理大规模文件描述符，因为它的性能几乎不受监视的文件描述符数量的影响。

```
int s = socket(AF_INET, SOCK_STREAM, 0);
bind(s, ...);
listen(s, ...)

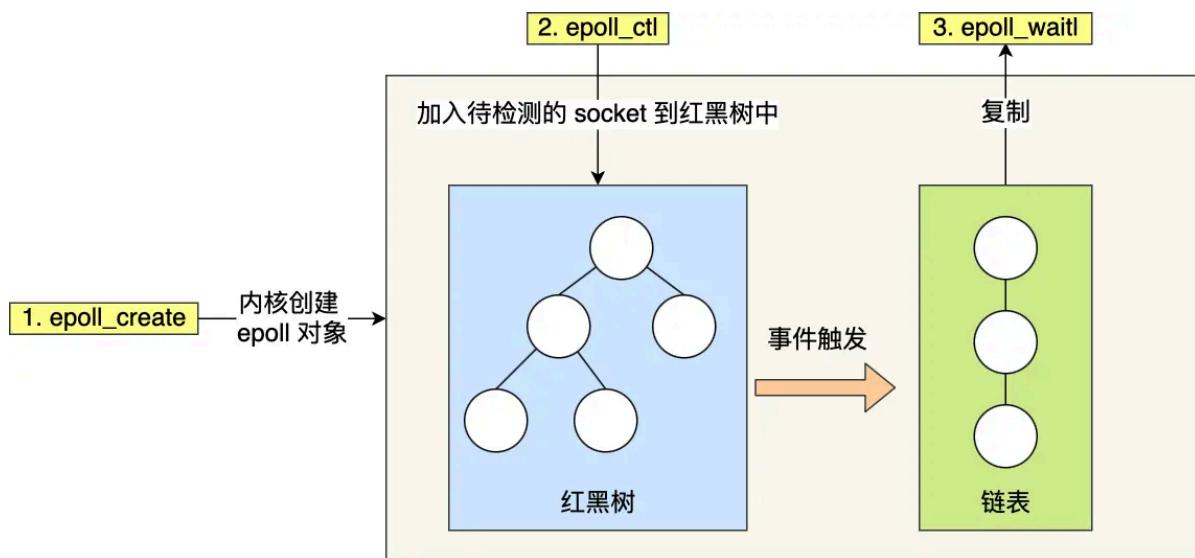
int epfd = epoll_create(...);
epoll_ctl(epfd, ...); //将所有需要监听的socket添加到epfd中

while(1) {
    int n = epoll_wait(...);
    for(接收到数据的socket){
        //处理
    }
}
```

epoll的使用方法如上所示，首先通过epoll_create创建一个epfd对象，将所有需要监视的socket放入这个epfd对象中，最后调用epoll_wait等待事件通知再进行处理。

epoll是怎样解决select/poll的问题呢？

1. epoll在内核里使用红黑树来跟踪进程所有待检测的文件描述字，把需要监控的socket通过epoll_ctl()函数加入内核中的红黑树里，红黑树是个高效的数据结构，增删改一般时间复杂度是O(logn)。而select/poll内核里没有类似epoll红黑树这种保存所有待检测的socket的数据结构，所以select/poll每次操作时都传入整个socket集合给内核，而epoll因为在内核维护了红黑树，可以保存所有待检测的socket，所以只需要传入一个待检测的socket，减少了内核和用户空间大量的数据拷贝和内存分配。
2. epoll使用事件驱动的机制，内核里维护了一个链表来记录就绪事件，当某个socket有事件发生时，通过回调函数内核会将其加入到这个就绪事件列表中，当用户调用epoll_wait()函数时，只会返回有事件发生的文件描述符的个数，不需要像select/poll那样轮询扫描整个socket集合，大大提高了检测的效率。



epoll的方式即使监听的Socket数量越多的时候，效率不会大幅度降低，能够同时监听的Socket的数目上限就为系统定义的进程打开的最大文件描述符个数。

边缘触发和水平触发

epoll支持两种事件触发模式，分别是边缘触发（edge-triggered, ET）和水平触发（level-triggered, LT）。

- 边缘触发：当被监控的socket有事件发生时，例如可读或可写，服务端只会从epoll_wait中苏醒一次。因此需要保证一次性将内核缓冲区的数据读取完毕。
- 水平触发：被监控的socket有事件发生时，服务端会不断的从epoll_wait苏醒，直到把内核缓冲区的数据读取完毕。

以取快递为例，边缘触发相当于你有一个快递到了，只给你发一次消息，即使你一直没取；而水平触发则是只要你没取就一直给你发消息。

如果使用边缘触发模式，I/O事件发生时只会通知一次，而且我们不知道到底能读写多少数据，所以在收到通知后应尽可能地读写数据，以免错失读写的机会。因此，我们会循环从文件描述符读写数据，那么如果文件描述符是阻塞的，没有数据可读写时，进程会阻塞在读写函数那里，程序就没办法继续往下执行。所以，边缘触发模式一般和非阻塞I/O搭配使用，程序会一直执行I/O操作，直到系统调用（如 read 和 write）返回错误，错误类型为EAGAIN或EWOULDBLOCK。一般来说，边缘触发的效率比水平触发的效率要高，因为边缘触发可以减少epoll_wait的系统调用次数。select/poll只有水平触发模式，epoll默认的触发模式是水平触发，但是可以根据应用场景设置为边缘触发模式。另外，使用I/O多路复用时，最好搭配非阻塞I/O一起使用（返回的事件可能是不可读写的，比如发生错误被丢弃）。

水平触发可以使用阻塞I/O和非阻塞I/O，而边缘触发必须使用非阻塞I/O（否则会在最后一次调用没有数据读写时阻塞）。

reactor, proactor与epoll的关系

Reactor和Proactor模式都是处理并发 I/O 的模式，但它们在处理I/O事件的方式上有所不同。Reactor主要用于同步非阻塞I/O，而Proactor用于异步I/O。epoll 是这些模式的一个实现工具，它提供了高效的事件通知机制，可以被用于实现Reactor或Proactor模式。

reactor

reactor模式主要由reactor和处理资源池组成：

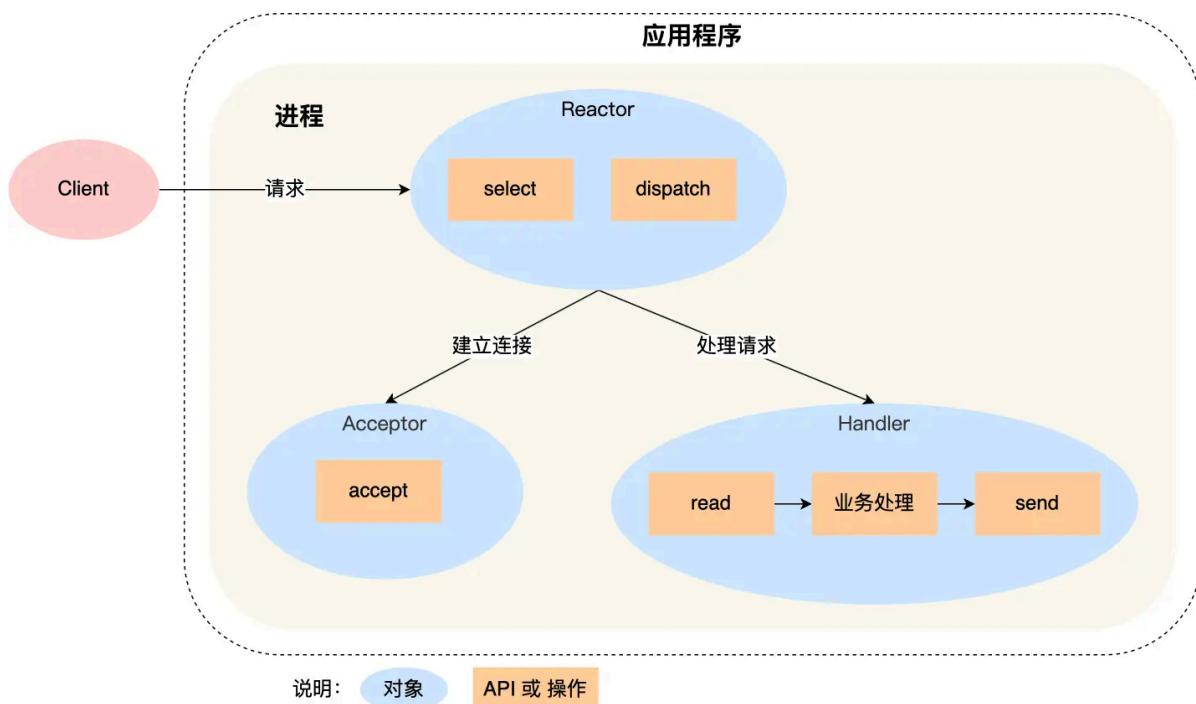
- reactor负责监听和分发事件，比如连接事件和I/O事件
- 处理资源池负责如何处理事件

reactor可以是单个或多个，处理资源池可以是单个进程/线程，也可以是多个进程/线程。reactor模式主要可分为：

1. 单reactor 单进程/线程
2. 单reactor 多进程/线程
3. 多reactor 单进程/线程

单reactor 单进程/线程

以经典的C/S架构为例，单reactor 单进程/线程模型的示意图如下：



上图主要可分为reactor, acceptor和handler三个组件：

1. reactor用于监听和分发事件
2. acceptor用于获取连接
3. handler用于处理业务

单reactor 单进程/线程模型流程可描述如下：

1. reactor通过I/O多路复用接口（图中是select）监听事件，并根据不同的事件类型分发（dispatch）给不同的处理对象

2. 如果是连接事件，则交付给acceptor处理，acceptor通过accept建立连接，并创建一个handler对象处理后续的响应事件
3. 如果是业务事件，则交付给handler对象进行处理，比如读->业务处理->发送

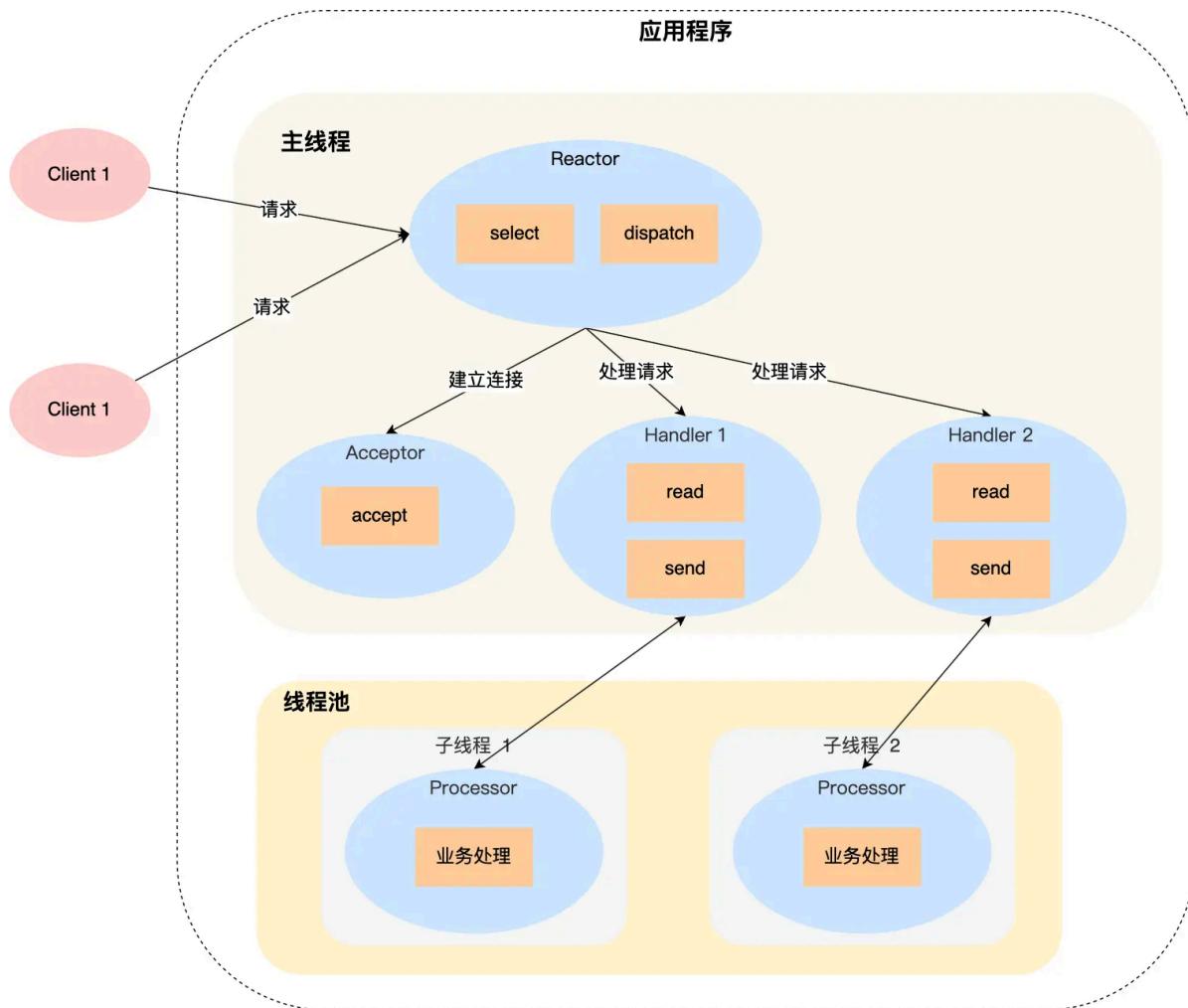
单reactor 单进程/线程实现较为简单，监听，分发处理事件都是在一个进程/线程中进行的，不用考虑进程/线程间通信。但也存在一些问题：

1. 无法充分利用多核CPU的性能
2. handler对象处理业务时，无法进行其他连接，如果业务处理较为复杂，则可能造成一定的响应延迟

redis 6.0版本之前采用的是单reactor 单进程的方案，因为redis在内存中进行处理。

单reactor 多进程/多线程

单reactor 多进程/线程模型的示意图如下：



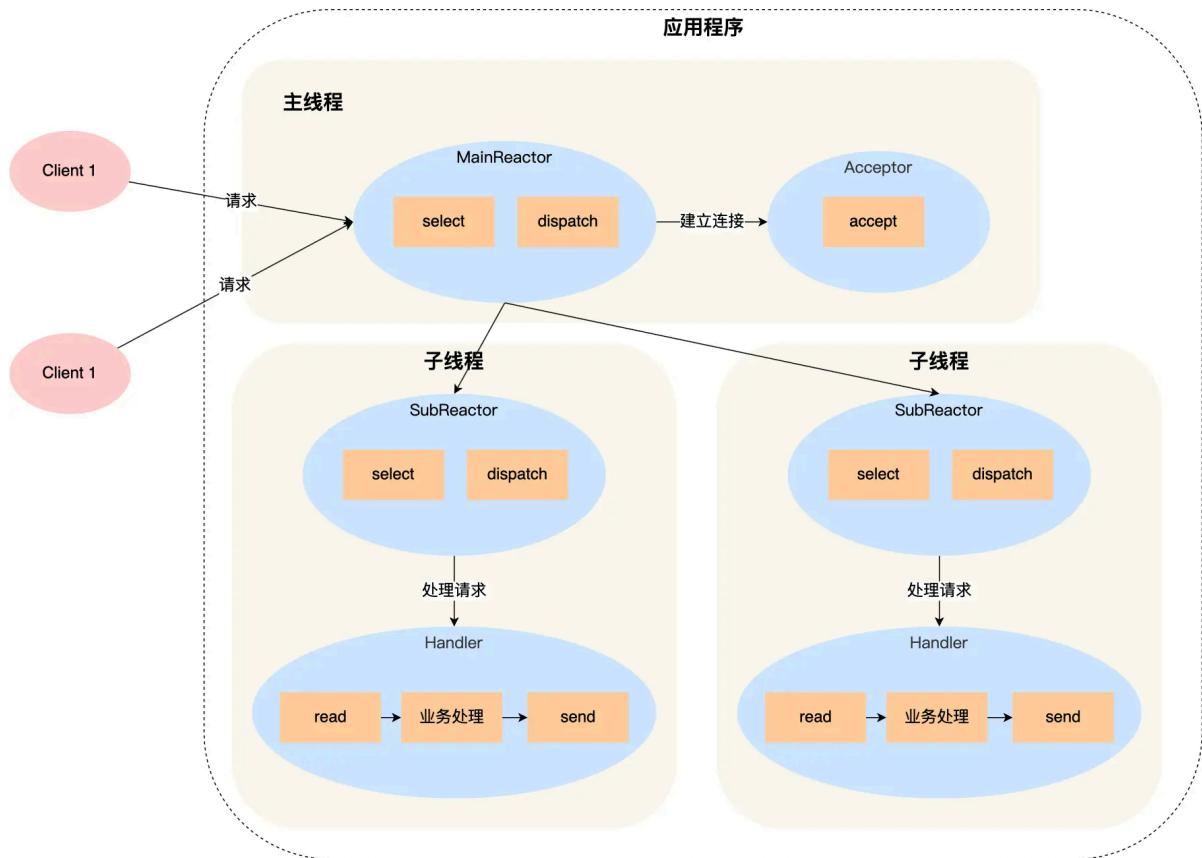
相较于单reactor 单进程/线程，单reactor 多进程/多线程的变化如下：

1. handler不再负责业务处理，而是交予线程池中的线程进行，只负责数据的读与发送。

由于采用了多线程，因此需要使用互斥锁等操作来避免多线程对共享资源的竞争。此外，多进程的开销远大于多线程的开销了，所以此模型一般是单reactor 多线程。虽然单reactor 多线程模型可以很好的利用多核CPU的性能优势，但由于只有一个reactor对象负责事件的监听和分发，因此遇到突然的高并发事件时，可能会出现性能瓶颈。

多reactor 多进程/线程

多reactor 多进程/线程模型的示意图如下：



多reactor 多进程/线程模型的流程可描述为：

1. mainreactor监控连接事件对象，当发生连接事件时，交付给acceptor建立连接，并将建立的连接分发给subreactor
2. subreactor继续监听分发的连接，监听到有业务处理事件时，创建一个handler对象来处理业务事件

多reactor 多进程/线程模型其实比单reactor 多线程模型简单：

1. mainreactor只负责新事件的连接，具体的业务处理由subreactor进行
2. mainreactor和subreactor的交互也很简单，无需subreactor返回

Netty和Memcache都采用了多 Reactor 多线程的方案。采用了多 Reactor 多进程方案的开源软件是 Nginx，不过方案与标准的多Reactor多进程有些差异。具体差异表现在主进程中仅仅用来初始化 socket，并没有创建mainReactor来accept连接，而是由子进程的Reactor来accept连接，通过锁来控制一次只有一个子进程进行accept（防止出现惊群现象），子进程accept新连接后就放到自己的Reactor进行处理，不会再分配给其他子进程。

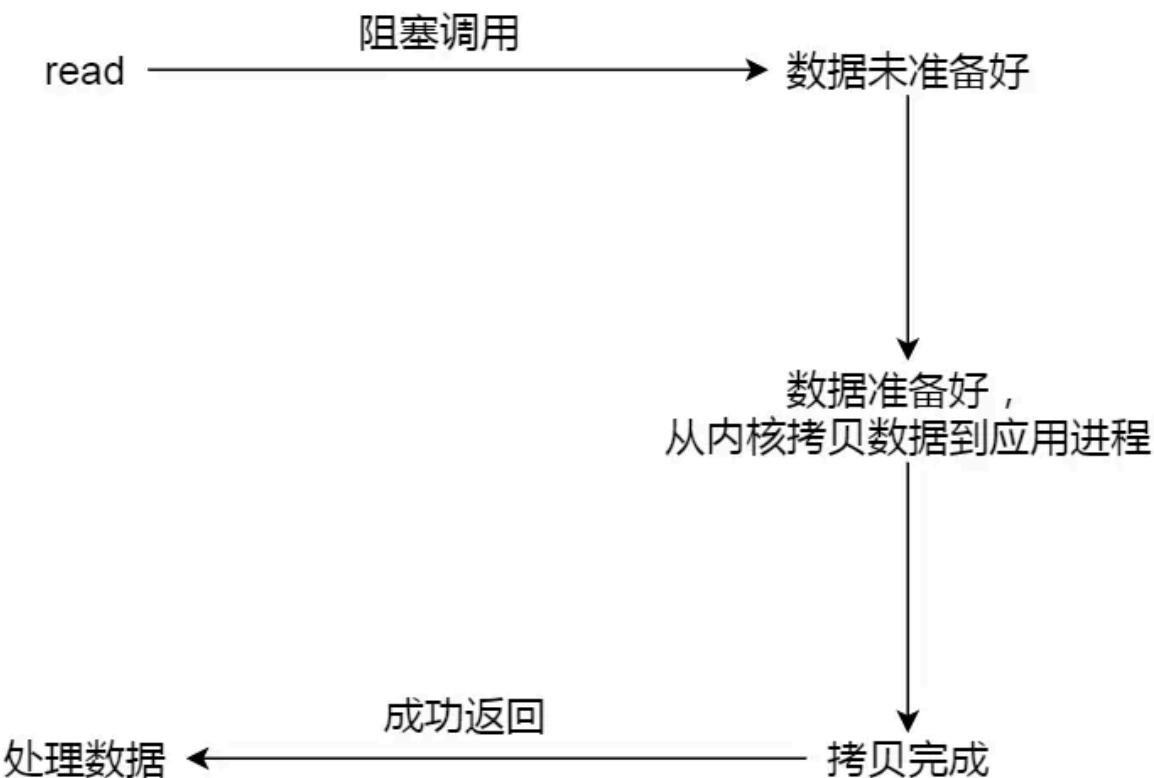
Proactor

最开始提到reactor模型是同步非阻塞I/O，而proactor是异步I/O。

阻塞I/O：进程在内核准备好数据（数据从磁盘到内核缓冲区）并将内核缓冲区中的数据拷贝至用户缓冲区完成前，进程一直是阻塞的。

应用进程

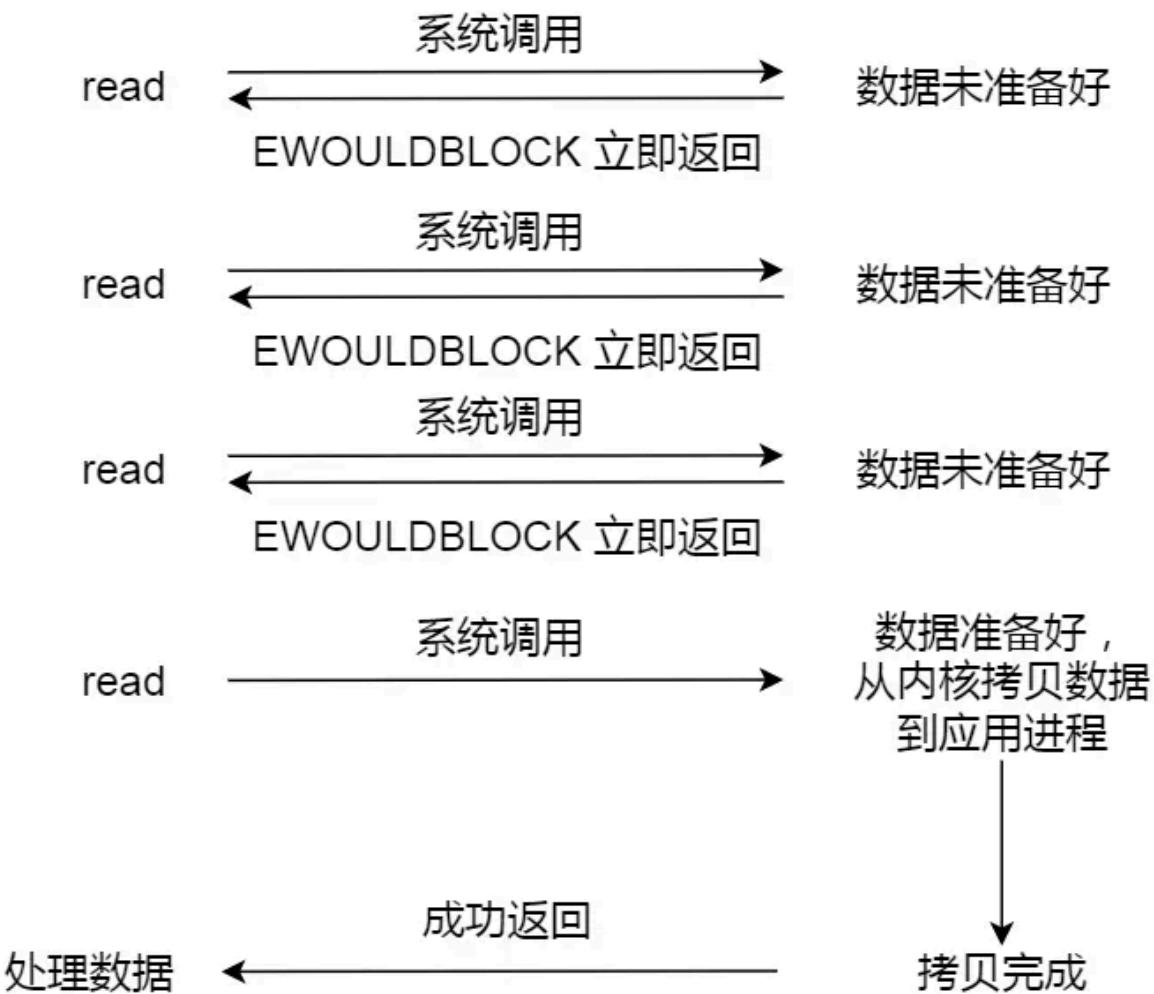
内核



非阻塞I/O：内核没准备好数据前（也就是会立刻返回正确或错误），进程可以去做其他事；过段时间再次发起调用，若内核已经准备好数据，需要等待内核将数据拷贝至用户缓冲区。

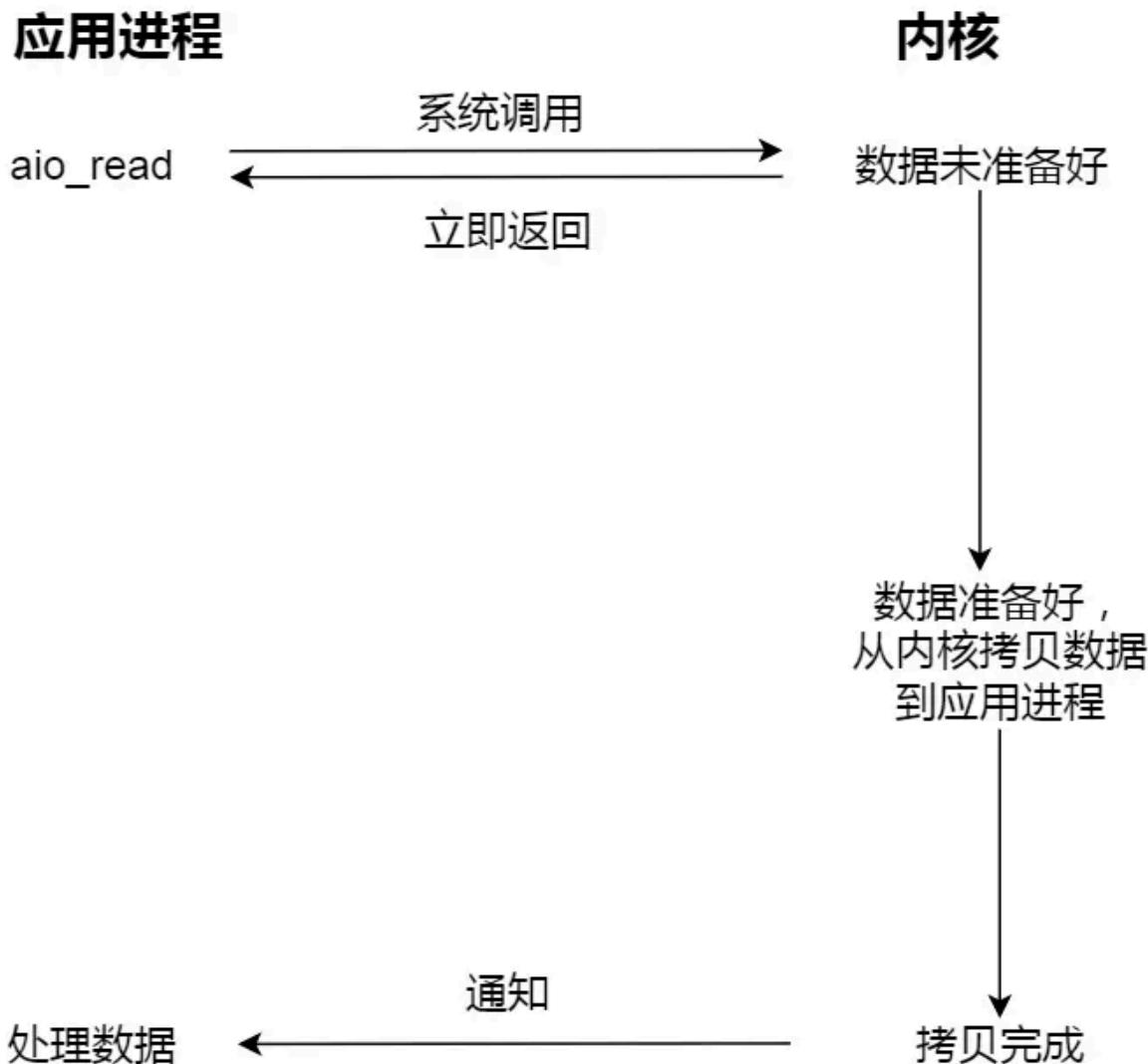
应用进程

内核



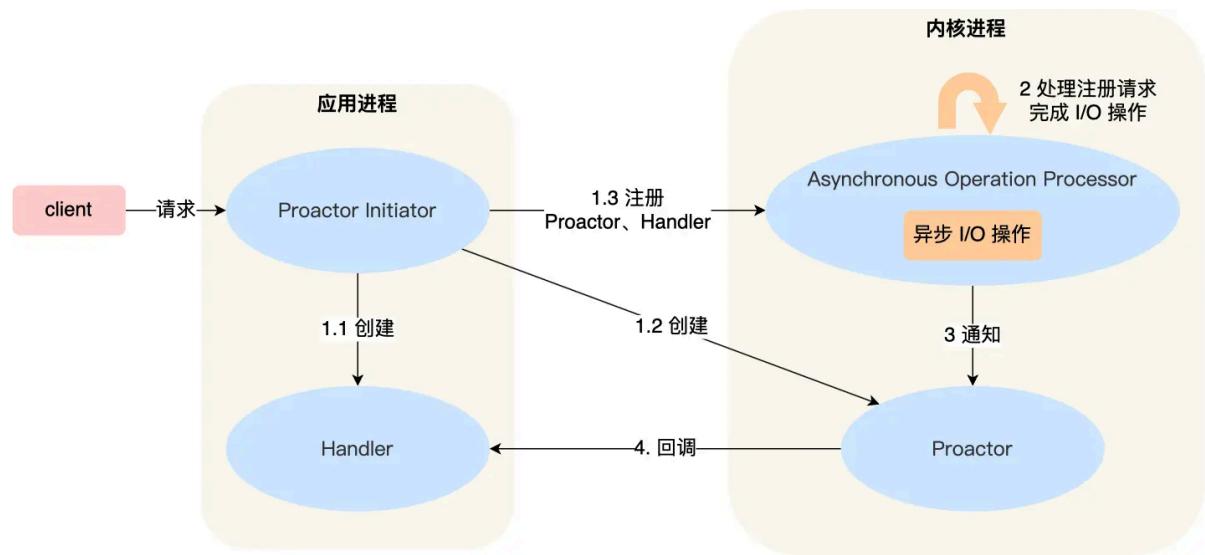
不管是阻塞I/O还是非阻塞I/O，这两者都是同步的，内核将数据从内核空间拷贝到用户空间的过程都是需要等待的，也就是说这个过程是同步的，如果内核实现的拷贝效率不高，read调用就会在这个同步过程中等待比较长的时间。

异步I/O：进程发起调用后，去做其他事；内核将数据拷贝至用户缓冲区后，通知进程进行处理（关键就在此，异步I/O是基于事件通知的）。



食堂打饭的例子来解释阻塞I/O，非阻塞I/O和异步I/O：阻塞 I/O 好比，你去饭堂吃饭，但是饭堂的菜还没做好，然后你就一直在那里等啊等，等了好长一段时间终于等到饭堂阿姨把菜端了出来（数据准备的过程），但是你还得继续等阿姨把菜（内核空间）打到你的饭盒里（用户空间），经历完这两个过程，你才可以离开。非阻塞 I/O 好比，你去了饭堂，问阿姨菜做好了没有，阿姨告诉你没，你就离开了，过几十分钟，你又来饭堂问阿姨，阿姨说做好了，于是阿姨帮你把菜打到你的饭盒里，这个过程你是得等待的。异步 I/O 好比，你让饭堂阿姨将菜做好并把菜打到饭盒里后，把饭盒送到你面前，整个过程你都不需要任何等待。

reactor是同步非阻塞I/O（因为使用I/O多路复用，I/O多路复用监听socket（文件描述符）是否有事件发生时是立即返回的），感知就绪I/O事件，也就是有事件发生时，os通知应用进程，进程进行处理；
proactor是异步I/O，感知已完成I/O事件，也就是有事件发生时，os处理完后再通知进程。



proactor模型流程可描述为：

1. proactor initiator创建handler和proactor
2. proactor initiator在asy...中进行handler和proactor注册
3. asy...完成I/O操作后通知proactor
4. proactor根据不同的事件类型调用不同的handler
5. handler完成事件处理

在Linux下的异步I/O是不完善的， aio系列函数是由POSIX定义的异步操作接口，不是真正的操作系统级别支持的，而是在用户空间模拟出来的异步，并且仅仅支持基于本地文件的aio异步操作，网络编程中的socket是不支持的，这也使得基于Linux的高性能网络程序都是使用Reactor方案。而Windows里实现了一套完整的支持socket的异步编程接口，这套接口就是IOCP，是由操作系统级别实现的异步I/O，真正意义上异步I/O，因此在Windows里实现高性能网络程序可以使用效率更高的Proactor方案。

由于磁盘读写特别慢，与内存速度相差可达10倍以上，有很多针对磁盘优化的方法用于提高系统吞吐量：

- 零拷贝
- 直接I/O（对应缓存I/O，也就是是否需要将磁盘中的数据拷贝至内存缓冲区的问题）
- 异步I/O
- Linux内核页高速缓存（PageCache）

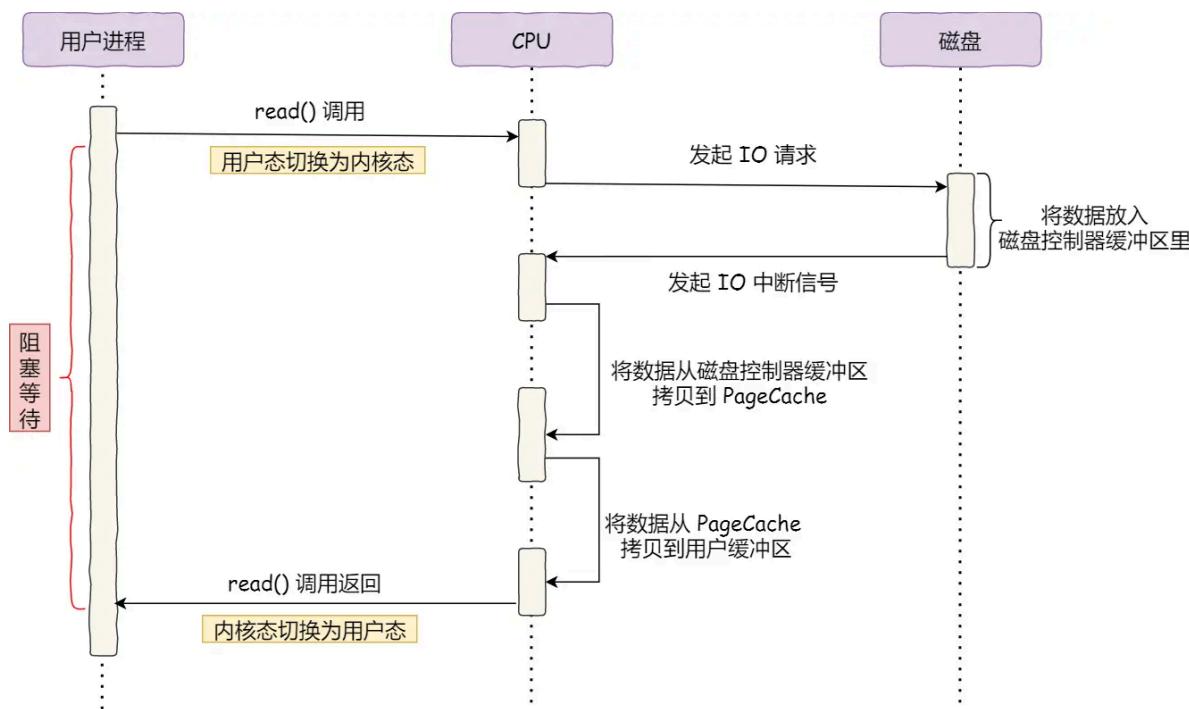
DMA

没有DMA之前的I/O（以读磁盘中的数据举例）：

- 进程发起read请求：
 - 进程通过调用read系统调用发起读取操作。这时，控制权从用户态转移到内核态（这种转换通常被称为“系统调用”或“陷入内核”）。
- 内核处理I/O请求：
 - 内核接收到读取请求后，解析请求并确定需要从磁盘读取的数据位置。
 - 内核向磁盘控制器发送I/O请求。这通常涉及向设备驱动程序发送命令，设备驱动程序再将这些命令转化为对具体硬件设备的操作。
- 磁盘控制器和数据传输：
 - 磁盘控制器处理来自内核的命令，从磁盘中读取数据到其内部缓冲区（通常称为磁盘缓冲区或硬件缓冲区）。
 - 一旦数据被读取到磁盘控制器的缓冲区，磁盘控制器会向CPU发出一个中断信号，表明数据已准备好被处理（准确来说应该是磁盘缓冲区慢了后发出中断信号）。
- OS处理中断：
 - 操作系统响应中断，保存当前进程的上下文（如CPU寄存器等状态），并处理中断。
 - 内核将数据从磁盘控制器的缓冲区复制到操作系统的页缓存（PageCache）中。
- 数据传输到用户空间：
 - 内核再将数据从页缓存复制到用户提供的缓冲区，也就是从内核空间到用户空间。
 - 一旦数据被复制到用户缓冲区，read系统调用完成，结果返回给用户程序。
- 返回用户态：
 - 系统调用完成后，操作系统将控制权返回给原始调用者，从内核态切换回用户态。

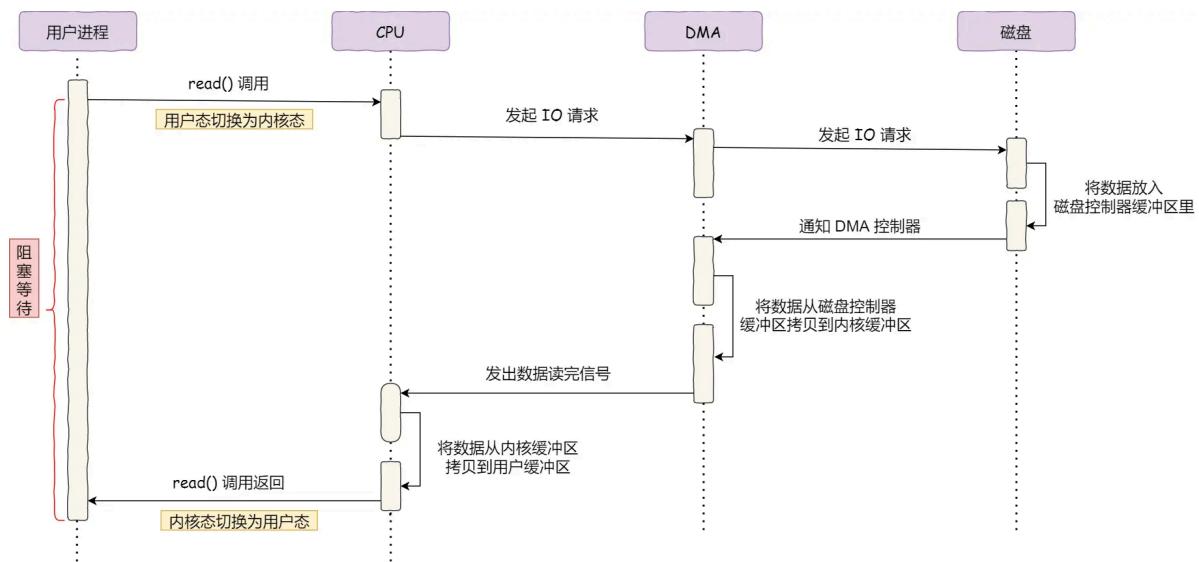
在Linux操作系统中，PageCache（页高速缓存）是内核用来缓存从硬盘读取的文件数据的一部分内存（局部性）。这样做的目的是为了提高文件访问的速度。当文件数据被读取时，它首先被存储在PageCache中。如果之后再次需要这些数据，操作系统可以直接从内存中获取，而不是重新从较慢的硬盘读取。这显著减少了访问延迟和提高了性能。如果pagecache中的空间不足了，会使用LRU淘汰最久未访问的缓存。另外，OS还可以将多个小的I/O请求合并成一个大的I/O请求。此外，PageCache还提供了预读功能。比如，假设read方法每次只会读32 KB的字节，虽然read刚开始只会读0 ~ 32 KB的字节，但内核会把其后面的32 ~ 64 KB也读取到PageCache，这样后面读取32 ~ 64 KB的成本就很低（比起读磁盘而言），如果在32 ~ 64 KB淘汰出PageCache前，进程读取到它了，收益就非常大。总体来说，虽然pagecache多了一次复制，但可以提高系统的安全性、稳定性和效率。

局部性原理基于这样一个观察：计算机程序倾向于重复使用近期使用过的数据项或者接近最近使用过的数据项的数据，也就是时间局部性（Temporal Locality）和空间局部性（Spatial Locality）。



上述过程中，涉及到数据的多次转移：磁盘->磁盘控制器缓冲区->pagecache->用户缓冲区，需要CPU对数据进行“搬运”，在传输大量数据（千兆网卡或从磁盘读大量数据）时，数据传输效率会很低并且CPU负荷会特别大。于是有了**DMA (Direct Memory Access)** 技术。

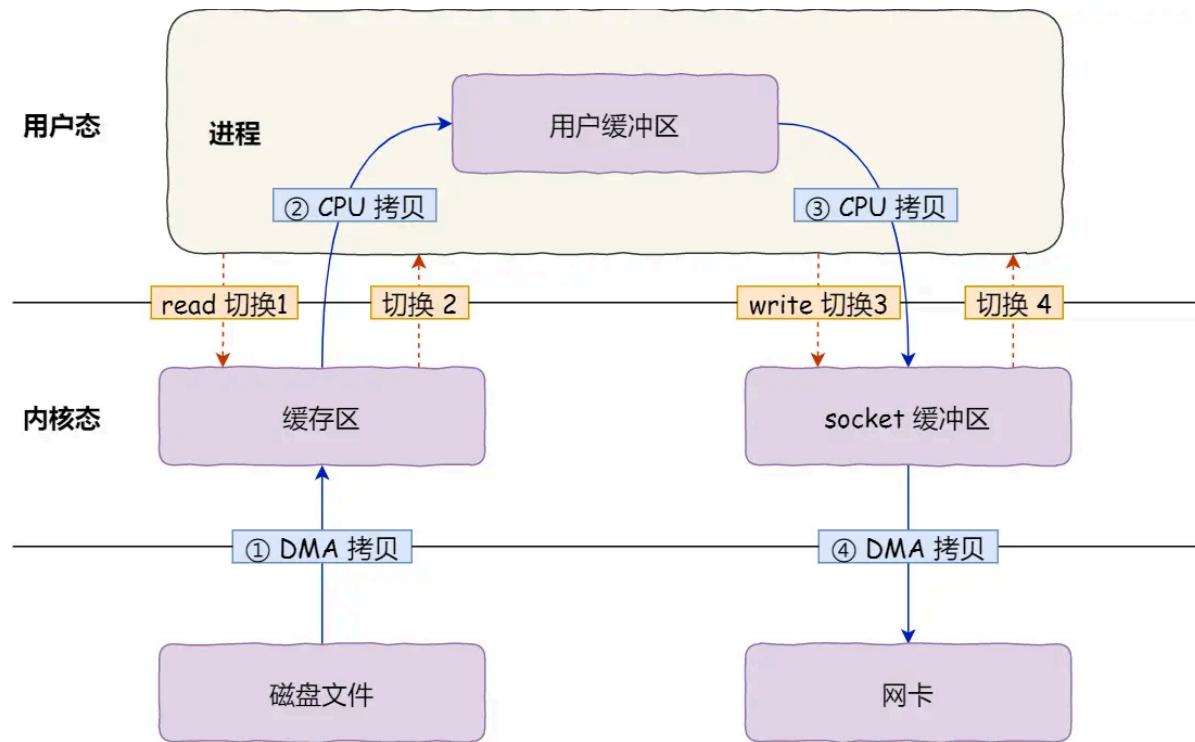
DMA技术出现后，上述I/O过程中由CPU搬运数据到pagecache的步骤由DMA控制器执行。当使用DMA时，CPU会初始化传输，之后它可以执行其他任务，直到数据传输完成时接收一个中断信号。这样，DMA可以在不占用CPU资源的情况下进行大量数据的快速传输。需要注意的是，**DMA只负责硬件系统数据到pagecache的传输；pagecache到进程缓冲区的数据传输仍然需要CPU**。早期DMA只存在于主板中，随着I/O设备的增多和需求不同，每个I/O设备都有自己的DMA。



传统的文件传输 (C/S架构)

当客户端向服务端请求数据/文件时，通常需要服务端将数据/文件从磁盘中读取出来，再通过网络协议发至客户端。上述过程通常需要用到下列系统调用：

```
read(file, tmp_buf, len);
write(socket, tmp_buf, len);
```



可以看到上述过程中发生了4次用户态/内核态的转换和4次数据拷贝，而我们只需要一次数据传输，这里面os状态切换和上下文保存以及数据拷贝都需要一定的时间和资源消耗，在高并发的场景中，性能和时延被累计和放大。因此，**提升系统性能需要减少用户态和内核态的上下文的切换和内存拷贝的次数。**

零拷贝

零拷贝是对于CPU而言，也就是没有从内存层面区拷贝数据，数据的拷贝由DMA直接完成。

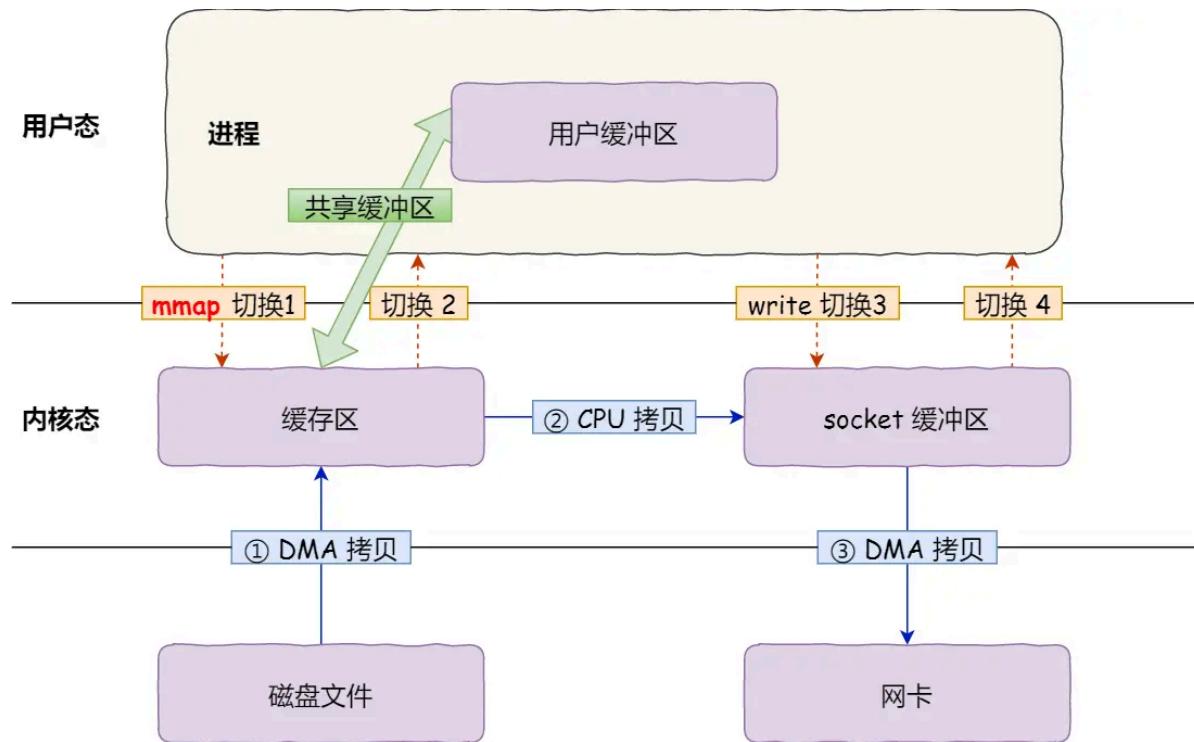
对于减少用户态和内核态的上下文切换关键在于减少系统调用的次数，一次系统调用必然会产生两次上下文切换。而单纯文件的传输过程中，无需对数据进行加工，用户的缓冲区就显得没有必要了。**注意零拷贝技术只适合对用户进程数据不用加工的场景。**

零拷贝技术通常有两种：

- mmap + write
- sendfile

mmap + write

通过使用mmap可以替换read，从而把“将磁盘中的数据/文件读取至用户缓冲区”转换为“将磁盘中的数据/文件读取至os的缓冲区即pagecache，再将这部分数据映射至用户空间，从而减少内核缓冲区到用户缓冲区的拷贝”。



通过mmap可以将内核缓冲区的数据映射至用户空间中，从而减少1次数据拷贝（记住不是两次噢，拷贝到socket缓冲区的步骤仍然是不能少的）。但是这样不就会产生安全问题了吗？内核又是如何避免的呢？

- 权限控制，内核会检查调用进程是否有足够的权限来访问对应的内存区域
- 页保护机制，通过页保护位 (rwx)，还可以使用cow来防止用户对共享内存的修改来影响其他进程和系统稳定性
- 地址空间布局随机化 (ASLR)
- ...

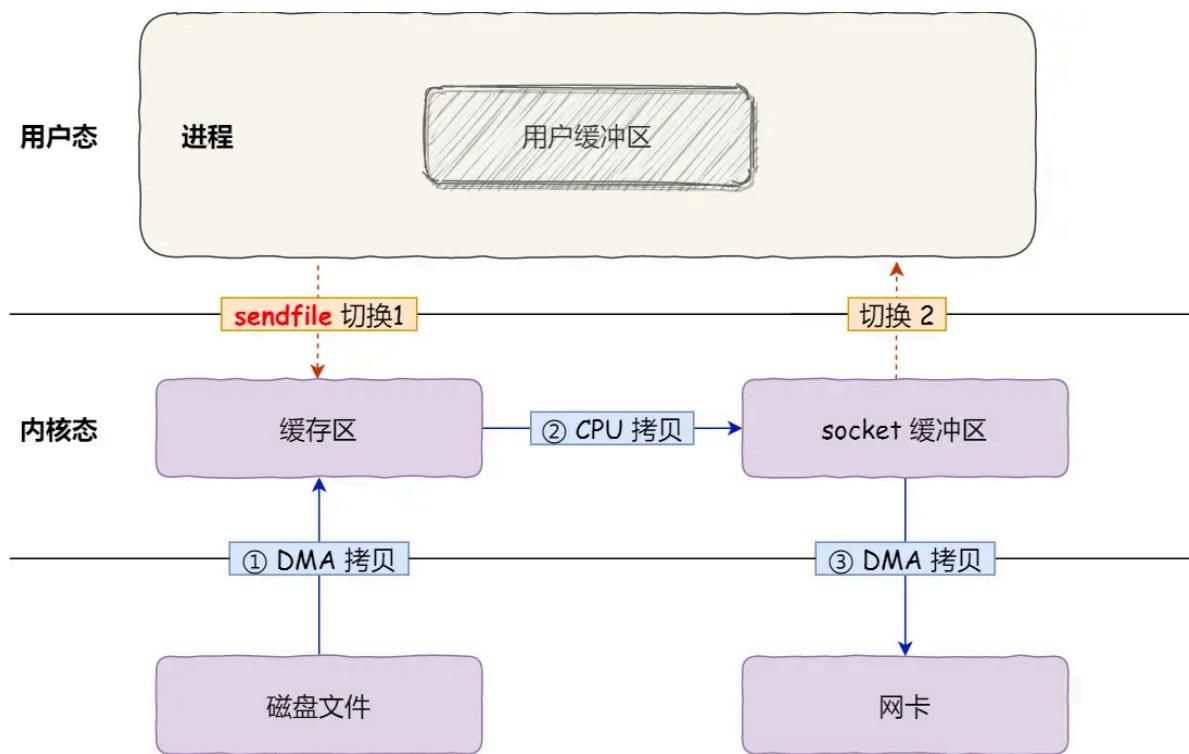
mmap+write还不是最理想的零拷贝，因为只减少了一次数据拷贝，系统调用仍然是2次。

sendfile

在Linux内核版本2.1中（现在稳定版已经到6.7.12了），提供了一个专门发送文件的系统调用函数 `sendfile()`，函数形式如下：

```
#include <sys/socket.h>
ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count);
```

它的前两个参数分别是目的端和源端的文件描述符，后面两个参数是源端的偏移量和复制数据的长度，返回值是实际复制数据的长度。通过sendfile，可以将系统调用减少为1次，减少两次内核态/用户态上下文切换。



但上述过程还不是真正的零拷贝技术，因为还有一次CPU的拷贝即将数据从内核缓冲区搬运到socket缓冲区。如果设备网卡支持SG-DMA (The Scatter-Gather Direct Memory Access) 技术（和普通的 DMA 有所不同），我们可以进一步减少通过CPU把内核缓冲区里的数据拷贝到socket缓冲区的过程。也就是只需1次系统调用，2次用户态/内核态上下文切换和2次数据拷贝（**注意这里还是需要先将磁盘中的数据拷贝至内存缓冲区**），极大的提高了数据/文件传输的性能。

```
# pyq @ 0x505951-Y7000P in ~ [15:07:42]
$ ethtool -k eth0 | grep scatter-gather
scatter-gather: on
    tx-scatter-gather: on
    tx-scatter-gather-fraglist: off [fixed]

# pyq @ 0x505951-Y7000P in ~ [15:07:43]
$ uname -a
Linux 0x505951-Y7000P 5.15.146.1-microsoft-standard-wsl2 #1 SMP Thu Jan 11
04:09:03 UTC 2024 x86_64 x86_64 x86_64 GNU/Linux
```

使用零拷贝技术的项目

Kafka：是一个分布式流处理平台，主要用于构建实时的数据流应用和高性能的数据管道。可以处理来自多个来源的大量数据，并能够以高吞吐量和低延迟的方式向多个消费者提供数据。Kafka在许多企业中广泛使用，包括在金融服务、电信、零售、物联网(IoT)、物流和许多其他领域。

Nginx

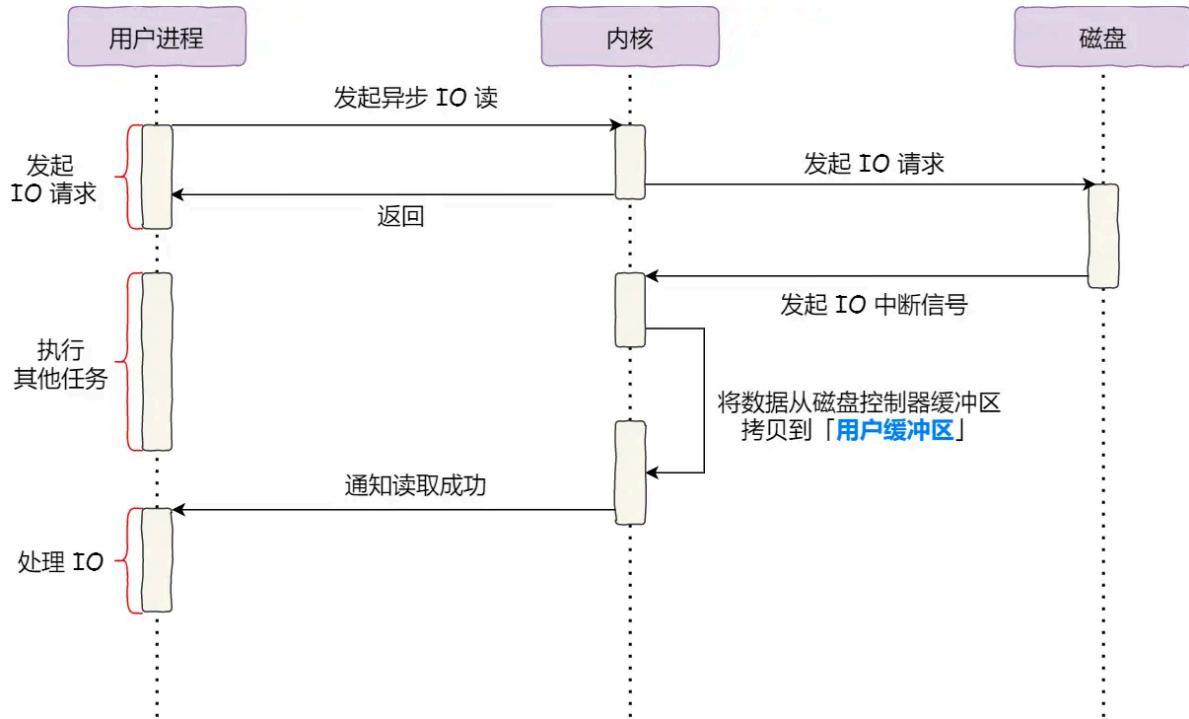
大文件的传输

虽然通过pagecache的缓存和预读功能，可以提升系统性能。但系统传输大文件/大数据时，pagecache的优势将会被削弱或消失，因为pagecache的容量是很小的，内容将会被迅速替代。另外，由于文件太大，可能某些部分的文件数据被再次访问的概率比较低，这样就会带来2个问题：

- pageCache由于长时间被大文件占据，其他「热点」的小文件可能就无法充分使用到pageCache，于是这样磁盘读写的性能就会下降了

- pageCache中的大文件数据，由于没有享受到缓存带来的好处，但却耗费DMA多拷贝到pageCache一次

既然这样，针对大文件的传输，我们需要避免将数据拷贝至pagecache，直接拷贝至用户缓冲区。此外，用户进程在发起I/O调用后就会进入阻塞状态，直到内核返回，这个问题可以通过异步I/O来解决。



异步I/O并没有涉及到PageCache，所以使用异步I/O就意味着要绕开PageCache。**绕开PageCache的I/O叫直接I/O，使用PageCache的I/O则叫缓存I/O。通常，对于磁盘，异步I/O只支持直接I/O。**在高并发的场景下，针对大文件的传输的方式，应该使用「异步 I/O + 直接 I/O」来替代零拷贝技术。

直接I/O的场景：

1. 大文件的传输
2. 应用程序已经实现了数据的缓存，不需要OS再通过pagecache来再次缓存了

而对于小文件，则更适合pagecache并使用零拷贝。在nginx中可以根据不同文件的大小设置传输方式。

文件系统的基本组成

文件系统负责计算机数据的持久化保存，文件系统管理的基本单位是文件，不同的文件系统对文件的管理组织方式不同。**文件系统中的每个文件包含两个数据结构，用于记录文件的元数据和目录层次结构：**

- **索引节点inode：** inode用于记录文件的元数据信息，包含文件的inode编号、大小、访问权限、创建时间、修改时间和存储在磁盘的位置等。inode和文件一一对应，是inode的唯一标识，inode存储在磁盘中（为了加快速度，内存中也有inode缓存）
- **目录项dentry：** dentry包含文件的名字（是的，inode并不包含文件的名字），inode指针和其他目录项的关系，多个目录项关联起来组成了目录结构。与inode不同，目录项由内核维护，缓存在内存中

为什么不将文件名字存储在inode中呢？

1. 分离文件名和文件元数据，使得硬链接的实现简单，可以通过多个目录项指向同一个inode，使一个inode（文件或目录）拥有多个名称
2. 节省空间和提高效率，如果将文件名存储在inode中，如果文件名变更则需要更大的inode或者重新分配inode（文件名字过长，inode大小一般固定），而目录项中的文件名字字段长度不固定，这样修改文件名字只需对目录项中的文件名字字段进行修改，无需影响inode
3. 优化性能，如果文件名存储在inode中，那根据文件名去查找文件时，需要遍历inode表。而将其存储在目录项中，则直接去其目录文件（存储文件名和inode的映射），访问对应的inode即可

由于inode唯一标识一个文件，而目录项包含文件名字。因此**inode和目录项是一对多的关系，一个文件可以有多个别名，硬链接就是为文件创建一个额外的目录项，目录项中文件名不同，但都指向相同的inode，实现可以使用不同文件名访问同一个文件。**

目录也是文件，文件存储的是文件数据，而目录存储的是子目录和文件的信息（这些信息通常以目录项的形式存在）。注意目录项不等于目录，目录是存储在磁盘中的一种文件，而目录项是缓存在内存中的，为了提高性能，操作系统会将读过的文件目录的目录项缓存在内存中。目录项既可以表示目录，也可以表示文件。

目录项和dentry的疑惑

- 目录项：这是磁盘上的数据结构，存储在目录文件中。每个目录项包含至少两个基本信息：文件名和对应的inode编号。这些目录项是文件系统的一部分，它们持久化存储在磁盘上，确保了文件系统在关机后仍能保持其结构和数据。
- dentry：这是操作系统内核中的一个缓存数据结构，存在于内存中。dentry用于存储文件系统中的目录项信息，以加速文件系统的查找和访问过程。dentry缓存文件名到inode的映射，以及目录结构之间的关系，从而使得文件名解析更快。

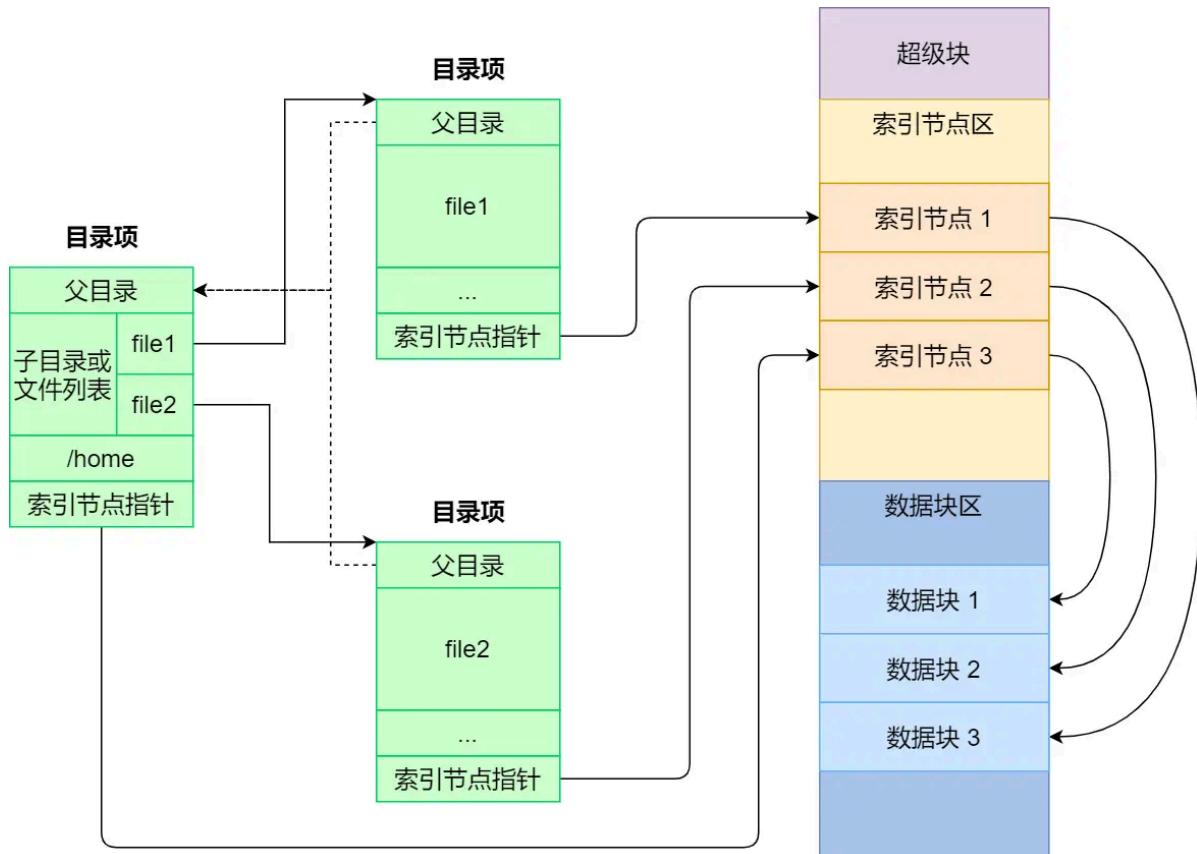
为什么说dentry只在内存中：

- 性能优化：dentry设计为内存中的结构是为了性能优化。通过将常用的文件名和inode映射存储在快速的RAM中，系统可以迅速响应文件访问请求，减少磁盘I/O操作，从而显著提高效率。
- 非持久化：dentry不需要持久化存储，因为它们仅为当前系统操作提供服务。系统关机或重启时，dentry缓存可以被重建，因为所有必要的信息（文件名和inode映射）已经安全地存储在磁盘上的目录项中。

结构体和存储方式

- 磁盘结构体：目录项在磁盘上可能没有一个固定的“结构体”形式，它们的具体存储格式依赖于特定的文件系统实现（如ext4, NTFS等）。但通常，这些信息是按照一定格式序列化存储的。
- 内存结构体：相比之下，dentry在内存中有明确的结构体定义，这些结构体由操作系统内核维护，用于快速访问和管理文件系统的目录树。

磁盘读写的最小物理单位是扇区，扇区最早为512B，现代的可能为4KB。扇区很小，如果操作系统以扇区作为读写单位效率会很低，于是出现了块block（Linux中通常为4KB），块是文件系统的逻辑结构，一个block可以包含多个扇区。



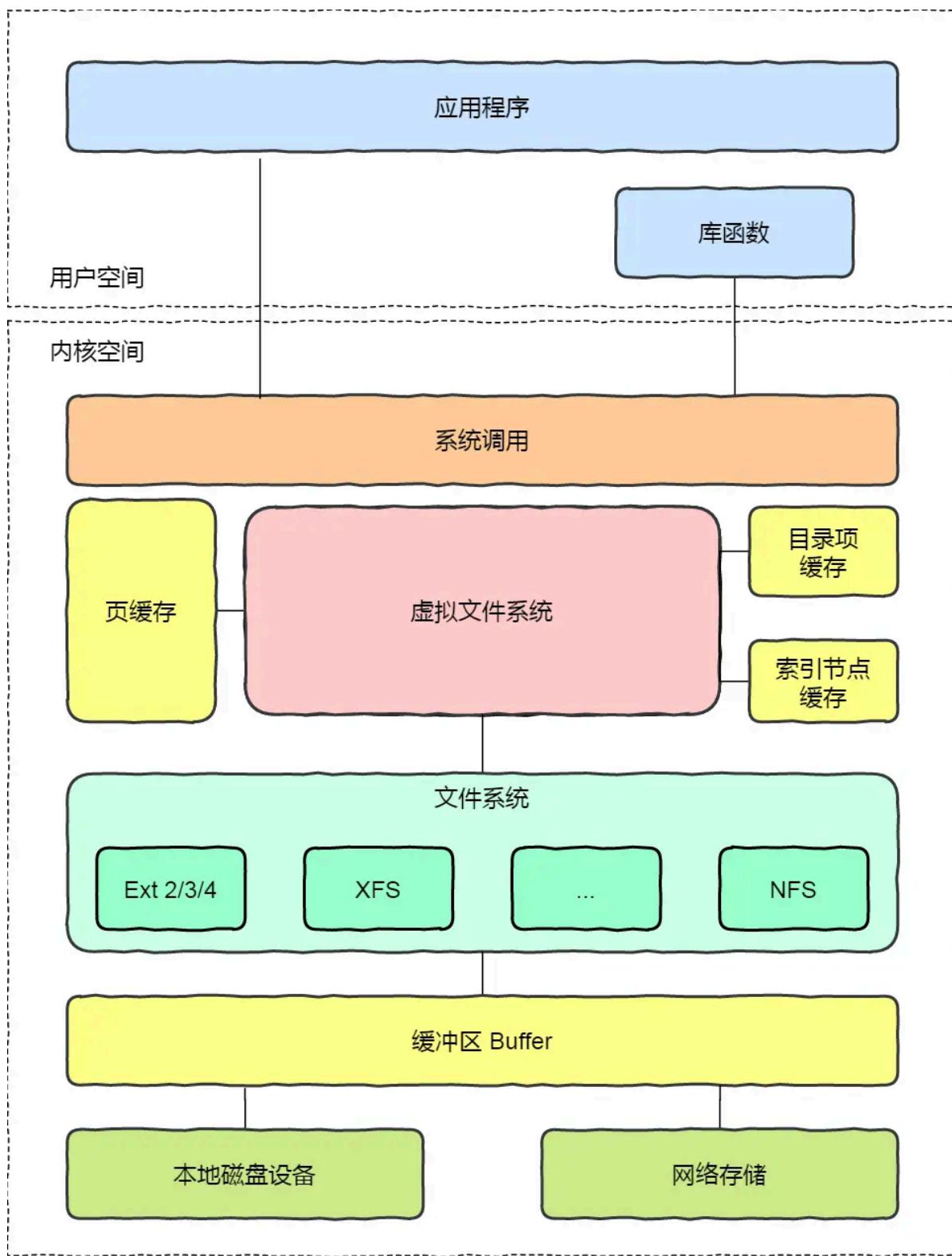
磁盘在格式化时，通常分为3个区域：

- 超级块：存储文件系统的信息，比如块个数、块大小和空闲空间等
- 索引节点区：存储索引节点
- 数据块区：存储具体的文件或目录数据

超级快在文件系统挂载进内存时加载进内存，索引节点区当文件被访问时加载进内存。

虚拟文件系统

文件系统种类有很多，为了屏蔽不同文件系统的差异，操作系统在用户空间和文件系统之间也就是内核中引入了虚拟文件系统（VFS）。VFS屏蔽了底层文件系统工作的细节原理，提供了操作管理文件统一的数据结构和接口。



Linux根据存储的位置不同，将文件系统分为3类：

1. 磁盘文件系统：数据直接存储在磁盘中，比如ext2/3/4, XFS
2. 内存的文件系统，不存储在磁盘（不需要持久化），占用内存空间，比如/proc和/sys，反应系统的实时数据，动态生成
3. 网络文件系统：访问其他计算机的数据，比如NFS和SMB

文件系统必须挂载到某个目录才能使用，比如Linux会把文件系统挂载到根目录。

文件的使用

在编写代码时，写入一个文件的基本过程如下：

```
fd = open(name, flag); # 打开文件  
...  
write(fd,...);          # 写数据  
...  
close(fd);              # 关闭文件
```

首先通过open()系统调用打开给定文件名的文件，并返回该文件的文件描述符；然后再使用write向该文件写入数据，参数为返回的文件描述符；最后通过close关闭文件描述符，避免资源泄露。操作系统会在进程的PCB中为其维护一个打开文件描述符表，用于跟踪进程打开的文件。操作系统在打开文件描述符表中维护着进程打开文件的状态和信息：

- 文件指针：系统跟踪上次读写位置作为当前文件位置指针，这种指针对打开文件的某个进程来说是唯一的；
- 文件打开计数器：文件关闭时，操作系统必须重用其打开文件表条目，否则表内空间不够用。因为多个进程可能打开同一个文件，所以系统在删除打开文件条目之前，必须等待最后一个进程关闭文件，该计数器跟踪打开和关闭的数量，当该计数为 0 时，系统关闭文件，删除该条目；
- 文件磁盘位置：绝大多数文件操作都要求系统修改文件数据，该信息保存在内存中，以免每个操作都从磁盘中读取；
- 访问权限：每个进程打开文件都需要有一个访问模式（创建、只读、读写、添加等），该信息保存在进程的打开文件表中，以便操作系统能允许或拒绝之后的 I/O 请求；

读文件和写文件的过程：

- 当用户进程从文件读取 1 个字节大小的数据时，文件系统则需要获取字节所在的数据块，再返回数据块对应的用户进程所需的数据部分。
- 当用户进程把 1 个字节大小的数据写进文件时，文件系统则找到需要写入数据的数据块的位置，然后修改数据块中对应的部分，最后再把数据块写回磁盘。

所以说，**文件系统的基本操作单位是数据块**。

文件的存储

文件数据的存储是在磁盘中，与数据在内存中的存储方式类似，文件数据在磁盘中的存储有以下2种方式：

- 连续地址空间存放
- 非连续地址空间存放：可分为链表方式和索引方式

连续地址空间存放

连续地址空间存放顾名思义是将文件数据存放在磁盘的连续块中。**这种方式比较高效，一次磁盘寻道即可读出文件全部数据，但需要提前知道文件的大小**，也就是需要在文件头（类似inode的东西，PS感觉这里说文件头名字不准确）中指明文件的起始块和大小。

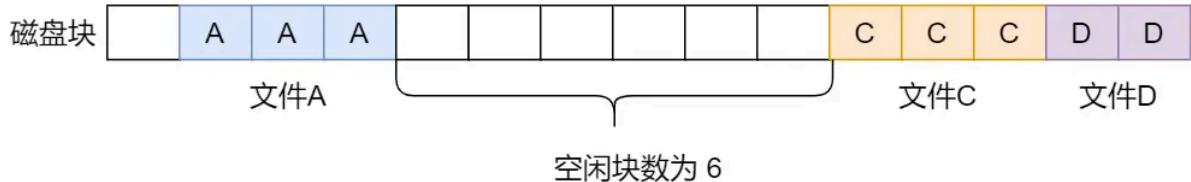
连续地址空间存放的缺点则是容易产生磁盘空间碎片和文件长度扩展难。容易产生磁盘空间碎片则是由于不同文件数据存放都是连续并且大小不一致的，比如我文件A、文件B和文件C是相邻的，大小分别是300MB、200MB和300MB。文件B删除后，文件A和文件C之间空出200MB容量，若新文件大于300MB，则可能造成这部分磁盘空间浪费，造成空间碎片。文件长度扩展难就更好理解了，因为是连续空间存放，涉及到文件数据的移动。



假设文件 B 被删除了，此时最大空闲块数是 6

如果新来的文件所需的块数 ≤ 6 ，这时新的文件就可以被正常存放

如果新来的文件所需的块数 > 6 ，这时新的文件就无法存放到磁盘



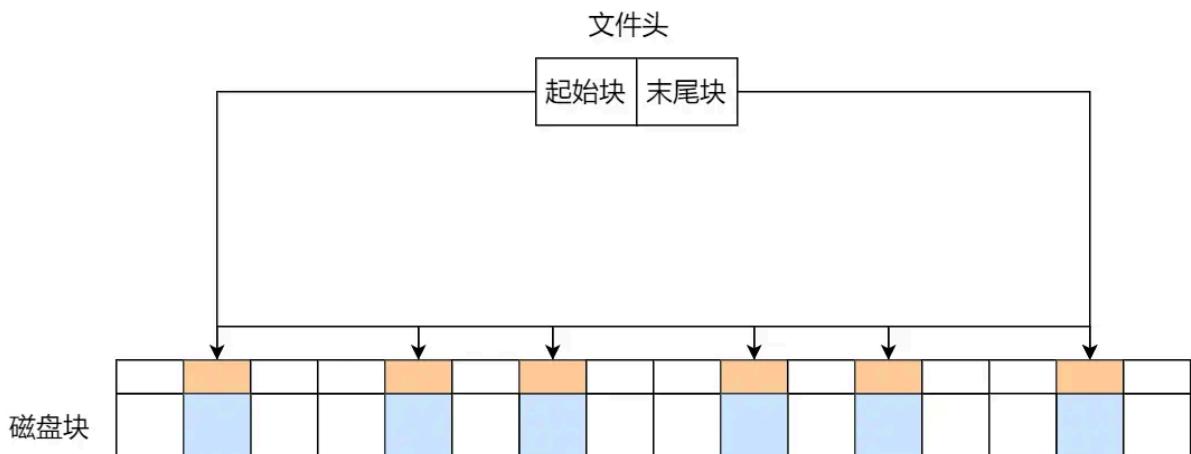
磁盘碎片的缺陷

非连续地址空间存放

链表

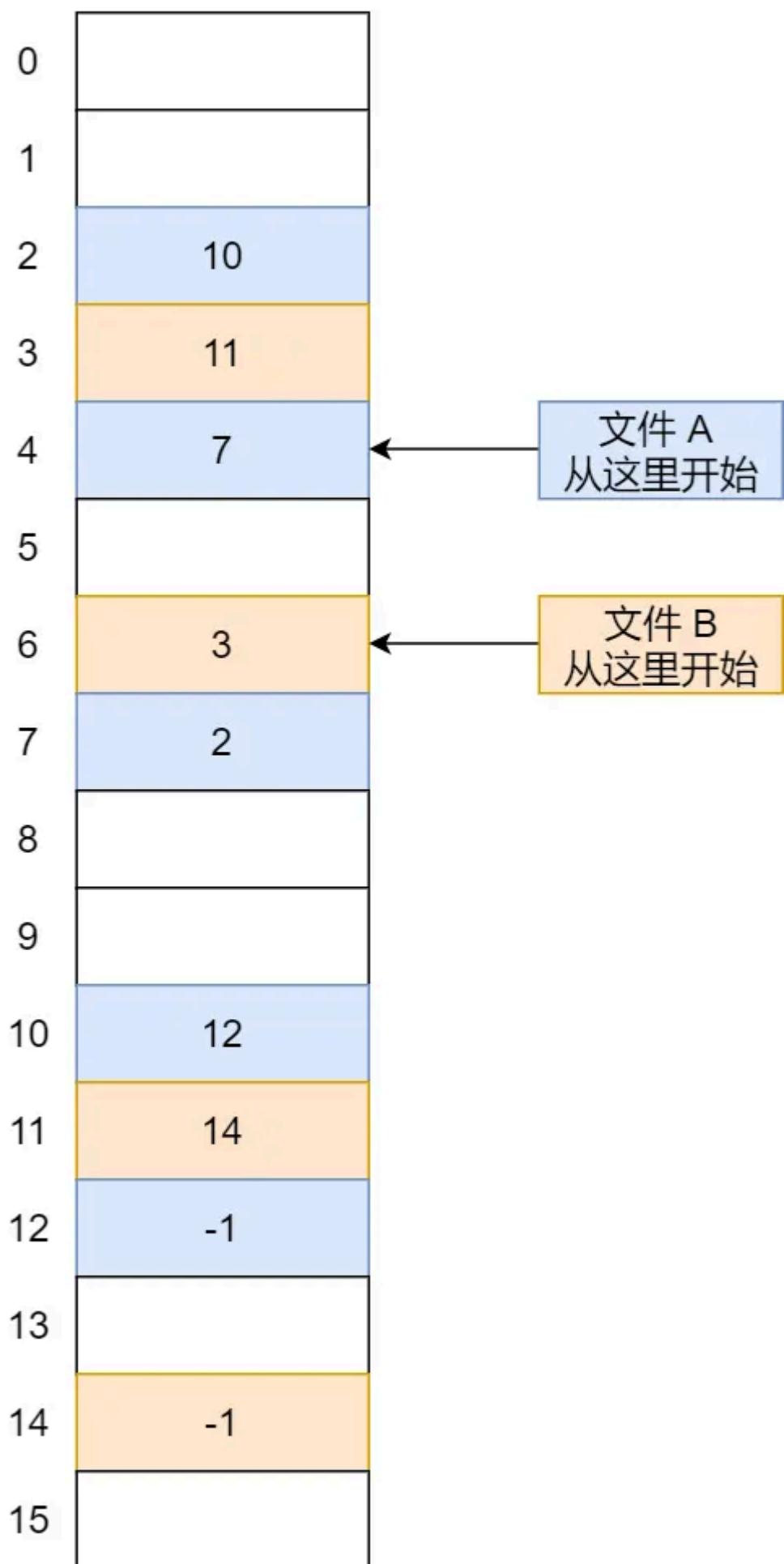
链表方式的文件数据存储可以消除磁盘空间碎片和文件长度扩展难的问题，链表方式可分隐式链表和显式链表。

隐式链表方式，文件头需要记录文件开始块和结束块的位置，每个文件块中必须留出一个位置存储下一个文件块的指针，这样从链表头开始可以顺着指针访问文件数据。隐式链接的缺点具有链表的缺点，只能通过指针顺序访问数据块，不能直接访问。此外，隐式链接的稳定性差，可能会导致文件块中的指针数据丢失。



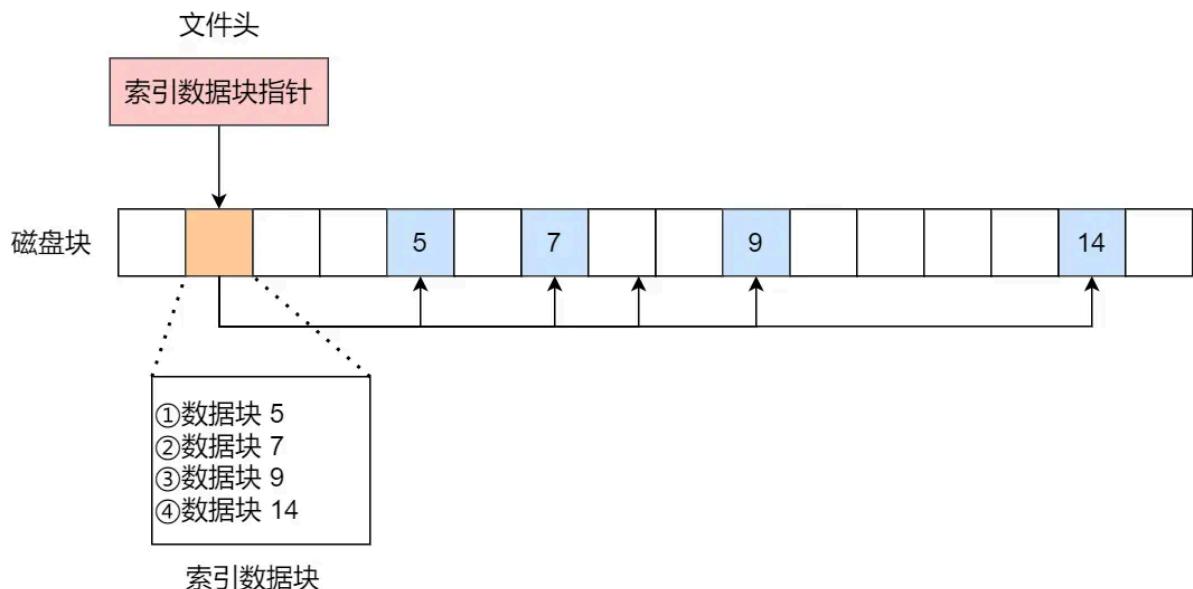
而显式链接则是将文件块中的指针显式的存储在内存中的一个表中即FAT（文件分配表），文件分配表每一个表项表示一个磁盘块，表项中的内容为文件下一个磁盘块的位置或者特殊的结束标识（例如 0xFFFFFFFF）。通过查找文件目录项中的起始号并根据FAT表可以访问文件数据。FAT支持直接访问，这里的直接访问并不是数组那样，我们依然需要通过链表指针顺序查找，但是由于FAT表存储在内存中，我们在顺序查找到所需文件数据所在块时，再读取对应的磁盘即可，而隐式链接则需要依次顺序读盘。由于显式链接将块指针存储在内存中，FAT方案不适合大磁盘，大磁盘使得FAT表会占用大量的内存。

磁盘块



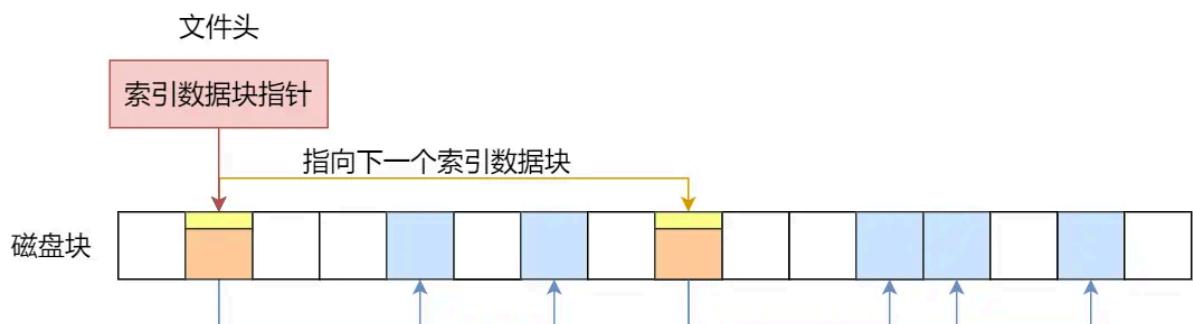
索引

索引方式通过为每个文件创建索引数据块（一个block），索引数据块中存储指向文件存储数据块的指针。创建文件时，需要在其文件头中包含索引数据块的指针，在将文件数据写入磁盘块时，现在磁盘中对其进行分配，再在索引数据块中添加该块指针。

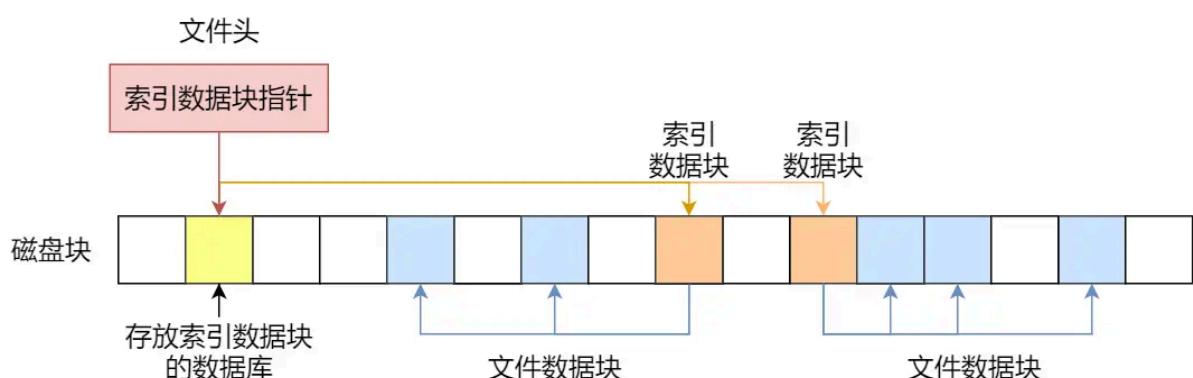


索引方式存储具有文件长度可以动态变化，不会产生磁盘空间碎片，支持顺序读写和随机读写的有点，缺点之一是存储索引带来的开销，每一个文件都需要为其分配一个索引块。文件数据很小，仍然需要为其分配一个索引块造成空间浪费；如果文件数据很大，一个索引块存储不下，则需要多个索引块，可以通过链表+索引或者索引+索引的方式解决。

链表+索引：在索引块中留位置存放指向下一个索引块的指针。



索引+索引：在一个索引块中存放多个索引块指针。



unix文件的存储方式

随机地址空间存储，链表地址空间存储和索引地址空间存储三种方式的优缺点：

方式	访问磁盘次数	优点	缺点
顺序分配	需访问磁盘 1 次	顺序存取速度快，当文件是定长时可以根据文件起始地址及记录长度进行随机访问	要求连续的存储空间，会产生外部碎片，不利于文件的动态扩充
链表分配	需访问磁盘 n 次	无外部碎片，提高了外存空间的利用率，动态增长较方便	只能按照文件的指针链顺序访问，查找效率低，指针信息存放消耗内存或磁盘空间
索引分配	m 级需访问磁盘 m+1 次	可以随机访问，易于文件的增删	索引表增加存储空间的开销，索引表的查找策略对文件系统效率影响较大

早期unix文件系统结合了上述3种方式的优缺点，根据不同的文件大小使用不同的存储方式：

- 如果存放文件所需的数据块小于10块，则采用直接查找的方式；
- 如果存放文件所需的数据块超过10块，则采用一级间接索引方式；
- 如果前面两种方式都不够存放大文件，则采用二级间接索引方式；
- 如果二级间接索引也不够存放大文件，这采用三级间接索引方式；

上述方式则需要文件头使用13个指针：

- 前面10个指针指向直接查找的10个数据块
- 第11个指针指向一级索引块
- 第12个指针指向二级索引块
- 第13个指针指向三级索引块

上述方案用在Linux Ext2/3文件系统中，解决了大文件的存储，但对于大文件查找效率较低，Ext4对其进行了优化。

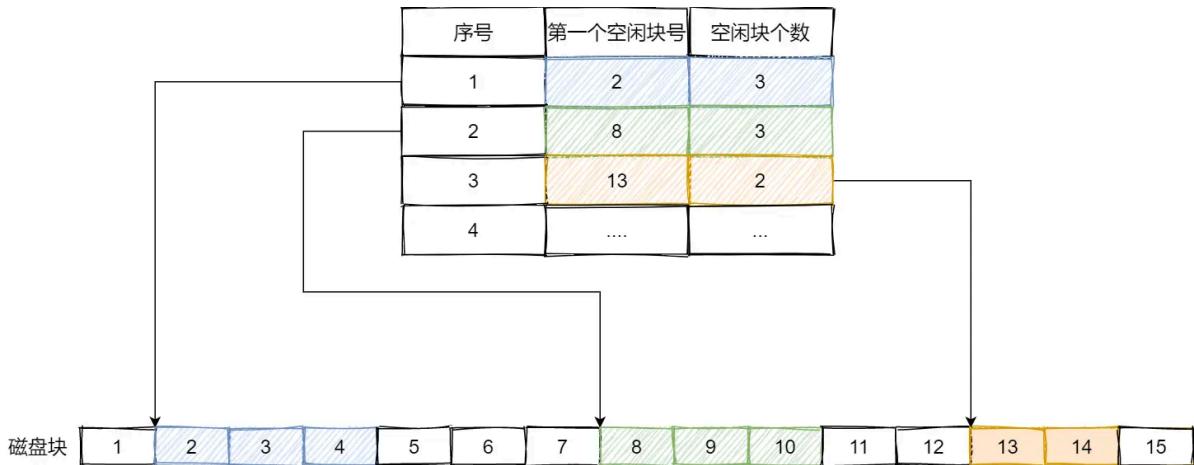
空闲空间管理

前面所述的文件存储空间方式是针对已存储文件的，那对于未存储的文件应该怎么为其分配空闲的磁盘空间呢，包括3种方法：

1. 空闲表法
2. 空闲链表法
3. 位图法

空闲表法

空闲表法则是为磁盘中的所有空闲空间维护一个空闲表，每个表项为空闲块的起始块号和大小，这种方式适合连续地址空间存储，并且如果磁盘中有很多小的空闲空间（空闲表的表项存放的是连续空间），空闲表将会变得很大。搜索和删除空闲空间都需要遍历空闲表。



空闲链表法

相较于空闲表法，空闲链表法使用指针保存下一个空闲块的位置，需要多大的空间就遍历链表取相应的数据块即可，添加则是将其添加到链表头，适合离散化数据存储，只需保留一个指向第一个空闲块的位置即可。空闲链表法简单但不能随机访问，此外每个块存储指针需要消耗一定的空间。

位图法

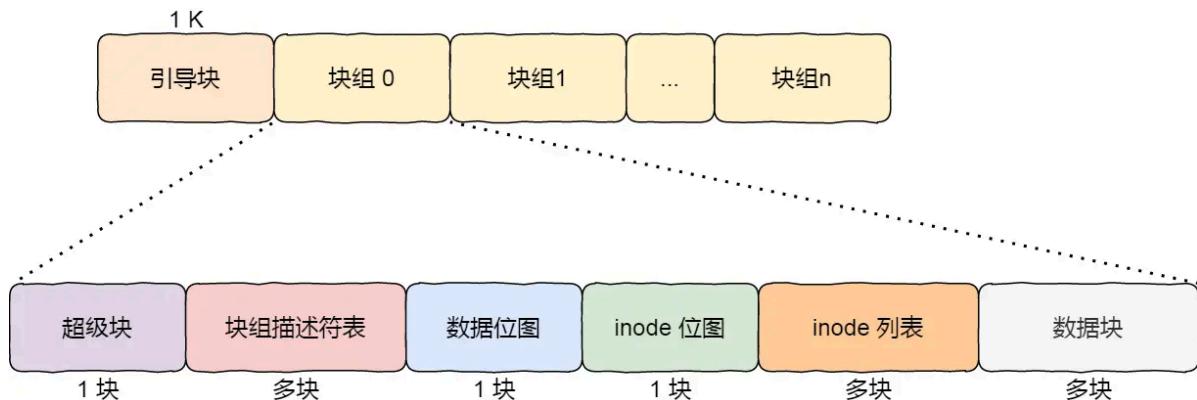
空闲表法和空闲链表法都不适合大型文件系统，会使空闲表和空闲链表的空间变的很大。

而位图法通过二进制的一位来关联对应的数据块，为0表示数据块空闲，为1表示数据块占用，每个数据块都对应一个二进制位。除了空闲块的管理，Linux对于空闲inode的管理也使用位图。

文件系统的结构

Linux通过位图来管理空闲块和空闲inode，如果只使用一个块来存储位图，只能管理很少的空间，因此Linux中将多个块联合在一起组成块组进行管理。

Linux Ext2文件系统结构如下：



最前面的第一个块是引导块，在系统启动时用于启用引导，接着后面就是一个一个连续的块组了，块组的内容如下：

- 超级块，包含的是文件系统的重要信息，比如inode总个数、块总个数、每个块组的inode个数、每个块组的块个数等等。
- 块组描述符，包含文件系统中各个块组的状态，比如块组中空闲块和inode的数目等，**每个块组都包含了文件系统中「所有块组的组描述符信息」**。
- 数据位图和inode位图，用于表示块组中对应的数据块或 inode 是空闲的，还是被使用中。
- inode列表，包含了块组中所有的inode，inode用于保存文件系统中与各个文件和目录相关的所有元数据。

- 数据块，包含文件的有用数据。

每个块组里有很多重复的信息，比如超级块和块组描述符表，这两个都是全局信息。如果系统崩溃破坏了超级块或块组描述符，有关文件系统结构和内容的所有信息都会丢失。如果有冗余的副本，该信息是可能恢复的。通过使文件和管理数据尽可能接近，减少了磁头寻道和旋转，这可以提高文件系统的性能。不过，Ext2的后续版本采用了稀疏技术。该做法是，超级块和块组描述符表不再存储到文件系统的每个块组中，而是只写入到块组0、块组1和其他ID可以表示为3、5、7的幂的块组中。

感觉快组描述符的同步是个问题

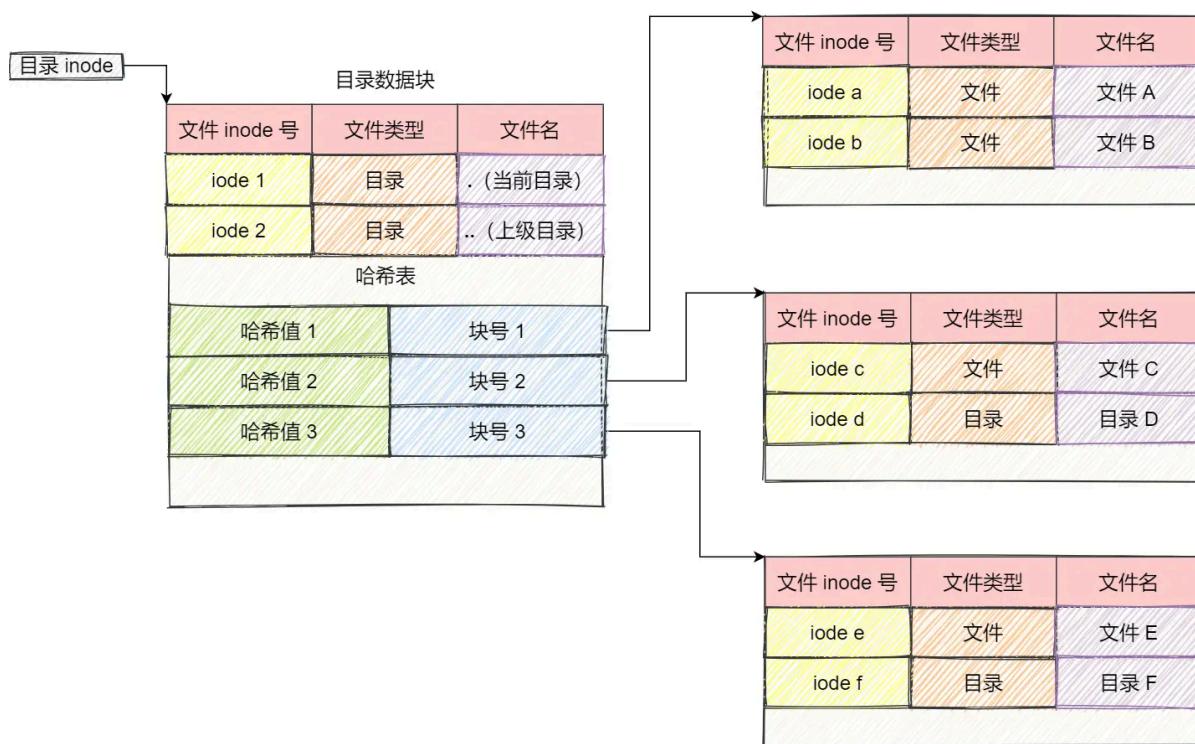
目录的存储

Linux中目录也是文件，普通文件存储的是文件的数据，而目录文件存储的则是目录下文件（普通文件和目录文件）的信息。目录可以通过vim打开，使用vim打开一个文件目录的内容如下：

```
vim
" Netrw Directory Listing
" /home/pyg
" Sorted by name
" Sort sequence: [ \/ $ . \<core>%(\.\d\+\%)= \> . \h\$ . \c\$ . \cpp\$ . \~=\$.* . \o\$ . \obj\$ . \info\$ . \swp\$ . \bak\$ . \~\$

" Quick Help: <F1>:help -go up dir D:delete R:rename s:sort-by x:special
" =====

../
./
..cache/
..config/
..oh-my-zsh/
..tmux/
..vim/
..vscode-server/
downloads/
go/
projects/
.bash_history
.bash_logout
.bashrc
.gitconfig
.motd_shown
.profile
.rediscli_history
.sudo_as_admin_successful
.tmux.conf.local
.tmux.conf@ --> .tmux/.tmux.conf
.viminfo
.zcompdump
.zsh_history
.zshrc
~
~
~
~
~
~
~
```

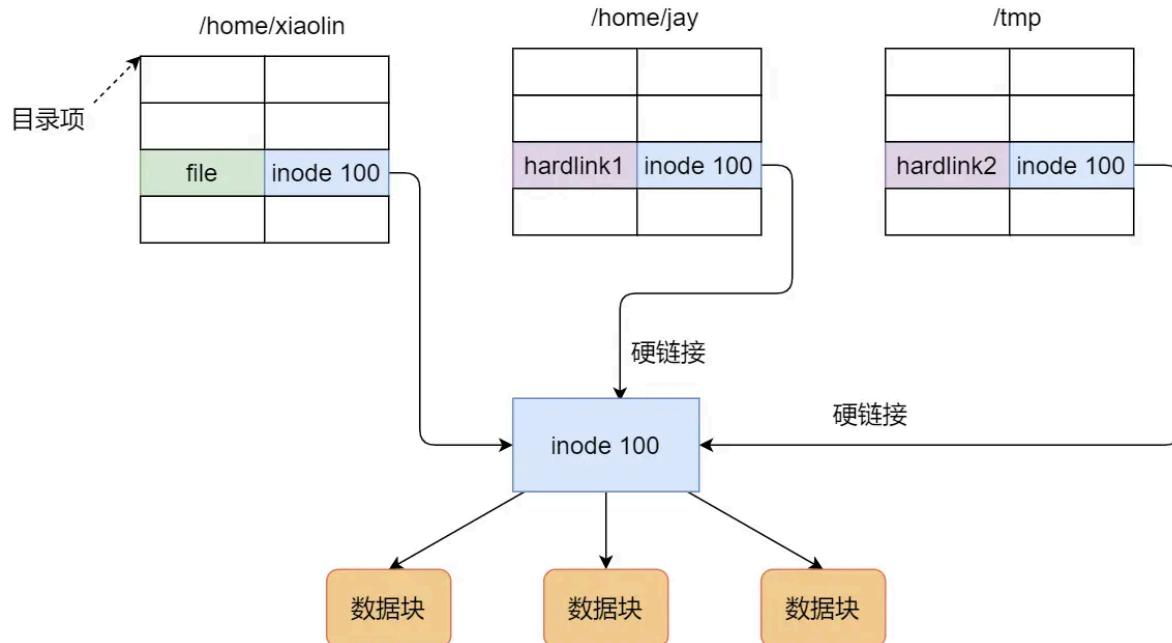


但通过遍历目录文件中进行文件和目录名匹配效率会比较低，因此通过保存其哈希值进行匹配，但需要解决哈希冲突的问题。此外，为了避免频繁读写磁盘，操作系统会将使用过的目录文件缓存至内存中，加快速度。

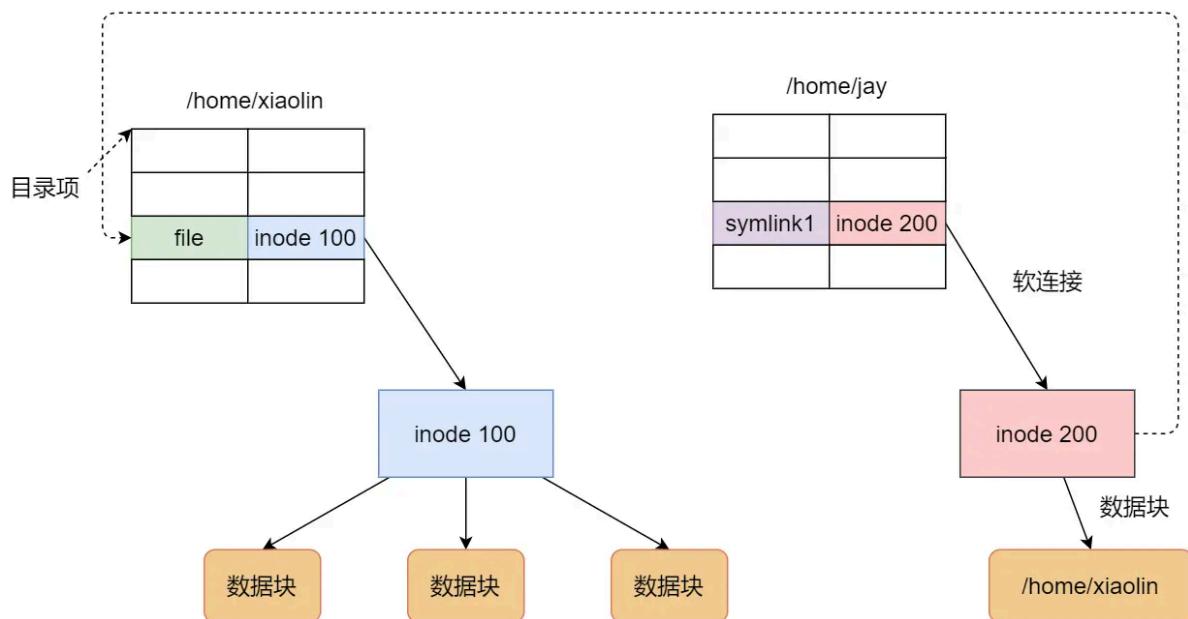
硬链接和软链接

在Linux系统中想要给一个文件或目录设置一个别名，使得可以通过这个别名来进行访问，可以通过**硬链接**和**软链接**实现。

硬链接和普通文件类似，但其目录项中的inode编号和源文件一致，多个目录项中的inode一致指向同一个文件，实现硬链接。硬链接的文件属性、内容和源文件一致，只有当所有的硬链接和源目标文件都被删除时，系统才会彻底删除该文件。硬链接不支持跨文件系统，因为不同文件系统的inode数据结构和列表可能不同。



软链接相当于创建一个新的文件，不同于硬链接中目录项的inode和源文件一致，软链接目录项中的inode号与源文件不相同，软链接的文件内容为源文件的链接，并且软链接支持跨文件系统。源文件被删除时不影响软链接的存在，但指向的文件打不开。



文件I/O

缓冲和非缓冲I/O

根据是否利用标准库缓冲，可把文件I/O分为缓冲I/O和非缓冲I/O：

- 缓冲 I/O，利用的是标准库的缓存实现文件的加速访问，而标准库再通过系统调用访问文件。
- 非缓冲 I/O，直接通过系统调用访问文件，不经过标准库缓存。

直接I/O和非直接I/O

根据是否利用操作系统的缓存（page cache），可把文件I/O分为直接I/O和非直接I/O：

- 直接I/O：不会将数据经过操作系统缓存，直接通过文件系统访问磁盘
- 非直接I/O：读操作时，将数据拷贝至操作系统缓存；写操作是先写入操作系统缓存，再写入磁盘

如果你在使用文件操作类的系统调用函数时，指定了 O_DIRECT 标志，则表示使用直接 I/O。如果没有设置过，默认使用的是非直接 I/O。

内核缓存中的数据在以下几种情况会被写入磁盘：

1. 调用write后，缓存中数据很多
2. 显式使用sync将内核缓存刷新到磁盘
3. 计算机已无物理内存，需要进行内存回收，会将缓存写入磁盘
4. 内核缓存的数据缓存时间超过某个阈值，缓存中的数据将被写入磁盘

阻塞与非阻塞I/O VS 同步与异步I/O

不多说了，前面总结过了。

将I/O分为两个过程的：

1. 数据准备的过程
2. 数据从内核空间拷贝到用户进程缓冲区的过程

阻塞I/O会阻塞在「过程 1」和「过程 2」，而非阻塞I/O和基于非阻塞I/O的多路复用只会阻塞在「过程 2」，所以这三个都可以认为是同步I/O。异步I/O则不同，「过程 1」和「过程 2」都不会阻塞。

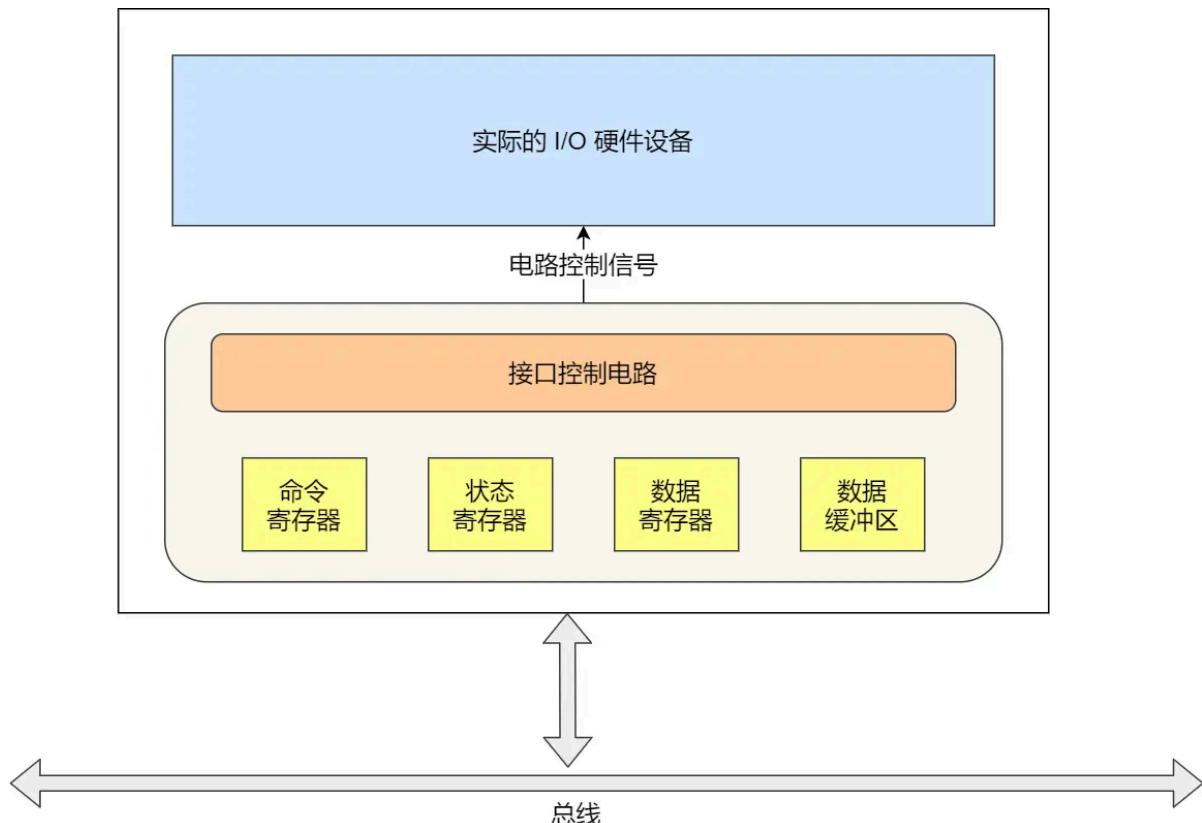
设备控制器

操作系统的一个重要功能是设备管理，计算机拥有很多设备，例如键盘、鼠标、显示器等，为了屏蔽这些设备的差异性，每个设备都有一个设备控制器，CPU通过与这些设备控制器进行交互来操控管理各种设备。

设备控制器通常有芯片，可以执行自己的逻辑，也有寄存器与CPU进行通信交互：

- CPU可以写入设备控制器中的寄存器，来发送数据和命令，对设备进行操作
- CPU可以从设备控制器中的状态寄存器读取信息，来获取设备状态（比如设备是否在工作）

设备控制器中的寄存器可分为命令寄存器、数据寄存器和状态寄存器，块设备还有缓冲区。



CPU对设备控制器中的寄存器进行读写比起直接读写设备会方便很多，也可以屏蔽很多差异。输入输出设备（或者说所有设备）可分为块设备和字符设备：

- 块设备：以块（block）为单位进行数据传输，将数据存储在块中，每个块有特定的地址，可以随机访问。硬盘和USB就是典型的块设备
- 字符设备：以字符为单位进行数据传输（通常是一个个字符流），只能按顺序读取，不能随机访问。鼠标和键盘是典型的字符设备

块设备由于以块为单位进行数据传输，数据通常较大，因此块设备控制器中通常还有缓冲区，从而避免对设备的频繁操作：

- CPU将数据写入缓冲区等到一定量时，再写入设备
- 缓冲区的数据达到一定量时，CPU才将其读入内存

CPU对设备控制器寄存器的读写可以通过2种方式：

1. 端口I/O，为设备控制器分配一个I/O端口（一种独立于内存池地址的逻辑地址，用于连接CPU和外部设备的接口，可以通过指令访问），通过特殊的汇编指令进行读写，比如in/out
2. 映射I/O，将设备控制器寄存器映射入内存中，直接进行读写

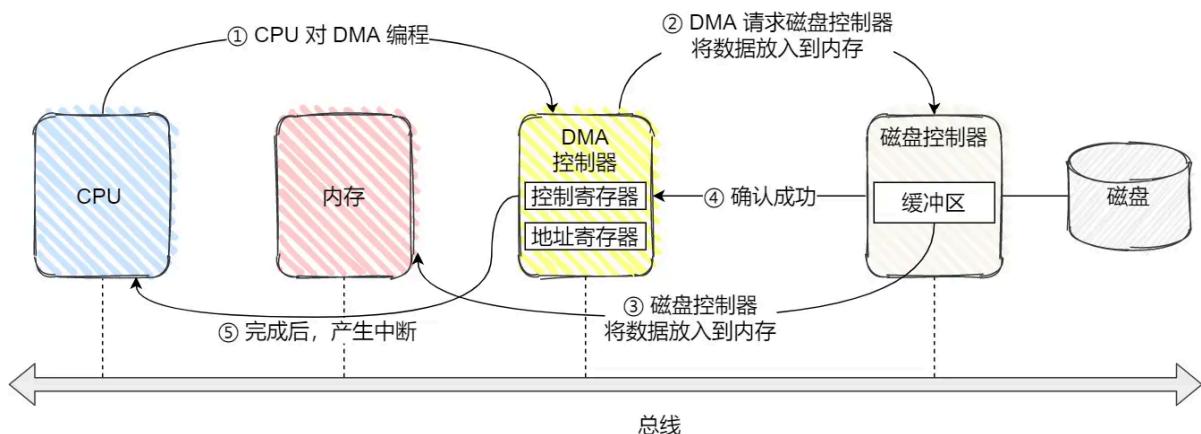
I/O控制方式

前面提到设备控制器通常有芯片，可以执行一些逻辑。假如CPU需要读取设备中的一些数据，CPU向设备控制器中的命令寄存器发送指令，CPU发送完成后就去执行自己的任务。当设备控制器完成设备数据的读取，应该如何通知CPU呢？比较常见的有2种方式：

- 轮询等待。很蠢的一种方式，CPU会一直查询设备控制器中的状态寄存器并等待，浪费不必要的时间
- 中断。中断可分为软中断（INT指令）和硬件中断，硬件则一般通过中断控制器通知CPU任务已完成

但中断对频繁读写数据的设备不太友好，比如磁盘，这样会经常打断CPU的工作。对于磁盘数据读取主要通过DMA，DMA的内容之前已经讲过了，这里再从设备的角度详细介绍一下具体的流程。

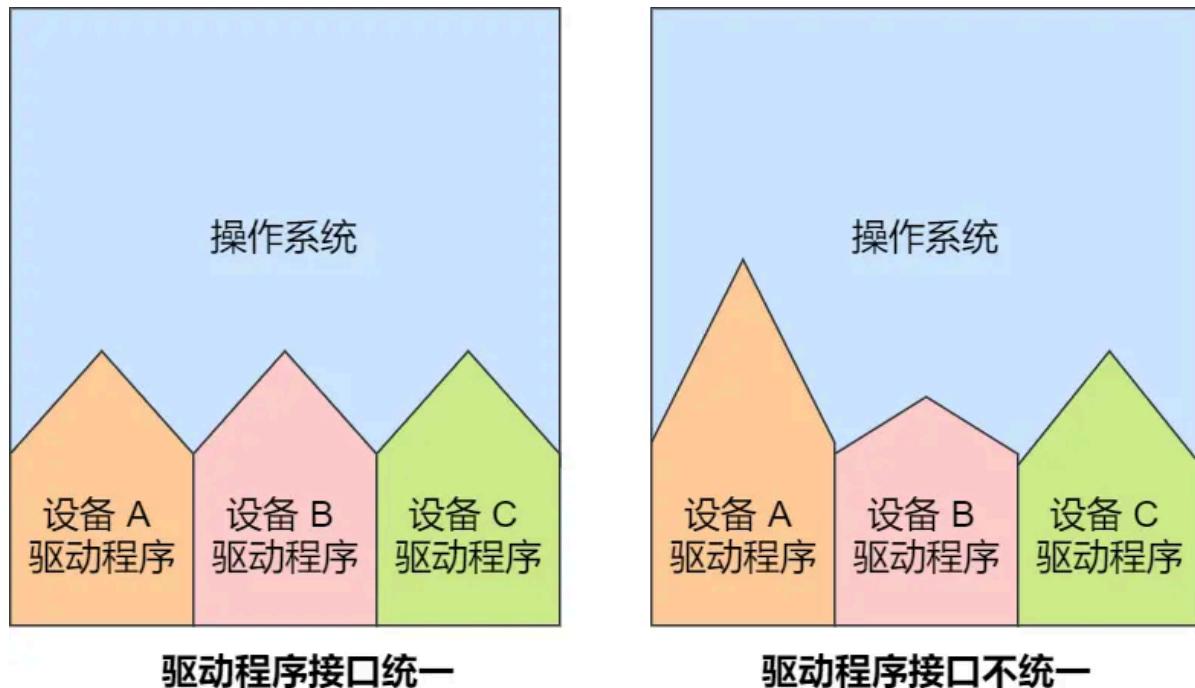
DMA也是硬件设备，有设备控制器。操作系统需要读取磁盘数据时，将控制指令和需要放置的内存地址发送至DMA设备控制器。DMA控制器再将命令发送至磁盘设备控制器，磁盘设备控制器将数据读取到内存后，在总线上发送一个确认成功的信号到DMA控制器，DMA控制器收到信号后，再通过中断通知操作系统任务已经完成。整个过程操作系统/cpu只有开始和结尾阶段有参与。



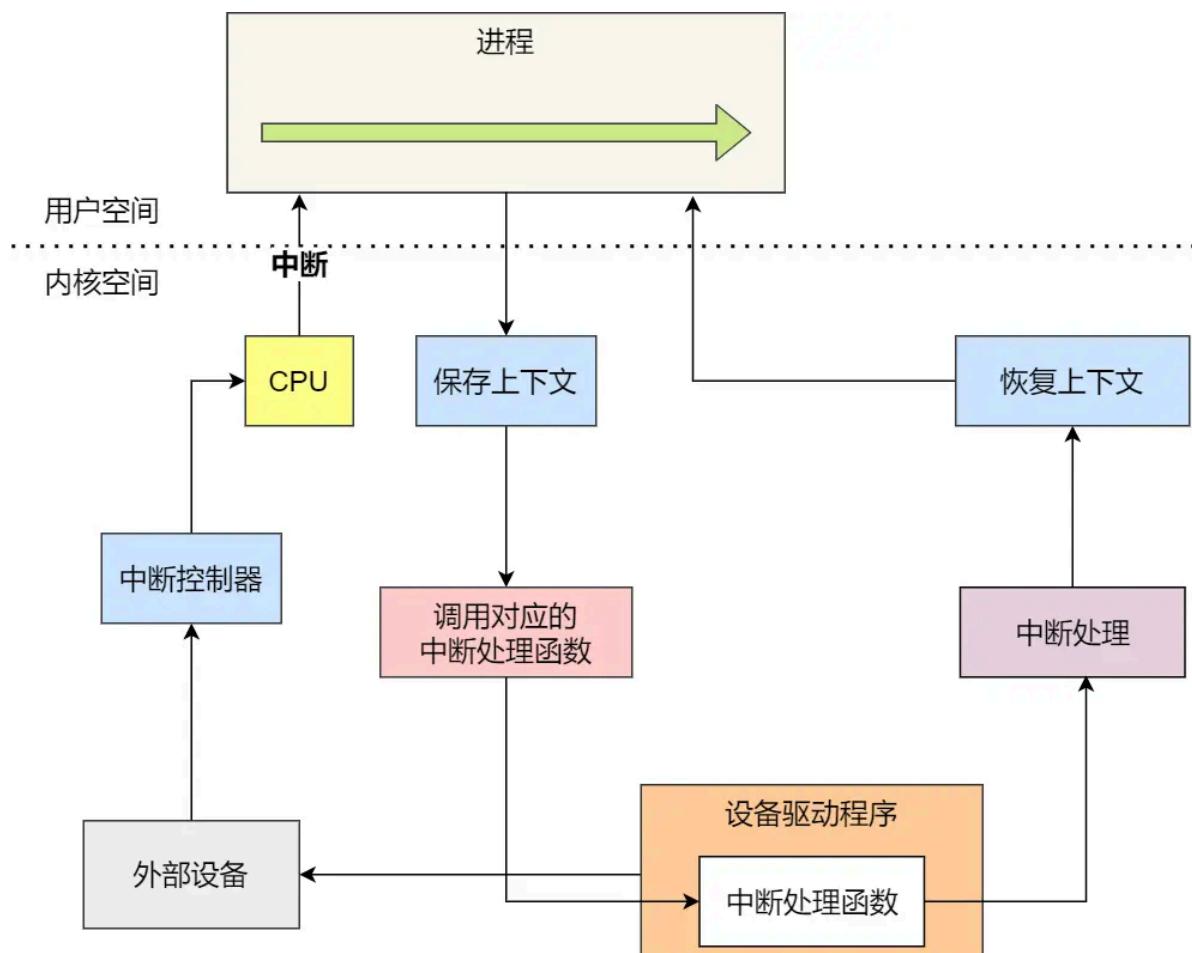
设备驱动程序

设备控制器屏蔽了不同设备的差异，但不同设备控制器中寄存器、缓冲区的使用方式又各不相同。因此，为了屏蔽不同设备控制器的差异，出现了设备驱动程序。设备控制器是硬件，而设备驱动程序是操作系统的一部分，是面向设备控制器设计的，通过设备驱动程序操作设备控制器。

设备驱动程序提供统一的接口接入操作系统。



当设备向操作系统发送中断信号通知事件完成时，操作系统会调用对应设备驱动程序的中断处理程序来处理，通常驱动程序初始化时会注册一个中断处理程序。



通用块层

对于块设备，为了减少不同块设备之间的差异，Linux通过一个统一的**通用块层**来管理不同的块设备。通用块层位于文件系统和块设备驱动程序之间，属于内核部分，主要有2个功能：

1. 向上为文件系统和应用程序提供块设备的通用接口，向下将不同的块设备抽象为统一的块设备，并在内核层提供统一的框架来管理不同的块设备驱动程序

2. 进行I/O调度，也就是对文件系统和应用和程序的I/O请求进行排序合并，提高磁盘读写效率

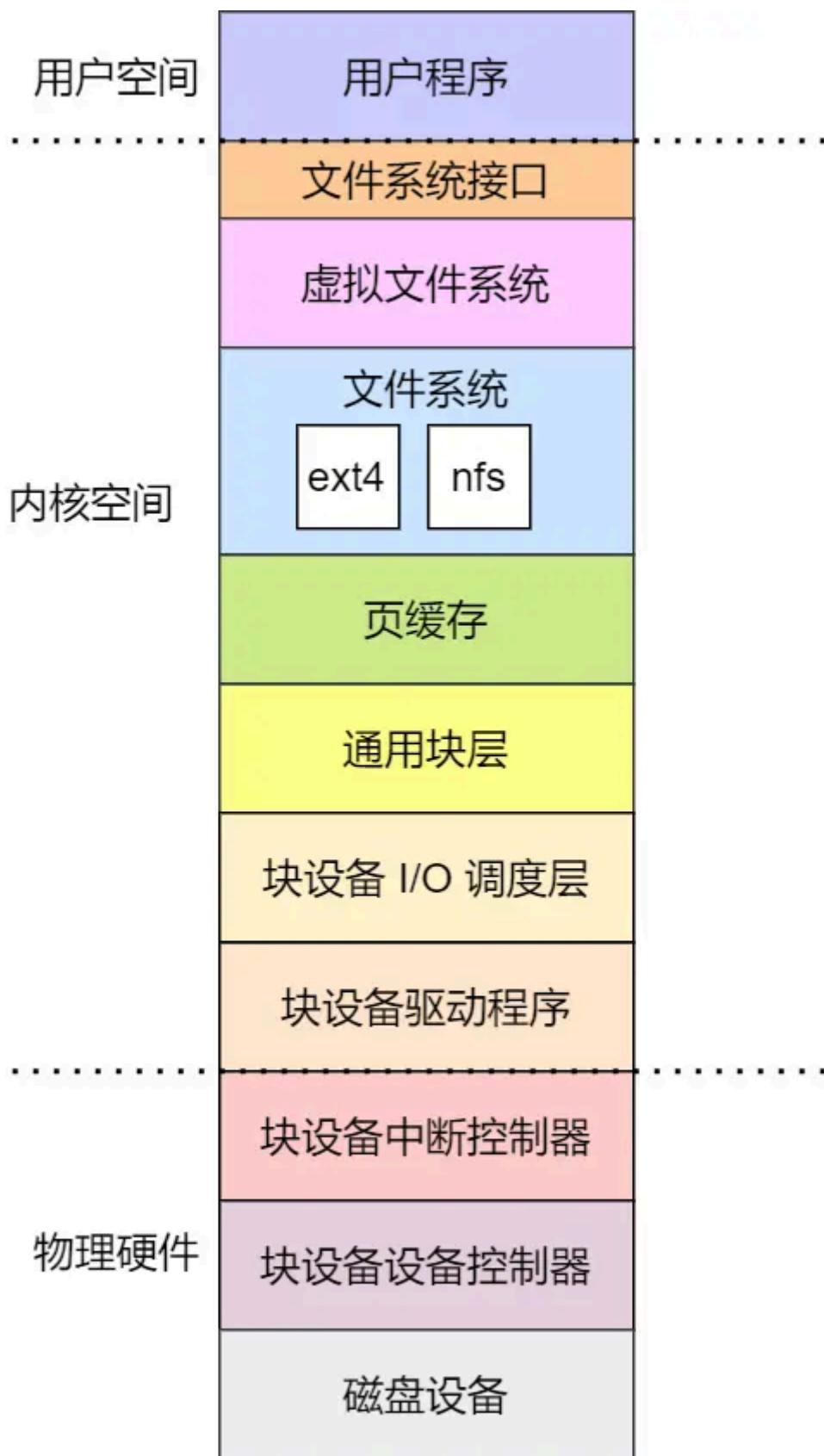
Linux支持5种I/O调度算法：

1. 没有调度算法，存在于虚拟机I/O中，具体的I/O调度由物理机负责
2. FIFO
3. 完全公平调度，大部分调度器的默认算法
4. 优先级调度
5. 最终期限调度

存储系统I/O软件分层

Linux存储系统的I/O从上自下可以分为3个层次

- 文件系统层：包括虚拟文件系统和其他文件系统，为应用程序提供了访问文件的标准接口，向下通过通用块层访问和管理磁盘数据
- 通用块层：包括块设备的I/O队列和I/O调度器
- 设备层：包括硬件设备，设备控制器和设备驱动程序，负责最终的物理I/O操作。



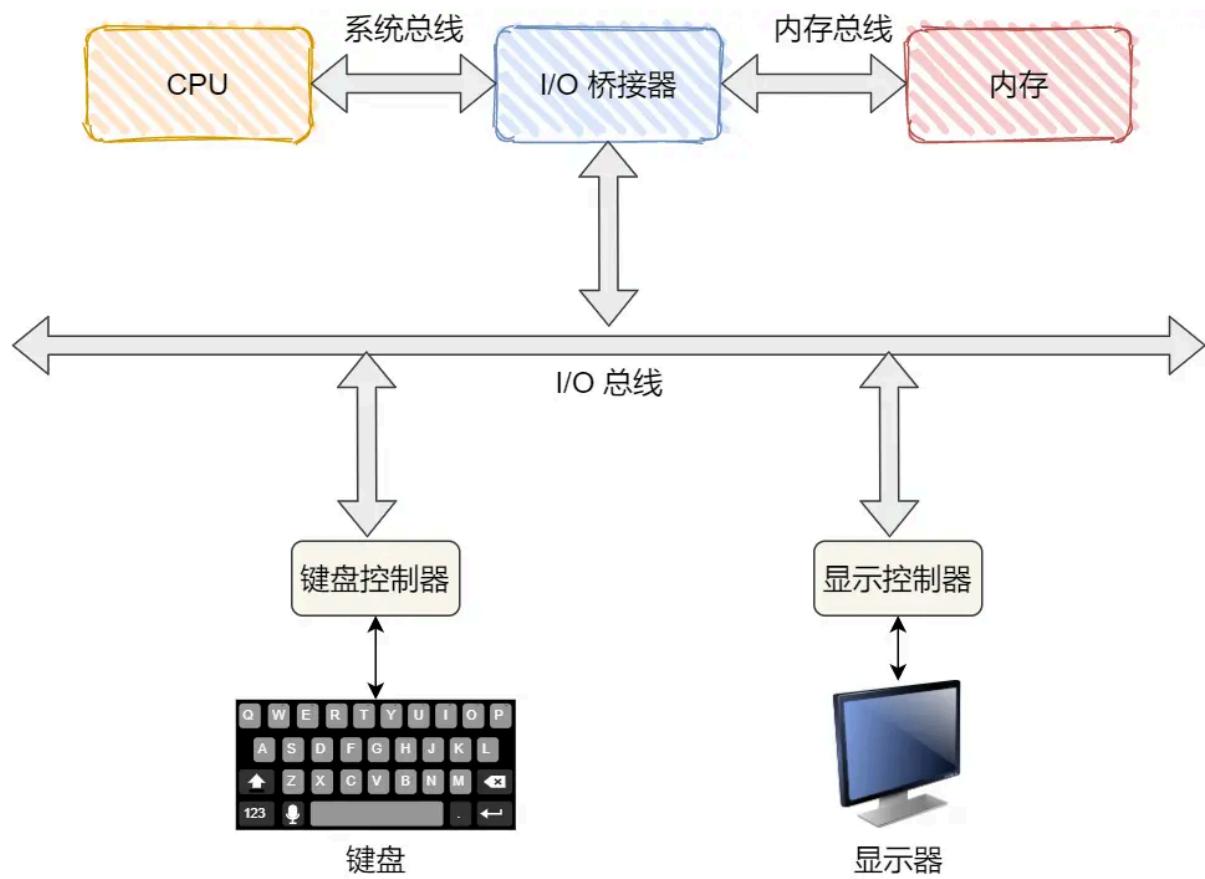
通过文件系统接口，使得Linux下万物皆文件。

由于存储系统的I/O很慢，Linux提供了很多缓存机制来提高I/O效率：

1. 通过**页缓存**，**索引节点缓存**和**目录项缓存**等多种机制，减少对块设备的直接调用
2. 通过**缓冲区**提高块设备的访问效率

键盘敲入字母会发生什么

通过I/O桥接器，将CPU、内存和I/O总线连接起来。



当使用键盘敲入字符，比如A时，键盘产生一个信号，键盘设备控制器将数据（扫描码）缓冲在寄存器中，并通过中断给CPU发送请求。CPU收到请求后，首先保存上下文，然后通过键盘的驱动程序中的中断处理程序进行处理。中断处理函数读取寄存器中的数据，如果是显示字符则会将其翻译成对应的ASCII码，并将其放置读缓冲区队列中。显示器的驱动程序会定期从读缓冲区队列中读取数据，并放置写缓冲区队列，再将写缓冲区队列中的数据写入显示设备控制器中寄存器的数据缓冲区中，最后再进行显示。一旦中断处理程序完成，CPU将恢复上下文。