目录

参考书目

书名	版本	简写	偏移量*
《C++ Primer》	中文第 5 版		26
《STL 源码剖析》			33
《C++标准库》	中文第 2 版		27
《算法导论》	中文第 3 版	CLRS	15
《深入理解计算机系统》	中文第 3 版	CSAPP	
《现代操作系统》	中文第 4 版		
《计算机网络：自顶向下方法》	中文第 6 版		
《Unix 网络编程 卷 1：套接字联网 API》	中文第 3 版	UNP	14
《数据库系统概念》	中文第 6 版		29
超全面的后端 C/C++面经整理			
《TCP/IP 详解 卷 1：协议》	中文第 2 版		20
《Linux 高性能服务器编程》			18
计算机网络微课堂： https://www.bilibili.com/video/BV1c4411d7jb			
2020 王道考研 数据结构： https://www.bilibili.com/video/BV1b7411N798			
2019 王道考研 操作系统： https://www.bilibili.com/video/BV1YE411D7nH			

* 偏移量：指本人所用电子版 PDF 的页码和书籍实际页码的差值，比如《C++ Primer》PDF 中为第 71 页，书籍实际页码为 45 页，那么差值为 71-45=26。本文出于阅读和跳转方便，标注的页码均为电子版 PDF 的页码。

本书主要整理和修改自：超全面的后端开发 C/C++面经整理分享：https://www.nowcoder.com/discuss/489210

整理者：Hansimov

C++

- C++经典面试题（最全，面中率最高）：https://zhuanlan.zhihu.com/p/75347892
- C/C++ 技术面试基础知识总结：https://github.com/huihut/interview#cc

1. 指针和引用的区别

- 《C++ Primer》第 71~75 页
- 浅谈 C++中指针和引用的区别：https://www.cnblogs.com/dolphin0520/archive/2011/04/03/2004869.html

	指针	引用
1	指针本身就是一个对象	引用本身并非对象，它只是为一个已经存在的对象所起的别名
2	允许对指针赋值和拷贝，在指针的生命周期内其可以先后指向不同的对象，给指针赋值就是令其存放一个新地址，指向一个新对象	一旦定义了引用，就无法令其再绑定到另外的对象，之后每次使用该引用都是访问其最初绑定的那个对象
3	指针无需在定义时赋初值，指针可以为空	引用必须初始化，且引用不能为空
4	sizeof 得到的是指针自身的大小，一般为 4 字	sizeof 得到的引用大小取决于被引用对象大小
5	指针可以有多级	引用只能有一级
6	可以用 const 指针	没有 const 引用
7	指针传参时需要解引用才能修改参数	引用传参时可以直接修改参数

2. 在函数参数传递时，何时用指针，何时用引用

1. 使用指针传递的场景

需要返回函数内局部变量的内存的时候用指针（返回局部变量的引用是没有意义的）。使用指针传参需要开辟内存，用完要记得释放指针，不然会内存泄漏。

2. 使用引用传递的场景

（1）对栈空间大小比较敏感（比如递归）时使用引用，使用引用传递无需创建临时变量，开销更小。

（2）类对象作为参数传递的时候使用引用，这是 C++ 类对象传递的标准方式。

- C++ 值传递、指针传递、引用传递详解： https://www.cnblogs.com/yanlingyin/archive/2011/12/07/2278961.html

1. 值传递：形参是实参的拷贝，改变形参的值并不会影响外部实参的值。从被调用函数的角度来说，值传递是单向的（实参->形参），参数的值只能传入，不能传出。当函数内部需要修改参数，并且不希望这个改变影响调用者时，采用值传递。

2. **指针传递：**形参为指向实参地址的指针，当对形参的指向操作时，就相当于对实参本身进行的操作。
3. **引用传递：**形参相当于是实参的“别名”，对形参的操作其实就是对实参的操作，在引用传递过程中，被调函数的形式参数虽然也作为局部变量在栈中开辟了内存空间，但是这时存放的是由主调函数放进来的实参变量的地址。被调函数对形参的任何操作都被处理成间接寻址，即通过栈中存放的地址访问主调函数中的实参变量。正因为如此，被调函数对形参做的任何操作都影响了主调函数中的实参变量。

3. 引用作为函数参数以及返回值的好处

对比值传递，引用传参的好处：

1. 在函数内部可以对此参数进行修改
   - 函数调用时，值的传递机制是通过 “形参 = 实参” 来对形参赋值达到传值目的，产生了一个实参的副本。函数内部对参数的修改都是针对形参，也即实参的副本，实参不会有任何更改。函数一旦结束，形参生命也宣告终结，做出的修改不会对变量产生影响
2. 提高函数调用和运行的效率
   - 没有了传值和生成副本的时间和空间消耗
   - 用引用作为返回值时，在内存中不产生被返回值的副本

但是引用传参有如下限制：

- **不能返回局部变量的引用：**因为函数返回以后局部变量就会被销毁
- **不能返回函数内部 new 分配的内存的引用：**虽然不存在局部变量的被动销毁问题，但如果被函数返回的引用只是作为一个临时变量而未被赋予一个实际的变量，那么该引用所指向的空间（由 new 分配）就无法释放，会造成内存泄露
- **可以返回类成员的引用，但是最好是 const：**因为如果其他对象可以获得该属性的非常量的引用，那么对该属性的单纯赋值就会破坏业务规则的完整性

4. 堆和栈有什么区别

C++ 堆和栈的区别，内存分配方式理解：https://blog.csdn.net/qq_35637562/article/details/78550953

		堆	栈
1	管理方式	程序员手动分配，容易产生内存泄露	编译器自动管理，无需手动控制
2	分配方式	堆都是动态分配的，没有静态分配的堆	栈既有静态分配，也有动态分配。静态分配由编译器完成，比如局部变量的分配。动态分配由 malloc 函数完成，且由编译器释放，无需手动实现（和堆不同）。
3	空间大小	堆的空间较大，在 32 位系统下，堆内存可达 4 GB，几乎不存在限制。	栈的空间较小，比如 VC6 下默认为 1 MB。

4	碎片问题	堆空间有频繁的分配（new）和释放（delete）操作，这会导致内存空间不连续，产生内存碎片	栈中的块先进后出，始终只有栈顶的块最先被弹出。
5	生长方向	堆的生长方向向上↑，也即内存地址不断增加	栈的生长方向向下↓，也即内存地址不断减小
6	分配效率	堆的分配效率低，因为是由 C/C++函数库提供的，机制较为复杂。在为一个堆分配内存时，需要按照一定的算法搜索可用的足够大的空间，没有的话还要特地增加程序数据段的内存空间。	栈的分配效率高，因为是机器系统提供的数据结构，会在底层提供对栈的支持（比如分配专门的寄存器存放栈的地址，入栈出栈都有专门的指令）。
7	适用场景	需要分配大量内存空间时使用堆	在大部分情况下使用栈，因为优点很多

5. 堆和栈哪个更快，为什么

栈更快一点。

- 操作系统在底层对栈提供支持，专门的寄存器存放栈的地址，专门的入栈出栈指令。
- 堆的操作是由 C/C++函数库提供的，分配堆内存时需要使用相关算法寻找大小合适的内存，并且获取堆的内容需要两次访问，第一次访问指针，第二次依据指针保存的地址访问内存。

6. new 和 delete 是如何实现的

- C++：new 和 delete 背后的实现原理：https://blog.csdn.net/tonglin12138/article/details/85699115
- C++学习——深度剖析 new 和 delete 原理：https://blog.csdn.net/weixin_41028343/article/details/102952840
- C++ new/delete 详解及原理：https://www.cnblogs.com/Duikerdd/p/11687621.html
- new 和 delete 的底层实现原理：https://blog.csdn.net/qq_42719751/article/details/90084812

new/new[] 和 delete/delete[] 都是 C++用来实现动态内存管理的操作符。new/new[] 申请空间，delete/delete[] 释放空间。operator new 和 operator delete 是系统提供的全局函数，new 和 delete 在底层调用全局函数来申请和释放空间。

1、对于内置类型

相同之处：申请内置类型的空间时，new 和 malloc 基本类似，delete 和 free 基本类似。

不同之处：（1）new 和 delete 申请和释放的是单个元素的空间，new[] 和 delete[] 申请和释放的是连续空间。（2）new 在申请空间失败时会抛出异常，malloc 会返回 NULL。

2、对于自定义类型

new：（1）调用 operator new 函数申请空间，该函数实际上通过 malloc 来申请空间；（2）在申请的空间上执行构造函数，完成对象的构造

delete：（1）在空间上执行析构函数，完成对象中资源的清理工作；（2）调用 operator delete 函数释放对象的空间，该函数实际上通过 free 来释放空间

new [N]：（1）调用 operator new[] 函数，在 operator new[] 中实际调用 operator new 函数完成 N 个对象空间的申请；（2）在申请的空间上执行 N 次构造函数

delete [N]：（1）在释放的对象空间上执行 N 次析构函数，完成 N 个对象中资源的清理；（2）调用 operator delete[] 释放空间，在 operator delete[] 中实际调用 operator delete 来释放空间

问：那么在 new[N] 和 delete[N] 过程中如何确定本次调用的是第几次函数？

答：在使用 new[] 时，调用 operator new[](size_t size) 函数，传参时传入的不是 sizeof(类型)* 对象个数；而是**在对象数组的大小上加上一个额外数据，用于编译器区分对象数组指针和对象指针以及对象数组大小**。在 VS2008 下这个额外数据占 4 个字节，一个 int 大小。但是，这个额外数据对外是不可见的。

operator new() 的全局函数原型

```
1  /*
2      operator new:
3      该函数实际通过 malloc 来申请空间，当 malloc 申请空间成功时直接返回；
4      申请空间失败，尝试执行空间不足应对措施，如果改应对措施用户设置了，则继续申请，否则抛异常。
5  */
6  void *__CRTDECL operator new(size_t size) _THROW1(_STD bad_alloc) {
7      // try to allocate size bytes
8      void *p;
9      while ((p = malloc(size)) == 0)
10         if (_callnewh(size) == 0) {
11             // report no memory
12             // 如果申请内存失败了，这里会抛出 bad_alloc 类型异常
13             static const std::bad_alloc nomem;
14             _RAISE(nomem);
15         }
16     return (p);
17 }
18
```

- operator delete() 的全局函数原型

```
1  /*
2      operator delete: 该函数最终是通过 free 来释放空间的
3  */
4  void operator delete(void *pUserData) {
5      _CrtMemBlockHeader * pHead;
```

- operator delete() 的全局函数原型

```
6
7       RTCCALLBACK(_RTC_Free_hook, (pUserData, 0));
8
9       if (pUserData == NULL)
10          return;
11
12      _mlock(_HEAP_LOCK); /* block other threads */
13      __TRY
14      /* get a pointer to memory block header */
15      pHead = pHdr(pUserData);
16
17      /* verify block type */
18      _ASSERTE(_BLOCK_TYPE_IS_VALID(pHead->nBlockUse));
19
20      _free_dbg(pUserData, pHead->nBlockUse);
21
22      __FINALLY
23      _munlock(_HEAP_LOCK); /* release other threads */
24      __END_TRY_FINALLY
25
26      return;
27  }
28
29  /*
30      free 的实现
31  */
32  #define free(p) _free_dbg(p, _NORMAL_BLOCK)
33
```

7. new 和 malloc 的异同

※ 细说 new 与 malloc 的 10 点区别：https://www.cnblogs.com/qg-whz/p/5140930.html

		new/delete	malloc/free
1	库函数和运算符	new 和 delete 是运算符。	malloc 和 free 是库函数。
2	分配内存的位置	new 从**自由存储区（free store）**上为对象动态分配内存空间。[1,2]	malloc 从**堆**上动态分配内存。
3	内存分配成功的返回值	new 内存分配成功时，返回**对象类型的指针**，类型严格与对象匹配，无须进行类型转换，故 new 是类型安全的操作符。[3]	malloc 内存分配成功时，返回 **void***，需要通过强制类型转换将 void* 指针转换成我们需要的类型。
4	内存分配失败的返回值	new 内存分配失败时抛出 **bad_alloc 异常**，不会返回 NULL。[4]	malloc 内存分配失败时会返回 **NULL**。

5	是否指定分配内存的大小	new 分配内存无需指定内存块的大小，编译器会根据类型信息自动计算。	malloc 需要显式地指出所需内存的大小。[5]
6	是否调用构造函数和析构函数	new 会调用构造函数，delete 会调用析构函数。[6,7]	malloc 不会调用构造函数。
7	处理数组	new[] 会分别调用构造函数初始化每一个数组元素，释放时为每个对象调用析构函数。使用 new[] 分配的内存必须使用 delete[] 释放，否则会只释放部分对象，造成内存泄露。	malloc 不知道在内存上申请的是数组还是别的什么，只会给你一块原始内存和地址。因此要用 malloc 动态分配数组的内存，需要手动指定数组的大小。
8	是否可以相互调用	operator new 和 operator delete 的实现可以基于 malloc 和 free。[8]	malloc 的实现不能调用 new。
9	是否可以被重载	operator new 和 operator delete 可以被重载。[9]	malloc 不允许重载。
10	扩充已分配内存	new 没有相关处理方式。	使用 malloc 分配内存后，可以用 realloc 函数重新分配内存。[10]
11	内存分配时内存不足	new 可以指定处理函数或重新制定分配器。[11]	malloc 没有相关处理方式。

一些说明：

1. 凡是通过 new 申请的内存，该内存即为自由存储区。
2. 自由存储区能否是堆？

   该问题等价于 new 能否在堆上动态分配内存。这取决于 operator new 的实现细节。自由存储区可以是堆，也可以是静态存储区，取决于 operator new 在何处为对象分配内存。

   特别地，new 甚至可以不为对象分配内存，比如**定位 new**：

```
new (place_address) type;
void * operator new (size_t, void *) // 不允许重定义该版本的 operator new
```

3. 类型安全很大程度上等价于内存安全，类型安全的代码不会试图访问未被授权的内存区域。
4. 使用 C 语言时，习惯在 malloc 分配内存后用 NULL 判断分配是否成功：

```
int *a  = (int *)malloc ( sizeof (int ));
if(NULL == a) {
    ...
} else {
    ...
}
```

   使用 C++时，则不应在 new 后使用 NULL 判断，而应捕捉 bad_alloc 异常：

```
try {
    int *a = new int();
```

```
3   } catch (bad_alloc) {
4       ...
5   }
```

5. new 不需要指定内存大小，malloc 则需要：

```
1   class A{...}
2   A * ptr = new A;
3   A * ptr = (A *)malloc(sizeof(A)); // 需要显式指定所需内存大小 sizeof(A);
4   // 当然，这里仅作示意，实际上不应使用 malloc 来处理自定义数据类型
```

6. new 分配对象内存有 2 步：

   （1）调用 operator new（或者 new[]）函数分配一块足够大的、原始的、未命名的空间存储特定类型的对象；

   （2）编译器执行相应的构造函数以构造对象，并传入初值；对象构造完成后，返回一个指向该对象的指针。

   delete 分配对象内存有 2 步：

   （1）调用对象的析构函数；

   （2）编译器调用 operator delete（或者 delete[]）函数释放内存空间。

7. 不应使用 malloc 和 free 函数处理 C++中的自定义数据类型（包括标准库中需要构造和析构的数据类型）。

8. 一种极简的 operator new 和 operator delete 的实现方式：

```
1   void * operator new(sieze_t size) {
2       if(void * mem = malloc(size))
3           return mem;
4       else
5           throw bad_alloc();
6   }
7   void operator delete(void *mem) noexcept {
8       free(mem);
9   }
```

9. 标准库为 operator new 和 operator delete 各定义了 4 个重载版本：

```
1   // 这些版本可能抛出异常
2   void * operator new(size_t);
3   void * operator new[](size_t);
4   void * operator delete(void *) noexcept;
5   void * operator delete[](void *0) noexcept;
6   // 这些版本承诺不抛出异常
7   void * operator new(size_t, nothrow_t&) noexcept;
8   void * operator new[](size_t, nothrow_t& );
9   void * operator delete(void *, nothrow_t& ) noexcept;
```

```
10  void * operator delete[](void *0, nothrow_t& ）noexcept;
```

10. realloc 先判断当前指针所指内存是否有足够的连续空间，如果有，原地扩大可分配的内存地址，并且返回原来的地址指针；如果空间不够，先按照新指定的大小分配空间，再将原有数据从头到尾拷贝到新分配的内存区域，最后释放原来的内存区域。
11. 在 operator new 因为申请的内存不足从而抛出异常之前，会先调用一个用户指定的错误处理函数 new_handler，这是一个指针类型，指向一个没有参数和返回值的错误处理函数：

```
1   namespace std {
2       typedef void (*new_handler)();
3   }
```

使用者需要调用 set_new_handler 指定错误处理函数，这是声明于 std 的一个标准库函数。set_new_handler 的参数为 new_handler 指针，指向 operator new 无法分配足够内存时应当调用的函数；其返回值也是指针，指向 set_new_handler 被调用前正在执行（但马上就要被替换）的那个 new_handler 函数。

```
1   namespace std {
2       new_handler set_new_handler(new_handler p) throw();
3   }
```

8. 既然有了 malloc/free，C++中为什么还需要 new/delete

有了 malloc/free 为什么还要 new/delete ：https://blog.csdn.net/liubing8609/article/details/85568356

- 由于内部数据类型没有构造与析构的过程，因此对它们而言 malloc/free 和 new/delete 是等价的。
- 对于非内部数据类型的对象而言，光用 malloc/free 无法满足动态对象的要求。对象在创建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数。

由于 malloc/free 是库函数而不是运算符，不在编译器控制权限之内，不能够把执行构造函数和析构函数的任务强加于 malloc/free。因此 C++语言需要一个能完成动态内存分配和初始化工作的运算符 new，以及一个能完成清理与释放内存工作的运算符 delete。

注意：malloc/free 是库函数，而 new/delete 是运算符。

问：既然 new/delete 的功能完全覆盖了 malloc/free，为什么 C++不摒弃 malloc/free 呢？

答：因为 C++程序经常要调用 C 函数，而 C 程序只能用 malloc/free 管理动态内存。

问：free 和 delete 能否换用？

答：如果用 free 释放“new 创建的动态对象”，那么该对象因无法执行析构函数而可能导致程序出错。如果用 delete 释放“malloc 申请的动态内存”，理论上讲程序不会出错，但是该程序的可读性很差。所以 new/delete 和 malloc/free 应该配对使用。

9. C 和 C++的区别

C++ 学习基础篇（一）—— C++与 C 的区别：https://blog.csdn.net/zqixiao_09/article/details/51235444

	C++	C
1	面向对象，有封装、继承和多态的特性 封装：隐藏了实现细节，使代码模块化 继承：通过子类继承父类的方法和属性实现了代码重用 多态：一个接口，多个实现，通过子类重写父类的虚函数，实现接口重用	面向过程
2	除 malloc 和 free 外，还有 new 和 delete	使用 malloc 和 free 管理内存
3	有函数重载和引用	无
4	允许变量定义语句在程序中的任何地方，只要在使用之前； 允许重复定义变量	在某些 C 语言的版本中，变量定义必须在函数开头部分； 不允许重复定义变量

10. C++和 Java 的联系与区别，包括语言特性、垃圾回收、应用场景等

	C++	Java
1	都是面向对象的语言	
2	编译成可执行文件直接运行	编译之后在 Java 虚拟机上运行，故 Java 有良好的跨平台特性，但是执行效率不如 C++
3	内存管理由程序员手动进行	内存管理由 Java 虚拟机自动完成，采用标记-回收算法
4	有引用和指针	只有引用，没有指针
5	有构造函数和析构函数	只有构造函数，没有析构函数

11. Java 的垃圾回收机制

略。

12. 为什么 C++没有实现垃圾回收？

- c++没有垃圾回收机制的原因：https://blog.csdn.net/jx232515/article/details/52749551

1. 垃圾回收会增加时间和空间开销：垃圾回收需要开辟一定的空间保存指针的引用计数，并对它们进行标记，还需要单独开辟一个线程在空闲的时候进行 free 操作。而 C++本身就是为了高效设计的，这不符合 C++的设计哲学。

2. C++诞生的年代内存很少，不需要多此一举
3. 垃圾回收影响了一些底层操作
4. 有替代方法：析构函数，智能指针，引用计数管理资源释放

13. C++和 Python 的区别

	C++	Python
1	是编译语言，编译后在特定平台运行	是解释型语言，使用解释器逐行解释为机器码执行，跨平台性好，但效率没有 C++高
2	使用花括号区分不同的代码块	使用缩进区分代码块
3	需要事先定义变量的类型	不需要事先定义变量类型
4	较为复杂	库函数多，使用更方便

14. C++中 const 的用法（定义，用途）

- 《C++ Primer》“第 2 章 变量和基本类型”“2.4 const 限定符”，第 79 页
- C 语言中 const 关键字的用法：https://blog.csdn.net/xingjiarong/article/details/47282255
- C++ const 的各种用法详解，const 用法深入浅出：https://www.cnblogs.com/wintergrass/archive/2011/04/15/2015020.html
- C++ const 关键字小结：https://www.runoob.com/w3cnote/cpp-const-keyword.html

const 的用法：

- 修饰全局和局部变量、引用：不能修改
- 修饰函数的参数：
  - 防止修改指针指向的内容

```
void StringCopy(char *strDestination, const char *strSource);
```

  - 防止修改指针指向的地址

```
void swap(int * const p1 , int * const p2)
```

- 修饰函数的返回值：

```
const char * GetString(void);   // 函数声明
const char *str = GetString();  // 函数使用
```

- 修饰指针：
  - 常量指针：不能通过该指针修改变量的值，但可以通过其他引用来修改变量的值。

```
int a=5;
const int* n=&a;
a=6;
```

  - 指针常量：指针本身是个常量，不能在指向其他的地址，写法如下：

```
int *const n;
```

- 指向常量的常指针：

```
const int *const n;
```

- 修饰类的实例（对象）：常量对象只能使用常量函数
- 修饰类的成员变量：表示常量不能被修改，类中的 const 成员变量不能直接初始化，初始化只能在类的构造函数的初始化表中完成。

```
1  class A {
2      A(int size);                  // 构造函数
3      const int SIZE;
4   };
5   A::A(int size) : SIZE(size) { … } // 构造函数的初始化表
```

- 修饰类的成员函数：表示该函数不会修改类中的数据成员，不会调用其他非 const 的成员函数。写在成员函数的右边。

const 的优点：（详见下面 define 和 const 的区别）

- 可以进行类型检查
- 可以保护被修饰的东西，防止意外修改，增强程序的健壮性
- 编译器通常不为普通 const 常量分配存储空间，而是将其保存在符号表中，这使得它成为一个编译期间的常量，无需存储和读内存的操作，提高了效率。

在 const 类型的对象上只能执行不改变其内容的操作。

- 使用 const 对象去初始化另一个对象是合法的，因为拷贝一个对象的值并不会改变它。
- 默认状态下，const 仅在文件内有效。当多个文件中出现了同名的 const 变量时，其实等同于在不同文件中分别定义了独立的变量。
- 若希望某些 const 变量可以在文件中共享，不希望编译器为每个文件分别生成独立的变量，也即只在一个文件中定义 const，在其他多个文件中声明并使用它，则不管是声明还是定义，都添加 extern 关键字，这样只需定义一次即可：

```
extern const int bufsize = fcn(); // file1.cc
extern cosnt int bufsize;         // file1.h，和 file1.cc 中定义的 bufsize 是同一个
```

  - 在文件 file1.cc 中定义并初始化了 bufsize，又因 bufsize 是一个 const 常量，故需使用 extern 限定，以允许其被其他文件使用
  - 在头文件 file1.h 中的 extern 限定，表示 bufsize 并非本文件所独有，其定义将在别处出现

顶层 const 和底层 const：

- 顶层 const：表示指针本身是个常量，也即指针的值不能改变，或者说，不能再指向其他对象。对任何数据类型都适用。

```
int i1 = 11;
```

```
int *const p1 = &i1; // 顶层 const，不允许改变 p1 的值，允许改变 i1 的值
```

- 底层 const：表示指针所指的对象是一个常量，也即指向的对象的值不能改变。一般用于指针和引用等复合类型的基本类型。

```
const int i2 = 22;
const int *p2 = &i2;        // 底层 const，允许改变指针 p2 的值，不允许改变 i2 的值
const int *const p3 = p2; // 右边的 const 是顶层 const，左边的 const 是底层 const
const int &r = i2;          // 用于声明引用的 const 都是底层 const
```

15. define 和 const 的联系与区别（编译阶段、安全性、内存占用等）

- C++ 宏定义 #define 和常量 const 的区别：https://www.runoob.com/note/12963

		define 宏定义	const 常量
	联系	都是定义常量的的一种方法	
1	类型	定义的常量没有类型，只是简单的字符替换	定义的常量有类型
2	安全性	不会进行类型安全检查，可能产生边际效应等错误	会进行类型安全检查
3	存储方式	可能会有多个拷贝，占用内存空间大（也有说不分配内存？），存储于程序的代码段中	只有一个拷贝，存储在静态存储区，占用内存空间小，存储于程序的数据段中
4	编译器处理	在预处理阶段进行替换，是“编译时”概念，不能对宏定义进行调试，生命周期结束于编译时期	在编译阶段确定值，是“运行时”概念，在程序运行使用，类似一个只读数据
5	定义域	不受函数作用域影响	受作用域影响，不能在定义域外使用
6	定义后能否取消	可以使用 #undef 使之前的宏定义失效	在定义域内永久有效
7	能否做函数参数	不能	能
8	能否定义函数	不难	能

16. C++ 中的 static 用法和意义

- c/c++ static 用法总结（三版本合一）：https://blog.csdn.net/mznewfacer/article/details/6898005
- https://github.com/huihut/interview#static

static 的作用主要有三个：

- **扩展生存期：**这是针对普通局部变量和 static 局部变量来说的。声明为 static 的局部变量的生存期不再是当前作用域，而是整个程序的生存期。
- **限制作用域：**这一点是对比普通全局变量和 static 全局变量来说的。对于全局变量，不论是普通全局变量还是 static 全局变量，其存储区都是静态存储区，因此在内存分配上没有什么区别。真正的区别在于：

  - 普通全局变量和函数：其作用域为整个程序或项目，外部文件（其它 cpp 文件）可以通过 extern 关键字访问该变量和函数。一般不提倡这种用法，如果要在多个 cpp 文件间共享数据，应该将数据声明为 extern 类型。
  - static 全局变量和函数：其作用域为当前 cpp 文件，其它的 cpp 文件不能访问该变量和函数。如果有两个 cpp 文件声明了同名的全局静态变量，那么他们实际上是独立的两个变量。static 函数的好处是不同的人编写不同的函数时，不用担心自己定义的函数和其它文件中的函数同名。
- **唯一性**：主要指类中的静态成员变量和函数。这是 C++对 static 关键字的重用，表示属于一个类而不是属于此类的任何特定对象的变量和函数。这是与普通成员函数的最大区别，也是其应用所在。
  - static 既不是限定作用域的，也不是扩展生存期的作用，而是指示变量/函数在此类中的唯一性。它是对整个类来说是唯一的，不属于某一个实例对象。
  - 对静态成员变量而言，成员函数不管是否为 static，在内存中都只有一个副本，普通成员函数调用时，需要传入 this 指针，static 成员函数调用时，无需 this 指针。

三种内存类型的生命期：

- 堆：由程序员自己分配和释放（malloc/free 或 new/delete），如果我们不手动释放，那就要到程序结束才释放。如果对分配的空间在不用的时候不释放而一味的分配，那么可能会引起内存泄漏。容量取决于虚拟内存，较大。
- 栈：生命期最短，到函数调用结束时。由编译器自动分配释放，容量较小。
  - 存放：主调函数中被调函数的下一句代码、函数参数和局部变量
- 静态存储区：生命期最长，延续到被程序员手动释放时（如果整个过程中都不手动释放，那就到程序结束时）。编译时由编译器分配，由系统释放。
  - 存放：全局变量、static 变量和常量

static 可以用来修饰变量，函数和类成员：

- 普通变量：被 static 修饰的变量就是静态变量，它会在程序运行过程中一直存在，会被放在静态存储区，main 函数运行前就分配了空间。有初值就用初值初始化，没有初值就用默认值初始化。局部静态变量的作用域在函数体中，全局静态变量的作用域在这个文件里。局部静态变量在函数调用结束后也不会被回收，会一直在程序内存中，直到该函数再次被调用，它的值还是保持上一次调用结束后的值。
- 普通函数：被 static 修饰的函数就是静态函数，静态函数只能在本文件中使用，不能被其他文件调用，也不会和其他文件中的同名函数冲突。
- 类中成员变量：被 static 修饰的成员变量是类静态成员，这个静态成员会被类的多个对象共用。

- 类中成员函数：被 static 修饰的成员函数也属于静态成员，不是属于某个对象的，访问这个静态函数不需要引用对象名，而是通过引用类名来访问。静态成员函数要访问非静态成员时，要通过对象来引用。

static 和 const 的区别：

- const 强调值不能被修改
- static 强调唯一的拷贝，对所有类的对象都共用

17. 计算下面几个类的大小

```
1  class A {};
2  int main(){
3    cout << sizeof(A) << endl; // 输出 1
4    A a;
5    cout << sizeof(a) << endl; // 输出 1
6    return 0;
7  }
```

- 空类的大小是 1， 在 C++ 中空类会占 1 个字节，这是为了让对象的实例能够相互区别。
  - 具体来说，空类同样可以被实例化，并且每个实例在内存中都有独一无二的地址。因此，编译器会给空类隐含加上 1 个字节，这样空类实例化之后就会拥有独一无二的内存地址。
  - 当该空白类作为基类时，该类的大小就优化为 0 了，子类的大小就是子类本身的大小。这就是所谓的空白基类最优化。
- 空类的实例大小就是类的大小，所以 sizeof(a)=1 字节，如果 a 是指针，则 sizeof(a) 就是指针的大小，也即 4 字节

```
1  class A { virtual Fun(){} };
2  int main(){
3    cout << sizeof(A) << endl; // 输出 4 (32 位机器) / 8 (64 位机器)
4    A a;
5    cout << sizeof(a) << endl; // 输出 4 (32 位机器) / 8 (64 位机器)
6    return 0;
7  }
```

- 因为有虚函数的类对象中都有一个虚函数表指针 __vptr，其大小是 4 字节

```
1  class A { static int a; };
2  int main(){
3    cout << sizeof(A) << endl; // 输出 1
4    A a;
5    cout << sizeof(a) << endl; // 输出 1
```

```
6     return 0;
7   }
```

- 静态成员存放在静态存储区，不占用类的大小，普通函数也不占用类大小

```
1   class A { int a; };
2   int main(){
3     cout << sizeof(A) << endl; // 输出 4
4     A a;
5     cout << sizeof(a) << endl; // 输出 4
6     return 0;
7   }
8
9   class A { static int a; int b; };;
10  int main(){
11    cout << sizeof(A) << endl; // 输出 4
12    A a;
13    cout << sizeof(a) << endl; // 输出 4
14    return 0;
15  }
```

- 静态成员 a 不占用类的大小，所以类的大小就是变量 b 的大小 即 4 个字节

18. struct 和 class 的区别

	struct	class
1	成员访问权限默认是 public	成员访问权限默认是 private
2	继承默认是 public 继承	继承默认是 private 继承
3		可以用作模板关键字（等同于 typename）

19. 【热门】C++ 的 STL 介绍（侯捷的籍和视频），包括内存管理 allocator，函数，实现机理，多线程实现等

- 《STL 源码剖析》“1.2 STL 六大组件 功能与运用”
- 《C++标准库》“第 6 章 标准模板库”
- STL 详解及常见面试题：https://blog.csdn.net/daaikuaichuan/article/details/80717222
- STL 简介和常见的面试题：https://blog.nowcoder.net/n/bdac96b3c0db437ca4646101f1c4255e?from=nowcoder_improve
- STL 总结与常见面试题+资料：https://zhuanlan.zhihu.com/p/147676383

STL 主要包括六大组件：容器（containers）、算法（algorithms）、迭代器（iterators）、仿函数（functors）、适配器（adapters）、空间配置器（allocators）。

		简要说明
1	容器	包含各种数据结构，用来管理某类对象的集合。主要分为三类： ① 序列式容器：有序（ordered）集合，array，vector，deque，list，forward_list

		② 关联式容器：已排序（sorted）集合，set，multiset，map，multimap ③ 无序容器：无序（unordered）集合，unordered_set，unordered_multiset，unordered_map，unordered_multimap
2	算法	用来处理容器中的元素，包含查找、排序、拷贝、修改、运算等算法。 并非容器类的成员函数，而是一种搭配迭代器使用的全局函数。 从实现角度来看，STL 算法是一种函数模板。
3	迭代器	用来遍历容器中的元素，是容器和算法间的胶合剂，是所谓的“泛型指针”。
4	仿函数	行为类似函数，可作为算法的某种策略。从实现来看，仿函数是一种重载了 operator() 的类或类模板。一般函数指针可视为狭义的仿函数。
5	适配器	用来修饰容器、仿函数或迭代器接口。比如 STL 中的 queue 和 stack 看似容器，实际上是适配器，底部都基于 deque。map 和 set 则基于 RB-Tree 适配器实现。
6	配置器	负责空间配置和管理。从实现来看，配置器是一个实现了动态空间配置、空间管理和空间释放的类模板。

各种容器的底层实现：（淡黄色为容器适配器）

- Containers library - cppreference.com：https://en.cppreference.com/w/cpp/container
- STL 常见面试题：https://www.jianshu.com/p/f921122ba125
- Reference - C++ Reference：https://www.cplusplus.com/reference/

类型	容器	头文件	底层实现	说明
序列式容器	array	<array>		
	vector	<vector>	动态数组，自动扩容	支持快速随机访问
	deque	<deque>	由一段一段的定量连续空间构成。采用一块 map（不是 map 容器）作为主控，其中每个元素（节点 node）都是指针，指向另一段（较大的）连续线性空间，称为缓冲区，它是 deque 的储存空间主体。	支持首尾（中间不能）快速增删，支持随机访问（但很复杂）
	list	<list>	双向链表	支持快速增删
	forward_list	<forward_list>	单向链表	
容器适配器	stack	<stack>	一般用 deque 或 list 实现	不允许遍历，没有迭代器
	queue	<queue>	一般用 deque 或 list 实现	不允许遍历，没有迭代器
	priority_queue	<queue>	一般用最大堆 max-heap 实现，实际上是一个 vector 表示的完全二叉树	不允许遍历，没有迭代器
关联式容器	set	<set>	用 RB-Tree 实现	有序，不重复
	multiset	<set>	用 RB-Tree 实现，插入操作是 insert_equal() 而非 insert_unique()	有序，可重复
	map	<map>	用 RB-Tree 实现	有序，不重复
	multimap	<map>	用 RB-Tree 实现，插入操作是 insert_equal() 而非 insert_unique()	有序，可重复
	unordered_set	<unordered_set>	用 hash table 实现	无序，不重复

类型	容器	头文件	底层实现	说明
无序容器	unordered_multiset	<unordered_set>	用 hash table 实现	无序，可重复
	unordered_map	<unordered_map>	用 hash table 实现	无序，不重复
	unordered_multimap	<unordered_map>	用 hash table 实现	无序，可重复

20. STL 中 vector 的实现

- 《C++标准库》“第 7 章 STL 容器”“7.1 容器的共通能力和共通操作”和“7.3 Vector”
- 《STL 源码剖析》“第 4 章 序列式容器”“4.2 vector”：第 148 ~ 161 页
- 面试：vector 类的简单实现：https://www.cnblogs.com/wxquare/p/4986552.html
- C++ vector（STL vector）底层实现机制（通俗易懂）：http://c.biancheng.net/view/6901.html
- 【决战西二旗】|理解 STL vector 原理：https://zhuanlan.zhihu.com/p/94087639
- c++11 初始化列表（initializer_list）：https://blog.csdn.net/li1615882553/article/details/86529889

vector 是一种序列式容器（Sequence Containers）。

在实现 vector 之前，首先简述容器的共通能力和共通操作，大部分都是必要条件。了解这些能力和操作，在设计时就知道大致的方向和思路。

容器三个最核心的能力：

1. 所有容器提供的都是“value 语义”而非“reference 语义”。容器进行元素的安插动作时，内部实施的都是 copy 或 move 动作，而不是管理元素的 reference。
2. 元素在容器内有特定顺序。容器中的迭代器可以遍历元素，多次迭代返回的元素次序相同，无序容器也是如此。这一能力的前提是没有对元素进行增删或修改。
3. 各项操作并非绝对安全，并不会检查每个可能发生的错误。调用者必须保证传入的实参合法，否则可能会产生不明确的行为。

容器的共通操作：

1. 初始化：默认构造函数、拷贝构造函数、析构函数
2. 赋值和 swap()
3. 与大小相关的操作函数：empty()、size()、max_size()
4. 比较
5. 元素访问：迭代器接口，begin()、end()、……

下面开始分析 vector 的实现。需要抓住如下几个关键点，不要陷入其他细枝末节：

- 本质上是一个动态增长的数组，包含一个指针指向一片连续的内存空间
- 当空间装不下的时候会自动申请一片更大的空间（空间配置器）
- 删除元素时空间并不会释放，只是清空了其中的数据

考察的知识点主要是：动态扩容。

- new：申请更大的内存空间
- memcpy：将旧的内存空间中的数据，按原有顺序移动到新的内存空间中

- delete：将旧的内存空间释放

实现 vector 的代码如下，一些注意点：

成员变量和函数	说明
INIT_CAPACITY，_size，_cap，*_arr	_size 表示当前元素个数，_cap 表示最大容量，*_arr 表示实际的数组，INIT_CAPACITY 表示初始容量 size_t 包含在头文件 <cstddef> 中
my_vector()	默认构造函数
my_vector(size_t, T)	带参数的构造函数 注意先检查容量是否足够，再 new 一个新数组赋值。
my_vector(const my_vector<T>&)	拷贝构造函数 注意 new 的大小应该是传入数组的_cap 而不是_size，同时学会 memcpy(dest, src, size) 的写法 memcpy 包含在头文件 <cstring> 中
my_vector<T> & operator=(const my_vector &)	赋值构造函数 注意自赋值，直接返回*this；若已有_arr，需 delete [] 然后同上面的拷贝构造函数，注意最后还要返回*this
my_vector(std::initializer_list<T>)	列表初始化构造函数 注意初始_cap 赋值，并且若容量不够还要继续扩容 new 一个新_arr，将传入数组的值依次放入 initializer_list 包含在头文件 <initializer_list> 中
~my_vector()	析构函数 若_arr 存在，则 delete []
T operator[](size_t) const	索引函数 注意使用 assert(idx < _size) 断言保证不越界 assert 包含在头文件 <cassert> 中
size_t size() const	当前元素个数
size_t capacity() const	当前最大容量
bool empty()	是否为空
void push_back(const T &)	在尾部添加元素 注意如果容量不够，需要先用 tmp 保存旧的_arr，然后 new 一个新的容量为新的_cap 的_arr，再用 memcpy 复制元素，最后 delete [] tmp 在_size 处放入 x 并将_size 加 1
void pop_back()	删除尾部元素 注意使用 assert(_size>0) 保证数组非空，再--_size
void insert(size_t, const T &)	在 pos 索引处插入元素 注意使用 assert(pos>=0 && pos<=_size) 确保插入位置合法。若容量不够，需要扩容，同 push_back。 将_size 加 1，然后从_size-1 到 pos+1 从后向前执行 _arr[i]=_arr[i-1]，最后_arr[pos] 放置传入元素
void erase(size_t)	在 pos 索引处删除元素

成员变量和函数	说明
	注意使用 assert(pos>=0 && pos<_size) 确保删除位置合法。然后从 pos 到_size-2 从前向后执行 _arr[i]=_arr[i+1]，最后将_size 减 1
std::ostream & operator<<(std::ostream &)	输出到标准输出流 注意返回 os。

vector 的实现

```
#include <iostream>
#include <cstddef> // size_t
#include <vector>
#include <cstring> // memcpy
#include <cassert> // assert
#include <initializer_list> // initializer_list

template <typename T>
class my_vector {
private:
    size_t INIT_CAPICITY = 16;
    size_t _size;
    size_t _cap;
    T* _arr;

public:
    // default constructor
    my_vector() : _arr(nullptr), _size(0), _cap(INIT_CAPICITY) {}

    // parameter constructor
    my_vector(size_t n, T x = 0) {
        _cap = INIT_CAPICITY;
        while (_cap < n)
            _cap *= 2;
        _arr = new T[_cap];
        _size = n;
        while (n--)
            _arr[n] = x;
    }

    // copy constructor
    my_vector(const my_vector<T> & v) {
        _size = v._size;
        _cap = v._cap;
        _arr = new T[v._cap];
```

vector 的实现

```
36          memcpy(_arr, v._arr, _cap * sizeof(T));
37      }
38
39      // assignment constructor
40      my_vector<T> & operator=(const my_vector& v) {
41          if (this == &v)
42              return *this;
43          if (_arr)
44              delete [] _arr;
45          _size = v._size;
46          _cap = v._cap;
47          _arr = new T[v._cap];
48          memcpy(_arr, v._arr, _cap*sizeof(T));
49          return *this;
50      }
51
52      // list constructor
53      my_vector(std::initializer_list<T> v) {
54          _cap = INIT_CAPICITY;
55          _size = v.size();
56          while(_cap < _size)
57              _cap *= 2;
58          _arr = new T[_cap];
59
60          size_t idx = 0;
61          for (auto const &x : v)
62              _arr[idx++] = x;
63      }
64
65      // destructor
66      ~my_vector() {
67          if (_arr)
68              delete [] _arr;
69      }
70
71      // [] index
72      T operator[](size_t idx) const {
73          assert(idx < _size);
74          return _arr[idx];
75      }
76
77      // size()
78      size_t size() const {
```

vector 的实现

```
79          return _size;
80      }
81
82      // capacity()
83      size_t capacity() const {
84          return _cap;
85      }
86
87      // empty()
88      bool empty() {
89          return _size==0;
90      }
91
92      // push_back()
93      void push_back(const T & x) {
94          if (_size == _cap) {
95              _cap *= 2;
96              T* tmp = _arr;
97              _arr = new T[_cap];
98              memcpy(_arr, tmp, _size*sizeof(T));
99              delete [] tmp;
100         }
101         _arr[_size++] = x;
102     }
103
104     // pop_back()
105     void pop_back() {
106         assert(_size > 0);
107         --_size;
108     }
109
110     void insert(size_t pos, const T & x) {
111         assert(pos>=0 && pos<=_size);
112         if (_size == _cap) {
113             _cap *= 2;
114             T * tmp = _arr;
115             _arr = new T[_cap];
116             memcpy(_arr, tmp, _size*sizeof(T));
117             delete [] tmp;
118         }
119         ++_size;
120         for (size_t i=_size-1; i>pos; --i)
121             _arr[i] = _arr[i-1];
```

vector 的实现

```
122         _arr[pos] = x;
123     }
124 
125     void erase(size_t pos) {
126         assert(pos>=0 && pos <_size);
127         for (size_t i=pos; i<_size-1; ++i)
128             _arr[i] = _arr[i+1];
129         --_size;
130     }
131 
132     // operator <<
133     std::ostream & operator<<(std::ostream &os) {
134         for (size_t i=0; i<_size; ++i)
135             os << _arr[i] << " ";
136         os << std::endl;
137         return os;
138     }
139 };
140 
141 int main() {
142     my_vector<int> v1 = my_vector<int>(5,0);
143     my_vector<int> v2 = {1,2,3,4,5};
144     v1 << std::cout; // 0 0 0 0 0
145     std::cout << v2[2] << " "
146               <<  v2.size() << " "
147               <<  v2.capacity() << " "
148               << std::endl; // 3 5 16
149     v2 << std::cout; // 1 2 3 4 5
150     v2.insert(2,0);
151     v2 << std::cout; // 1 2 0 3 4 5
152     v2.erase(4);
153     v2 << std::cout; // 1 2 0 3 5
154     v1.push_back(2);
155     v1 << std::cout; // 0 0 0 0 0 2
156     while (v2.size()) {
157         v2.pop_back();
158         v2 << std::cout;
159     }
160     std::vector<int> v3 = {5,6,7};
161     v3.insert(v3.end(), 1); // 5 6 7 1
162     for (auto const & x: v3)
163         std::cout << x << " ";
164     return 0;
```

vector 的实现
165 } 166

21. string 的实现

22. vector 使用的注意点及其原因，频繁对 vector 调用 push_back() 对性能的影响和原因

注意点：不要频繁插入或删除元素

原因：

- 插入操作：vector 在容器大小不够时，重新申请一块两倍大小的空间，并将原容器的元素拷贝到新容器中，并释放原空间，这个过程耗费大量时间和空间
- 删除操作：需要复制元素

解决方案：

- 事先指定 vector 的大小
- 考虑使用 list

23. vector 中 emplace_back 和 push_back 的差别

24. vector 会迭代器失效吗？什么情况下会迭代器失效？

- 迭代器失效的几种情况总结：https://blog.csdn.net/lujiandong1/article/details/49872763

迭代器失效分三种情况（数据结构）考虑：数组型，链表型，树型。

1. 数组型（vector）：分配在连续的内存中。① insert 和 erase 操作都会使得删除点和插入点之后的元素挪位置，所以插入点和删除掉之后的迭代器全部失效。也就是说，insert(*iter) 或 erase(*iter)，然后再 iter++，是没有意义的。② 此外，如果原有的空间不够，会进行扩容，并将原来的元素移动到新的内存位置，则原有的所有迭代器都失效。
   - 解决方法：erase(*iter) 的返回值是下一个有效迭代器的值，iter = v.erase(iter)
2. 链表型（list）：使用了不连续分配的内存。插入不会使得任何迭代器失效；删除运算使指向删除位置的迭代器失效，但是不会失效其他迭代器。
   - 解决办法有两种：erase(*iter) 会返回下一个有效迭代器的值；erase(iter++)
3. 树形（map）： 使用红黑树来存储数据。插入不会使得任何迭代器失效；删除运算使指向删除位置的迭代器失效，但是不会失效其他迭代器。
   - erase 迭代器返回值为 void，所以要采用 erase(iter++) 的方式删除迭代器。

注意：erase(iter) 后迭代器 iter 完全失效，iter 不能参与任何运算，包括 iter++和*iter

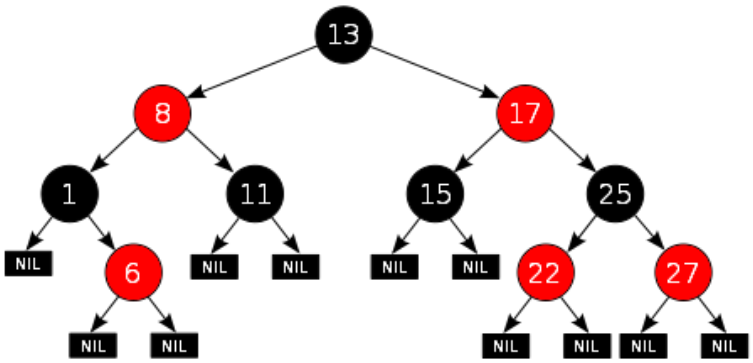
25. STL 中 vector、list 和 deque 的区别

26. 红黑树详解

- 《算法导论》“第 13 章 红黑树”
- 硬核图解面试最怕的红黑树【建议反复摩擦】：https://blog.csdn.net/qq_35190492/article/details/109503539
- 带着面试题学习红黑树操作原理：https://xie.infoq.cn/article/d6c0c97f99262ca1bf9d2e41a
- 轻松搞定面试中的红黑树问题：https://blog.csdn.net/silangquan/article/details/18655795
- 红黑树(一)之原理和算法详细介绍：https://www.cnblogs.com/skywang12345/p/3245399.html
- 30 张图带你彻底理解红黑树：https://www.jianshu.com/p/e136ec79235c
- 我画了 20 张图，给女朋友讲清楚红黑树：https://www.cxyxiaowu.com/7374.html
- 图解红黑树：https://www.cnblogs.com/ahuntsun-blog/p/12458115.html
- Red/Black Tree Visualization：https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

不要死记插入和删除的各种情形和操作，既不现实也不明智，重要的是理解为什么这样做。

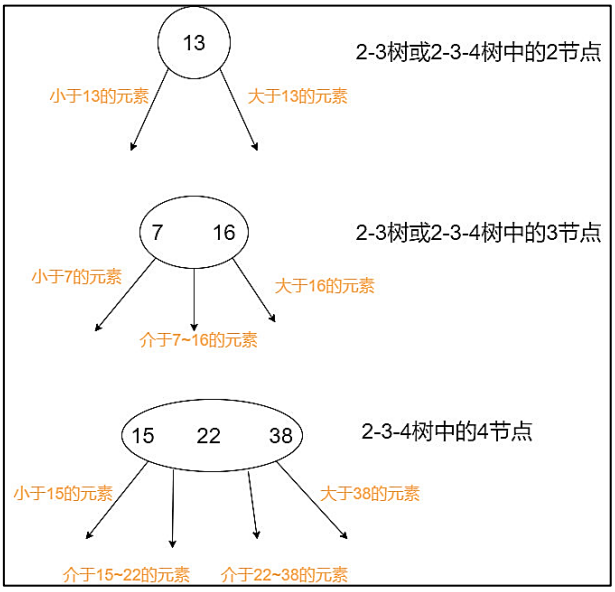
5 条性质



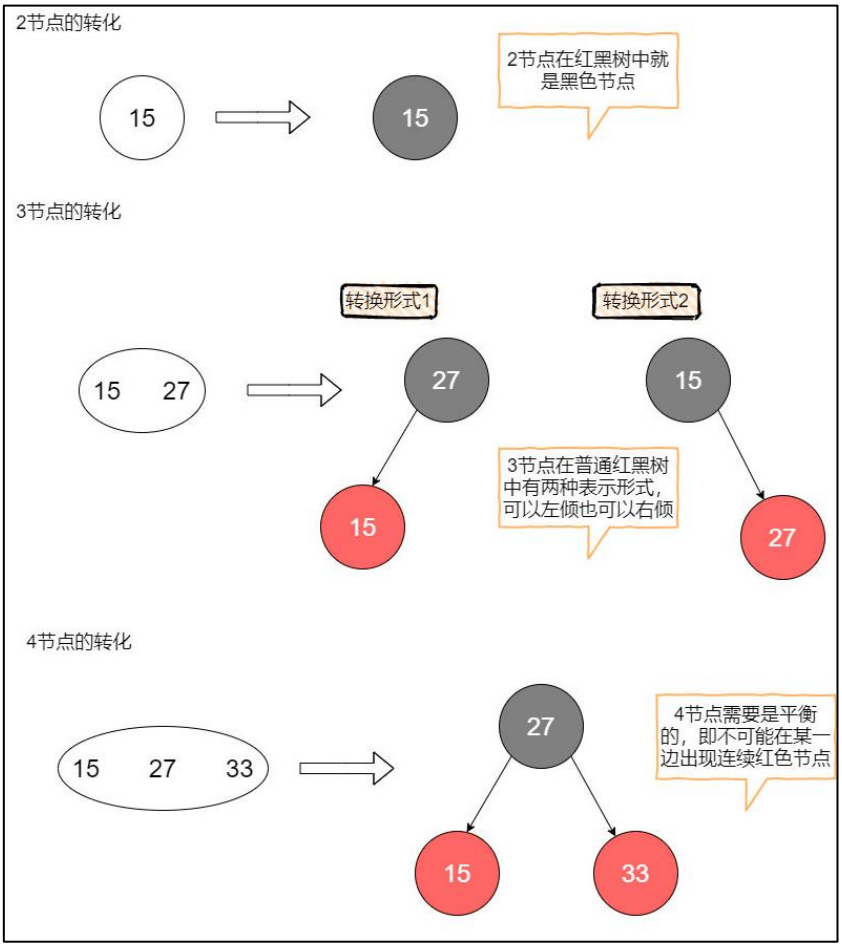
红黑树是一个二叉搜索树，具有如下 5 条性质：

1. 每个节点要么是**红色**的，要么是**黑色**的
2. 根节点是**黑色**的
3. 每个叶节点（NIL）是**黑色**的
4. 如果一个节点是**红色**的，那么它的两个子节点都是**黑色**的
5. 任一节点到其叶节点的路径上，都包含数量相同的**黑色**节点（黑色完美平衡）
   - 如果一个节点存在**黑色**的子节点，那么该节点肯定有两个子节点

从 2-3-4 树到红黑树



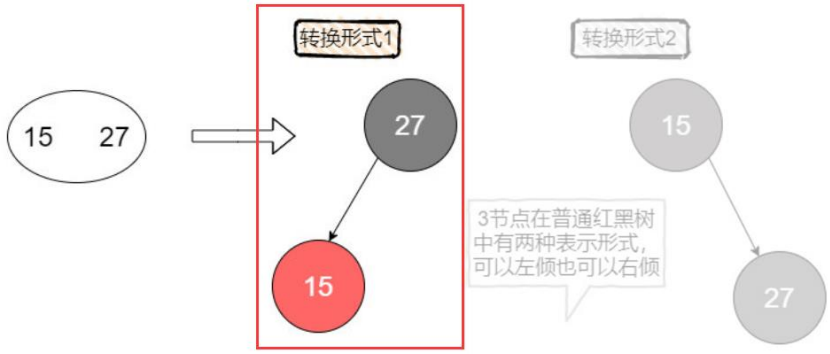
2-3-4 树是阶数为 4 的 B 树



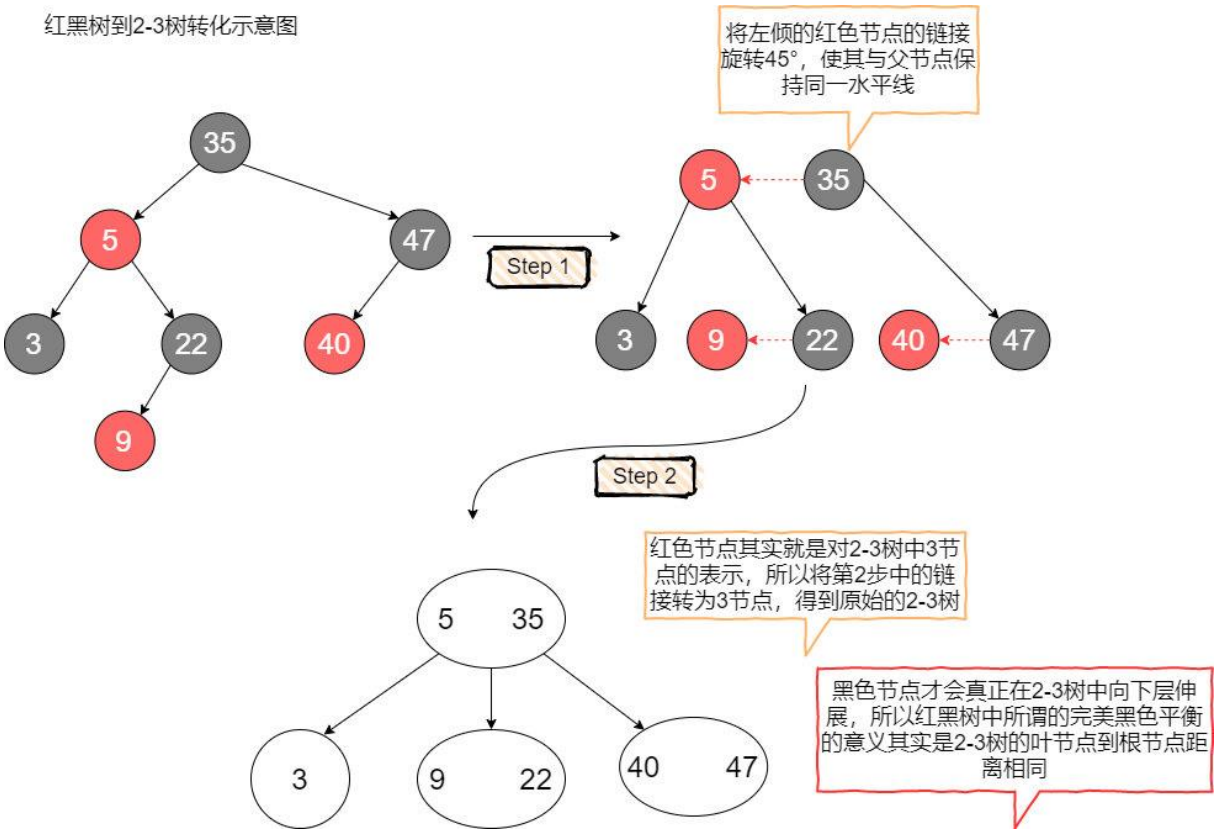
2-3-4 树到红黑树的转化

红黑树事实上就是 2-3 树，且是左倾形式的红黑树，左倾产生了如下限制：

- 如果树中出现红色节点，那么必然是左子节点。



红黑树实际上就是左倾 2-3 树



红黑树到 2-3 树的转化

- 《算法导论》中给出的是红黑树基于 2-3-4 树实现，其中 4 节点要求平衡（即 4 节点必须用黑色父亲和左右两个红色儿子表示，红色儿子不能出现在同一边）。
- 《算法 4》中给出的红黑树基于 2-3 树实现，而且这种实现的红黑树十分特殊，它要求 3 节点在红黑树中必须用左倾的红色节点来表示。这种限定能够极大减少红黑树调整过程中的复杂性，我们将在接下来的内容中体会到这一点。

这里以《算法 4》中基于 2-3 树概念模型的左倾红黑树为例，原因如下：

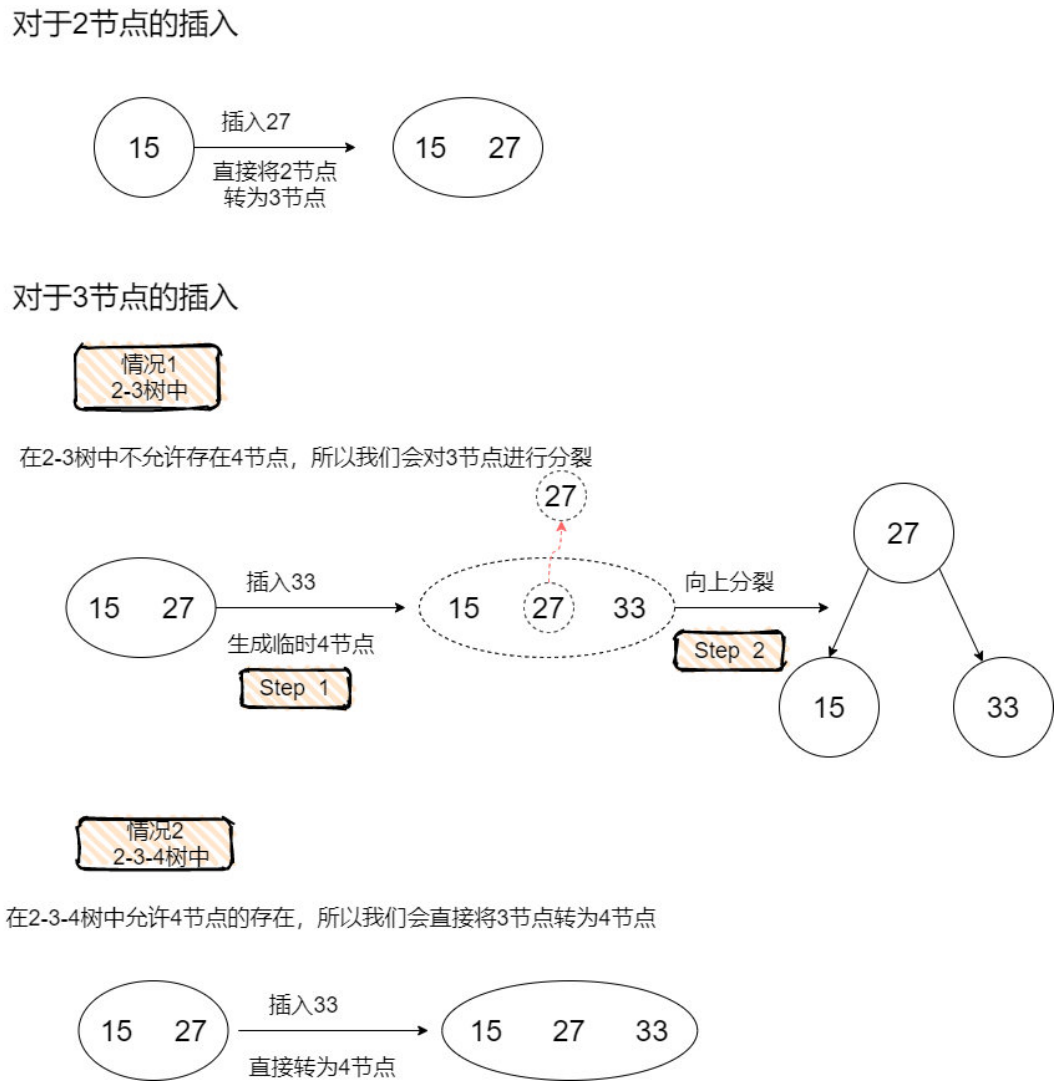
- 不用考虑 2-3-4 树中复杂的 4 节点分裂
- 左倾红黑树进一步降低了调平的难度
- 《算法导论》中对于红黑树删除场景的阐述并不够具体，许多关键环节都用“经过一定的旋转和变色处理”来带过，不利于新手学习。

2-3 树的插入

在了解红黑树的插入删除操作之前，需要先了解 2-3 树的插入删除操作，这样才能理解红黑树中染色和旋转背后的意义。

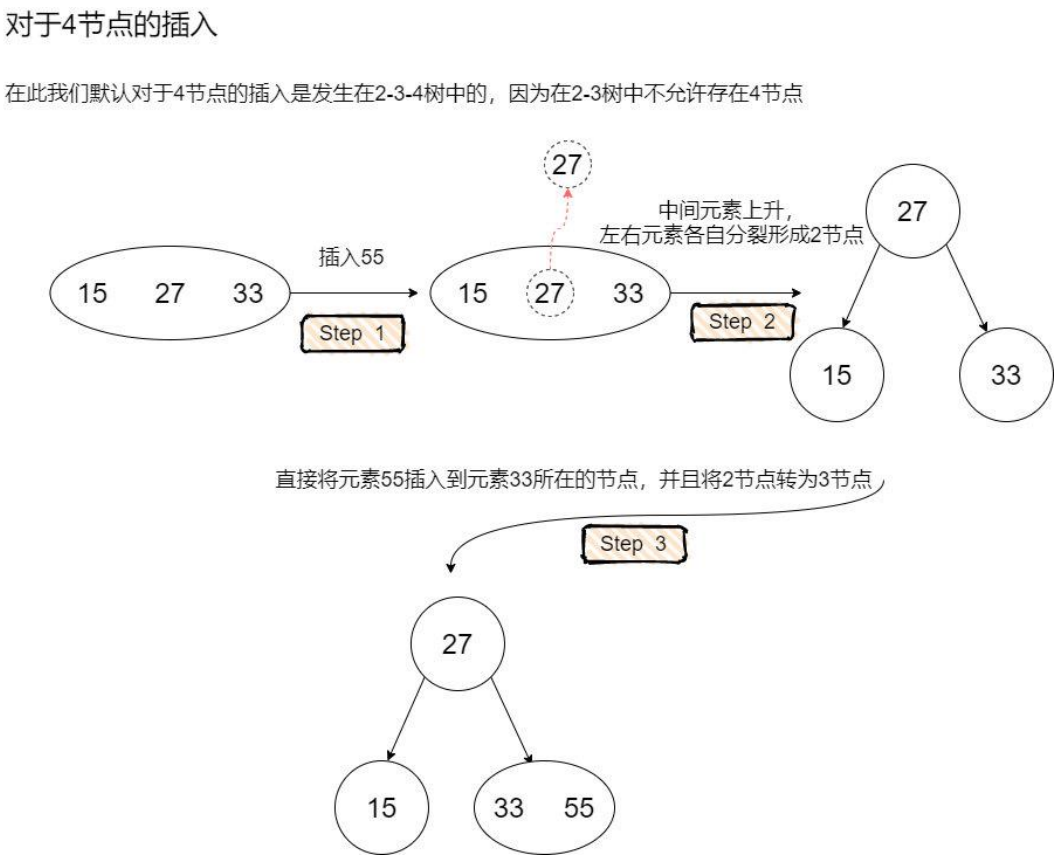
让我们来看一下对于 2-3 树的插入。我们的插入操作需要遵循一个原则：

- 先将这个元素尝试性地放在已经存在的节点中
  - 如果要存放的节点是 2 节点，那么插入后会变成 3 节点
  - 如果要存放的节点是 3 节点，那么插入后会变成 4 节点（临时），然后对可能生成的临时 4 节点进行分裂处理，使得临时 4 节点消失



2-3 树的插入（2 节点和 3 节点）

在 2-3-4 树中向 4 节点插入：



2-3-4 树中向 4 节点插入

事实上，这正对应了红黑树在**插入**的时候一定会把**待插入节点**涂成**红色**，因为红色节点的意义是**与父节点进行关联**，形成概念模型 2-3 树中的 3 节点或者临时 4 节点。

而红黑树之所以需要在插入后进行调整，正是因为可能存在着概念模型中的临时 4 节点（反应在红黑树中是双红的情况）。

试想在 2-3 树中如果待插入节点是个 2 节点，那么反映在红黑树中，不正好对应着黑色父节点吗？而在黑色父节点下面增加一个红色儿子，确实不会违背红黑树的任何规则，这也对应着我们向 2-3 树中的 2 节点插入一个元素，只需要简单的把 2 节点变成 3 节点。

2-3 树的删除

接下来让我们来看一下对于 2-3 树的删除。对于 2-3 树的删除我们主要要考虑待删除元素在 2 节点这种情况，因为如果待删除元素在 3 节点，那么可以直接将这个元素删除，而不会破坏 2-3 树的任何性质（删除这个元素不会引起高度的变化）。

当待删除元素在 2 节点的时候，由于删除这个元素会导致 2 节点失去自己唯一的元素，引发 2 节点自身的删除，会使得树中某条路径的高度发生变化，树变得不平衡。

因此我们有两种方案去解决这个问题：

第一种方案，先删除这个 2 节点，然后对树进行平衡调整。

第二种方案，我们想办法让这个被删除的元素不可能出现在 2 节点中。

本文选择第二种方案，我们在搜索到这个节点的路径中，不断地判断当前节点是否为 2 节点，如果是，就从它的兄弟节点或者它的父节点借一个元素，使得当前节点由 2 节点成为一个 3 节点或者一个临时 4 节点（视具体情况而定，在后面的红黑树部分会详细介绍）。

这种操作会产生一种结果：除非当前节点是根节点，否则当前节点的父节点一定是一个非 2 节点（因为搜索的路径是自上而下，父节点已经进行过了这种操作，所以不可能是 2 节点），那么我们可以保证到达叶子节点的时候，也能顺利的从父节点或者兄弟节点处借到元素，使得自己成为非 2 节点。从而能够直接删除某个元素（现在这个元素不在 2 节点中了）。

红黑树的旋转

红黑树能自平衡主要靠两种操作：旋转和变色。

- **旋转**：分为左旋和右旋。
  - 左旋：以下图 1 为例，对 A 进行左旋。
    - ① 其右子节点 C 替换到 A 的当前位置，并将 A 作为它的左子节点；
    - ② C 的左子节点 D 变成 A 的右子节点
  - 右旋：以下图 2 为例，对 A 进行右旋
    - ① 其左子节点 B 替换到 A 的当前位置，并将 A 作为它的右子节点
    - ② B 的右子节点 E 变成 A 的左子节点
- **变色**：红→黑或者黑→红

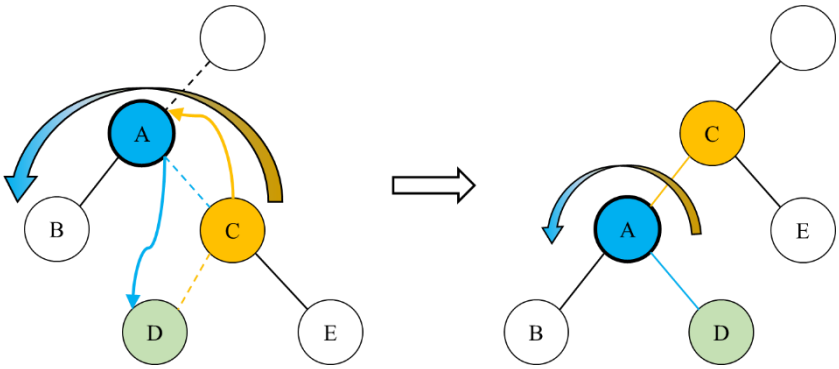


图 1：对节点 A 进行左旋

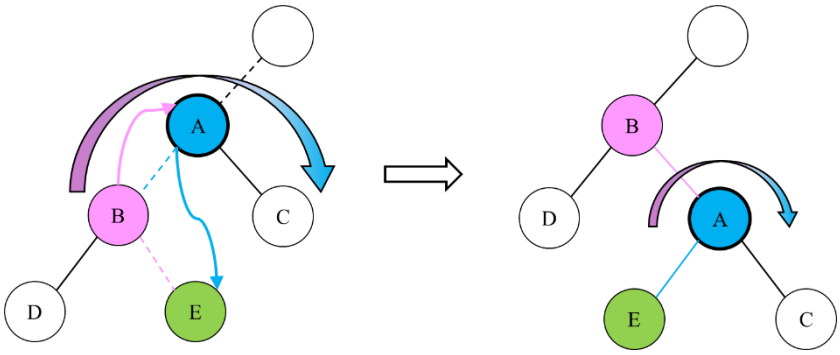


图 2：对节点 A 进行右旋

旋转操作产生的影响是：

- 不会影响旋转节点的父节点，父节点以上的结构还是不变。
- 左旋只影响旋转节点及其右子树，不影响左子树
- 右旋只影响旋转节点及其左子树，不影响右子树
- 往哪边旋，就相当于将树中的节点往哪边多分配一点

红黑树的查找

很简单，和二叉搜索树相同。左子树的节点都小于当前节点，右子树的节点都大于当前节点。

红黑树的插入

插入操作有两个步骤：

- **查找插入位置：**过程和上面的查找相同。
  - 插入的节点颜色默认为红色，因为如果用黑色，就会破坏性质 5，需要做自平衡。
- **插入后自平衡**

下面讨论不同的插入情形应当采取的操作

	插入情形	操作
1	空树	将插入节点作为根节点，并设为**黑色**
2	插入节点的 key 已经存在	
3	插入节点的父节点为**黑色**	
4	插入节点的父节点为红色	
5		

红黑树的删除

27. 跳表

28. STL 中 map 的实现

29. STL 中 hash 表的实现

30. STL 中 set 和 map 的联系与区别

31. STL 中 unordered_map 和 map 的区别

32. 解决哈希冲突的方式

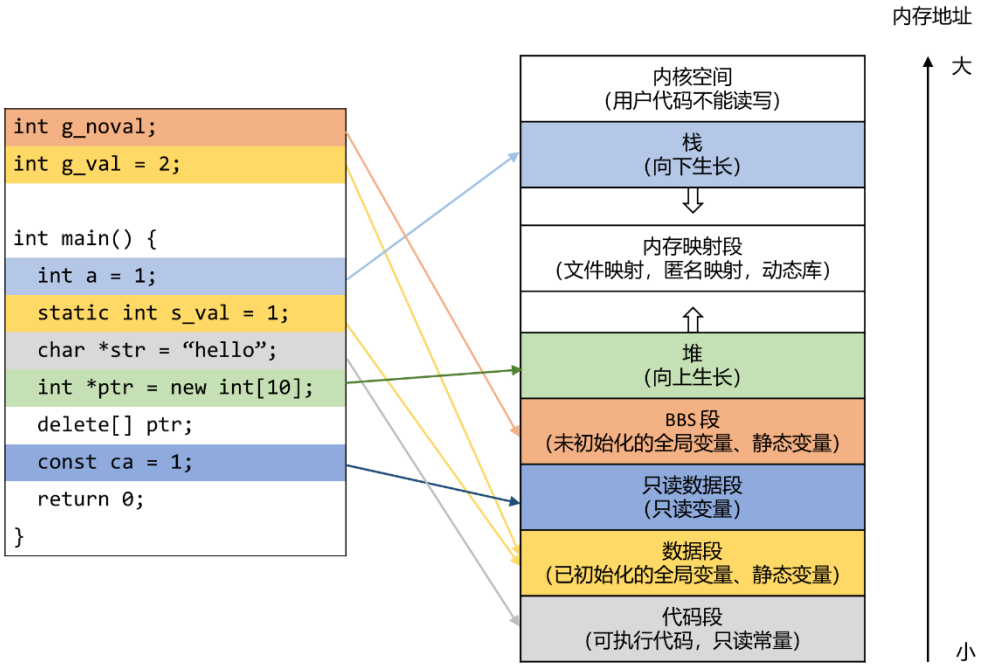
33. 返回值优化

34. set、map 和 vector 的插入复杂度

35. C++程序的编译过程

36. C++ 中的重载和重写的区别

37. 【热门】C ++ 内存管理，内存分配原理



C/C++内存布局

38. 介绍面向对象的三大特性，并且举例说明每一个

39. 多态的实现（和下个问题一起回答）

40. 【热门】C++ 虚函数相关（虚函数表，虚函数指针），虚函数的实现原理

41. 实现编译器处理虚函数表应该如何处理

42. 基类的析构函数一般写成虚函数的原因

43. 构造函数为什么一般不定义为虚函数

44. 构造函数或者析构函数中调用虚函数会怎样

45. 纯虚函数

46. 静态绑定和动态绑定的介绍

47. 深拷贝和浅拷贝的区别（举例说明深拷贝的安全性）

48. 对象复用的了解，零拷贝的了解

49. 介绍 C++ 所有的构造函数

50. 什么情况下会调用拷贝构造函数（三种情况）

51. 结构体内存对齐方式和为什么要进行内存对齐？

52. 内存泄露的定义，如何检测与避免？

53. C++的智能指针有哪些

54. 调试程序的方法

55. 遇到 coredump 要怎么调试

56. inline 关键字说一下，和宏定义有什么区别

57. 模板的用法与适用场景，实现原理

58. 成员初始化列表的概念，为什么用成员初始化列表会快一些（性能优势）

59. 将 pair 和 vector 作为 unordered_map 的 key

- pair 作为 unordered_map unordered_set 的键值 C++：https://youngforest.github.io/2020/05/27/unordered-map-hash-pair-c/
- c++ - Why can't I compile an unordered_map with a pair as key?：https://stackoverflow.com/questions/32685540/why-cant-i-compile-an-unordered-map-with-a-pair-as-key
- unordered_map：http://www.cplusplus.com/reference/unordered_map/unordered_map/
- How to use std::pair as key to std::unordered_map in C++：https://www.techiedelight.com/use-std-pair-key-std-unordered_map-cpp/
- hash - C++ Reference：http://www.cplusplus.com/reference/functional/hash/

C++ 代码实现

1. 快速选择

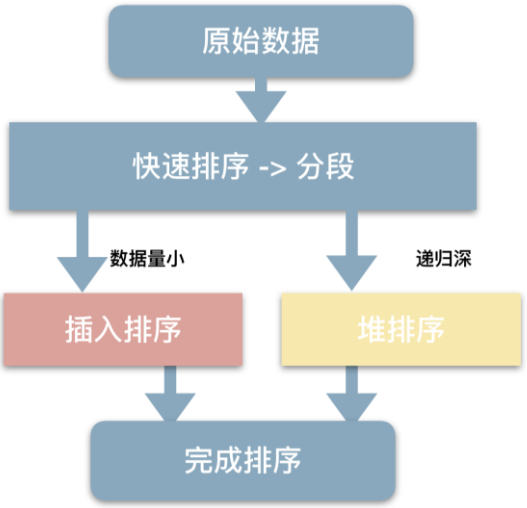
参见“高频代码题解”“0215 数组中的第 K 个最大元素”“关键思路 1：快速选择”。

2. STL 中 sort 的实现

- 内省排序：https://zh.wikipedia.org/wiki/%E5%86%85%E7%9C%81%E6%8E%92%E5%BA%8F
- 谈谈 STL 中的 std::sort：https://liam.page/2018/09/18/std-sort-in-STL/
- 知无涯之 std::sort 源码剖析：https://feihu.me/blog/2014/sgi-std-sort/#stdsort%E7%9A%84%E5%AE%9E%E7%8E%B0
- C++一道深坑面试题：STL 里 sort 算法用的是什么排序算法？：https://zhuanlan.zhihu.com/p/36274119

STL 中的 sort 算法综合了快速排序、插入排序和堆排序

- 数据量大时采用快速排序（quick sort），分段归并。
- 一旦分段后的数据量小于某个门槛（16），为避免快排的递归调用带来过大的额外负荷，就改用插入排序（insertion sort）。
- 如果递归层次过深，还会改用堆排序（heap sort）。



内省排序（Introsort），由 David Musser 在 1997 年设计。内省排序算法首先从快速排序开始，当递归深度超过一定深度（深度为排序元素数量的对数值）后转为堆排序。

采用这个方法，内省排序既能在常规数据集上实现快速排序的高性能，又能在最坏情况下仍保持 O(nlogn) 的时间复杂度。由于这两种算法都属于比较排序算法，所以内省排序也是一个比较排序算法。

在快速排序算法中，一个关键操作就是选择基准点（Pivot）：元素将被此基准点分开成两部分。最简单的基准点击择算法是使用第一个或者最后一个元素，但这在排列已部分有序的序列上性能很糟。Niklaus Wirth 为此设计了一个快速排序的变体，使用处于中间的元素来防止在某些特定序列上性能退化为 O(n^2)的状况。这个 3 基准中位数选择算法从序列的第一，中间和最后一个元素获取中位数来作为基准，虽然这个算法在现实世界的数据上性能表现良好，但经过精心设计的“破解序列”仍能大幅降低此变体算法的性能。这样就有攻击者精心设计序列发送到因特网服务器以进行拒绝服务（DoS）攻击的潜在可能性。

3. 快速排序

- 官方题解 - 排序数组：https://leetcode-cn.com/problems/sort-an-array/solution/pai-xu-shu-zu-by-leetcode-solution/
- 【动画模拟】10000 字快排？：https://leetcode-cn.com/problems/sort-an-array/solution/dong-hua-mo-ni-yi-ge-kuai-su-pai-xu-wo-x-7n7g/
- 快速排序：https://zh.wikipedia.org/wiki/%E5%BF%AB%E9%80%9F%E6%8E%92%E5%BA%8F
- 当我谈排序时，我在谈些什么🤔：https://leetcode-cn.com/problems/sort-an-array/solution/dang-wo-tan-pai-xu-shi-wo-zai-tan-xie-shi-yao-by-s/
- 知无涯之 std::sort 源码剖析：https://feihu.me/blog/2014/sgi-std-sort/#stdsort%E7%9A%84%E5%AE%9E%E7%8E%B0
- 十二种排序算法包你满意（带 GIF 图解）：https://leetcode-cn.com/problems/sort-an-array/solution/shi-er-chong-pai-xu-suan-fa-bao-ni-man-yi-dai-gift/

cs-notes/_tests/quick_sort.cpp

```
1  #include "_utils.h"
2
3  #include <iostream>
4  #include <vector>
```

cs-notes/_tests/quick_sort.cpp

```cpp
#include <cstdlib>
#include <algorithm>

using namespace std;

int rand_partition(vector<int> &a, int L, int R) {
    int pivot = rand() % (R-L+1) + L;
    swap(a[pivot], a[R]);
    int last = a[R];
    int i=L;
    for (int j=L; j<R; ++j) {
        if (a[j] < last)
            swap(a[j], a[i++]);
    }
    swap(a[i], a[R]);
    return i;
}

void _quick_sort(vector<int> & a, int L, int R) {
    if (L<R) {
        int q = rand_partition(a, L, R);
        _quick_sort(a, L, q-1);
        _quick_sort(a, q+1, R);
    }
}

void quick_sort(vector<int> & a) {
    _quick_sort(a, 0, a.size()-1);
}

int main() {
    vector<int> nums = {2,3,5,6,4,7,1,0};
    disp(nums);
    quick_sort(nums);
    disp(nums);

    return 0;
}

```

4. STL 中 stable_sort 的实现

5. 堆的实现（优先队列）

参见“高频代码题解”“0215 数组中的第 K 个最大元素”“关键思路 2：堆”。

6. 堆排序

- 排序算法的 c++实现——堆排序：https://www.cnblogs.com/yinheyi/p/10836167.html
- 排序数组：https://leetcode-cn.com/problems/sort-an-array/solution/pai-xu-shu-zu-by-leetcode-solution/
- 数据结构 - 堆排序(heap sort) 详解及代码(C++)：https://blog.csdn.net/caroline_wendy/article/details/31357053
- 建堆算法：https://blog.csdn.net/lz233333/article/details/61629935

对一个数组进行堆排序，分为两步：① 建堆；② 排序。

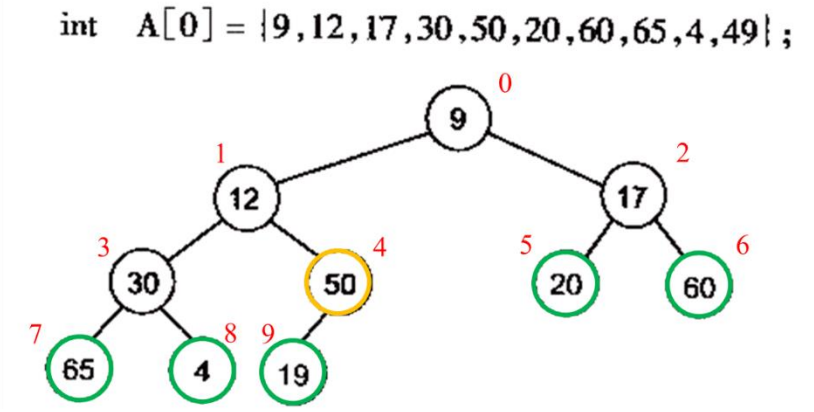
堆是通常用数组表示，是一个完全二叉树。要将一个数组建成堆，需要对其进行调整。

假如数组的索引从 0 开始，那么对于索引为 i 的节点：

- 其左子节点索引为 2*i+1，右子节点索引为 2*i+2
- 其父节点索引为 $\lfloor (i-1)/2 \rfloor$

叶子节点视为已经调整好的堆节点，因此只需要从第一个非叶节点开始调整即可。

堆上第一个非叶节点的索引为 $\lfloor (len-1)/2 \rfloor$（可以理解成最后一个叶节点的父节点），也即下图中索引为 4 的节点 50。



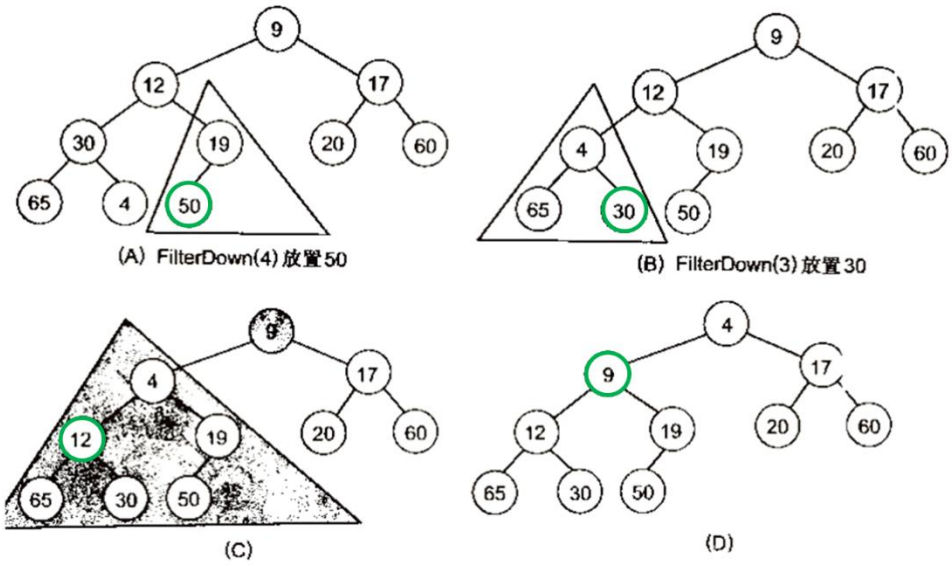
叶子节点视为已经调整好的堆节点，故从第一个非叶节点 50 开始调整

下图展示了一个小根堆的调整流程，大根堆类似。从索引 4 开始，一直调整到索引 0。

这里的调整就是下沉操作，在小（大）根堆中，将当前被调整节点与左右子节点里更小（大）的那个交换。

- 图 a 调整索引 4 的 50，将其与左子节点 19 交换
- 图 b 调整索引 3 的 30，将其与右子节点 4 交换
- 索引 2 的 17 不需要调整
- 图 c 调整索引 1 的 12，将其与左子节点 4 交换，此时 12 的两个左右子节点 65 和 30 都比它大，故不需要再下沉

- 图 d 调整索引 0 的 9，将其与左子节点 4 交换，此时 9 的两个左右子节点 12 和 19 都比它大，故不需要再下沉，此时堆已调整完成



接下来就很简单了。注意剩下来对堆进行排序的操作实际上是破坏堆的，只是在每次排序中，[0,i-1] 范围的元素依旧是满足堆的性质的。

- 只要依次将堆顶的元素 nums[0] 放到堆底 nums[i]，i 从 len-1 到 1，这样每次堆底就是排序好的了，因为小（根）堆的堆顶一定是堆中最小（大）的元素
- 然后再调整此时的堆顶元素到合适位置，注意此时需要控制堆数组的右边界为 i-1，因为最右边的 nums[i] 已经是排序好的了，不能再动

cs-notes/_tests/heap_sort.cpp

```
1   #include "_utils.h"
2
3   #include <iostream>
4   #include <vector>
5   #include <functional>
6
7   using namespace std;
8
9   bool cmp(int a, int b) {
10      return a < b;
11  }
12
13  void heapify(vector<int> &nums, int i, int r_bound, bool (*cmp)(int, int)) {
14      while (i*2+1 <= r_bound) { // left child is not null
15          int L = i*2 + 1;
16          int R = i*2 + 2;
17          int greater_idx;
18
19          // get greater child idx;
```

cs-notes/_tests/heap_sort.cpp

```
20          if (L<=r_bound && cmp(nums[i], nums[L]))
21              greater_idx = L;
22          else
23              greater_idx = i;
24          if (R<=r_bound && cmp(nums[greater_idx], nums[R]))
25              greater_idx = R;
26
27          // if greater than both child, break
28          // else swap to greater child, and adjust the new child
29          if (greater_idx == i)
30              break;
31          else {
32              swap(nums[greater_idx], nums[i]);
33              i = greater_idx;
34          }
35      }
36  }
37
38  void build_heap(vector<int> &nums, bool (*cmp)(int, int)) {
39      int len = nums.size();
40      for (int i=(len-1)/2; i>=0; --i) {
41          heapify(nums, i, len-1, cmp);
42      }
43  }
44
45  void heap_sort(vector<int> &nums, bool (*cmp)(int,int)) {
46      build_heap(nums, cmp);
47      disp(nums);
48      int len = nums.size();
49      for (int i = len-1; i>=1; --i) {
50          swap(nums[0], nums[i]);
51          heapify(nums, 0, i-1, cmp);
52      }
53  }
54
55  int main() {
56      vector<int> nums = {4,6,3,7,8,5,9,1,2};
57      disp(nums);
58      heap_sort(nums, [](int a, int b){ return a < b;});
59      disp(nums);
60
61      return 0;
62  }
```

cs-notes/_tests/heap_sort.cpp
63

7. 归并排序

cs-notes/_tests/merge_sort_recur.cpp

```
1  #include "_utils.h"
2
3  #include <iostream>
4  #include <string>
5  #include <vector>
6  #include <algorithm>
7
8  using namespace std;
9
10 void _merge_sort(vector<int>& arr, vector<int>& tmp, int b, int e) {
11     if (b >= e)
12         return ;
13     // Divide and sort
14     int b1 = b, e1 = b + (e-b)/2;
15     int b2 = e1+1, e2 = e;
16     _merge_sort(arr, tmp, b1, e1); // arr[b1:e1] is sorted
17     _merge_sort(arr, tmp, b2, e2); // arr[b1:e2] is sorted
18     // Merge
19     int i = b;
20     while (b1<=e1 && b2<=e2)
21         tmp[i++] = (arr[b1] <= arr[b2]) ? arr[b1++] : arr[b2++];
22     while (b1<=e1)
23         tmp[i++] = arr[b1++];
24     while (b2<=e2)
25         tmp[i++] = arr[b2++];
26     // copy(tmp.begin()+b, tmp.begin()+e+1, arr.begin()+b);
27     for (int j=b; j<=e; ++j)
28         arr[j] = tmp[j];
29 }
30
31 void merge_sort(vector<int>& arr) {
32     vector<int> tmp(arr.size(), 0);
33     _merge_sort(arr, tmp, 0, arr.size()-1);
34 }
35
36 int main() {
```

cs-notes/_tests/merge_sort_recur.cpp

```
37      vector<int> arr = {2,3,5,1,6,9,8,4,7};
38      // vector<int> arr = {1,3,2,3,1};
39      disp(arr);
40      merge_sort(arr);
41      disp(arr);
42
43      return 0;
44  }
45
```

8. 插入排序

- 【排序算法】插入排序(C++实现)：https://blog.csdn.net/left_la/article/details/8656425

cs-notes/_tests/insertion_sort.cpp

```
1   #include "_utils.h"
2   #include <iostream>
3   #include <vector>
4
5   using namespace std;
6
7   void insertion_sort(vector<int> & a) {
8       for (int i=1; i<a.size(); ++i) {
9           int val = a[i];
10          int j = i;
11          while (--j >= 0) {
12              if (a[j] > val)
13                  a[j+1] = a[j];
14              else
15                  break;
16          }
17          a[j+1] = val;
18      }
19  }
20
21  int main() {
22      vector<int> a = {6,5,2,9,1,8,7,3,4};
23      disp(a);
24      insertion_sort(a);
25      disp(a);
26      return 0;
27  }
```

cs-notes/_tests/insertion_sort.cpp
28

9. 希尔排序

- 【排序算法】插入排序(C++实现)：https://blog.csdn.net/left_la/article/details/8656425

10. 二分查找

- 二分查找有几种写法？它们的区别是什么？：https://www.zhihu.com/question/36132386
- lower_bound - C++ Reference：http://www.cplusplus.com/reference/algorithm/lower_bound/
- binary_search - C++ Reference：http://www.cplusplus.com/reference/algorithm/binary_search/

C++标准库中的 binary_search 包含在头文件 <algorithm> 中，其实现基于 lower_bound（也在 <algorithm> 头文件中），但是做了两个改动：

- 添加了一条找到就返回索引的语句
- 若没有不存在数组中，则返回-1（而不是原来的 left）

不要小看二分查找，实现起来细节很多，直接看代码注释即可。

- 代码第 4 行：返回 target 的索引，若没有，则返回-1
- 代码第 9 行：使用 left < right，也即我们的搜索范围为 [left, right)，左闭右开空间，这么做可以避免很多边界条件的问题
- 代码第 10 行：不用 mid = (left+right)/2，避免溢出

- **cs_notes/_tests/binary_search.cpp**

```
1   #include "_utils.h"
2
3   // return index of target, if none, return -1
4   int my_binary_search(const std::vector<int> & nums, int target) {
5       int left = 0;
6       int right = nums.size();
7       int mid;
8
9       while (left < right) { // search through [left, right)
10          mid = left + (right-left)/2; // prevent overflow
11          if (nums[mid]==target)
12              return mid;
13          else if (nums[mid]<target)
14              left = mid + 1;
15          else
16              right = mid;
17      }
```

- **cs_notes/_tests/binary_search.cpp**

```
18      return -1;
19  }
20
21  int main() {
22      std::vector<int> v = {0,1,2,3,4,5,6,7};
23      disp(my_binary_search(v,4));
24      disp(my_binary_search(v,8));
25      disp(my_binary_search(v,-1));
26      return 0;
27  }
28
```

11. lower_bound 和 upper_bound 的使用

- std::lower_bound - cppreference.com: https://en.cppreference.com/w/cpp/algorithm/lower_bound
- std::upper_bound - cppreference.com: https://en.cppreference.com/w/cpp/algorithm/upper_bound
- lower_bound: http://www.cplusplus.com/reference/algorithm/lower_bound/
- upper_bound: http://www.cplusplus.com/reference/algorithm/upper_bound/

lower_bound(itr1, itr2, val)

- 在迭代器 itr1 和 itr2 之间的数必须有序，且从小到大排列
- 二分查找从左到右第一个**不小于 val** 的数，返回其迭代器
- 如果想从右到左找到第一个小于 val 的数，则使用反向迭代器，并传入 greater 参数

upper_bound(itr1, itr2, val)

- 在迭代器 itr1 和 itr2 之间的数必须有序，且从小到大排列
- 二分查找第一个**大于 val** 的数，返回其迭代器

若传入比较函数 greater，也即 lower_bound(itr1, itr2, val, greater<int>())

- 其含义为：对于 itr1 和 itr2 之间的值，若 a > b（greater(a,b)）为 true，则 a 排在 b 之前（左边）
- 所以这里其实是针对输入迭代器 itr1 和 itr2 的要求，若是整数数组，则 greater 比较器的要求是，传入的数组必须是逆序排列的，也即从大到小，否则比较返回的就是不正确的
- 注意这里比较器并不像 sort 函数中那样具备修改函数本身的能力，而只是对输入迭代器的限制

所以对于从小到大有序的数组 v，下面两种写法是等价的：

```
int L1 = lower_bound(v.begin(), v.end(), p) - v.begin();
int U2 = v.rend() - upper_bound(v.rbegin(), v.rend(), p, greater<int>());
```

此外，若想找到从右向左第一个小于 val 的数，只需在 lower_bound 的结果减去 1 即可：

```
// 二分查找 小于 p 的（从右向左）第一个元素
int L3 = lower_bound(v.begin(), v.end(), p) - v.begin() -1;
// 二分查找 不大于 p 的（从右向左）第一个元素
int U3 = upper_bound(v.begin(), v.end(), p) - v.begin() -1;
```

lower_bound 和 upper_bound 的使用

```
#include "_utils.h"
#include <vector>
#include <algorithm>  // lower_bound, upper_bound
#include <cstdio>     // printf

using namespace std;

int main() {
    // 数组 v 必须有序，且从小到大排列
    vector<int> v = {0,1,2,3,4,5,6,7,8,9};
    int p = 4;
    // 二分查找 不小于 p 的第一个元素
    int L1 = lower_bound(v.begin(), v.end(), p) - v.begin();    // 4
    // 二分查找 不小于 p 的第一个元素，与上一种等价
    int U2 = v.rend() - upper_bound(v.rbegin(), v.rend(), p, greater<int>());
// 4

    // 二分查找 大于 p 的第一个元素
    int U1 = upper_bound(v.begin(), v.end(), p) - v.begin(); // 5
    // 二分查找 大于 p 的第一个元素，与上一种等价
    int L2 = v.rend() - lower_bound(v.rbegin(), v.rend(), p, greater<int>());
// 5

    printf("%d %d %d %d \n", L1, U2, U1, L2);

    // 二分查找 小于 p 的（从右向左）第一个元素
    int L3 = lower_bound(v.begin(), v.end(), p) - v.begin() -1; // 3
    // 二分查找 不大于 p 的（从右向左）第一个元素
    int U3 = upper_bound(v.begin(), v.end(), p) - v.begin() -1; // 4

    printf("%d %d \n", L3, U3);

    return 0;
}

```

12. reverse 的实现

- reverse - C++ Reference：http://www.cplusplus.com/reference/algorithm/reverse/
- 《C++标准库》“11.8 变序型算法”，第 610 页
- C++ reverse 函数源码解析：https://blog.csdn.net/xiangxianghehe/article/details/90637239

C++中的 reverse 定义在头文件 <algorithm> 中。其实现基于 iter_swap。

核心思路就是用双指针，每个指针是一个双向迭代器，分别指向序列容器的头尾，然后向中间移动，交换迭代器，直到相遇。

- **cs_notes/_tests/impl_reverse.cpp**

```cpp
#include <iostream>
#include <vector>
// #include <algorithm> // reverse

template <typename BidirectionalIterator>
void my_reverse(BidirectionalIterator first, BidirectionalIterator last) {
    while ((first!=last) && (first!=--last)) {
        std::iter_swap(first, last);
        ++first;
    }
}

void disp_vector(std::vector<int> & v) {
    for (auto const &ele:v)
        std::cout << ele << " ";
    std::cout << std::endl;
}

int main() {
    std::vector<int> v = {1,2,3,4,5};
    disp_vector(v);
    my_reverse(v.begin(), v.end());
    disp_vector(v);
    return 0;
}

```

13. iter_swap 的实现

- iter_swap - C++ Reference：http://www.cplusplus.com/reference/algorithm/iter_swap/

C++中的 iter_swap 包含在头文件 <algorithm> 中。其实现基于 swap。

- **cs_notes/_tests/imple_iter_swap.cpp**

```cpp
#include <iostream>
#include <vector>
#include <algorithm> // swap, iter_swap
#include "_utils.h"

template <typename ForwardIter1, typename ForwardIter2>
void my_iter_swap(ForwardIter1 itr1, ForwardIter2 itr2) {
    std::swap(*itr1, *itr2);
}

int main() {
    std::vector<int> v = {1,2,3,4,5};
    disp_vec(v);
    my_iter_swap(v.begin(), v.end()-1);
    disp_vec(v);
}

```

14. swap 的实现

swap - C++ Reference：http://www.cplusplus.com/reference/algorithm/swap/

C++中的 swap 包含在头文件 <algorithm> 中。其实现基于一个临时变量。

- **cs_notes/_tests/impl_swap.cpp**

```cpp
#include <iostream>
#include <algorithm> // swap
#include "_utils.h"

template <typename T>
void my_swap (T& a, T& b) {
    T c(a);
    a = b;
    b = c;
}

int main() {
    int x = 1, y = 2;
    disp(x,y);
    my_swap(x, y);
    disp(x,y);
    return 0;
```

- **cs_notes/_tests/impl_swap.cpp**

```
18  }
19
```

15. rotate 的实现

rotate - C++ Reference：http://www.cplusplus.com/reference/algorithm/rotate/

C++中的 rotate 包含在头文件 <algorithm> 中。实现方法非常精妙，借用了 swap，需要拿纸笔画一画才能完全理解。

几个注意的点：

- 第 7 行：first != next 判断条件
  - 其实也可以用 fisrt != last，但是用 first != next 可以提前结束，不需要多余的 swap
- 第 10 ~ 11 行：若 next == last，则 next = mid
  - next == mid 表明原数组中 mid 的右侧元素已经处理完成（也即全都放到新数组的左边）
  - 故需要将 next 移动到 mid 处，以让 first 继续向右行进（处理 mid 的左侧元素）直到碰到 mid
- 第 12 ~ 13 行：若 first == mid，则 mid = next
  - first == mid 表明原数组中 mid 的左侧元素已经处理完成（也即全都放到了新数组的右边）
  - 故需要将 mid 移动到 next 处，以让 next 继续向右行进（处理 mid 的右侧元素）直到碰到 last
- 第 10 和 12 行可以同时满足，此时有 first == next，直接终止

这里实现的是左旋转，如果是右旋转，只需要使用反向迭代器，也即把 begin() 和 end() 改成 rbegin() 和 rend()。

- **cs_notes/_tests/impl_rotate.cpp**

```
1   #include <vector>
2   #include "_utils.h"
3
4   template <typename ForwardIter>
5   void my_rotate(ForwardIter first, ForwardIter mid, ForwardIter last) {
6       ForwardIter next = mid;
7       while (first != next) {
8       // `first != last` is also corrected, but more swaps
9           std::swap(*first++, *next++);
10          if (next == last)     // right of mid processed
```

- **cs_notes/_tests/impl_rotate.cpp**

```
            next = mid;
        else if (first == mid) // left of mid processed
            mid = next;
    }
}

int main() {
    // roate left
    std::vector<int> v1 = {1,2,3,4,5};
    my_rotate(v1.begin(), v1.begin()+3, v1.end());
    disp(v1);

    // rotate right
    std::vector<int> v2 = {1,2,3,4,5};
    my_rotate(v2.rbegin(), v2.rbegin()+3, v2.rend());
    disp(v2);

    return 0;
}

```

操作系统

1. 【热门】进程与线程的区别和联系

- 进程和线程的区别(超详细)：https://blog.csdn.net/ThinkWon/article/details/102021274
- 【面试】线程进程区别：https://juejin.cn/post/6844903809081163790

		进程	线程
1	联系	一个进程中至少有一个线程，可以有多个线程	
2	调度	进程是操作系统资源分配的基本单位	线程是处理器任务调度和执行的基本单位
3	资源分配	每个进程都有独立的代码和数据空间（上下文）	同一进程的各个线程共享代码和数据空间，不过每个线程有独立的运行栈和程序计数器
4	切换	从一个进程中的线程切换到另一个进程中的线程会引起进程切换 进程切换时需要保存当前执行进程的上下文，载入将执行进程的上下文	从同一进程中的线程切换不会引起进程切换 线程切换时只需保存和设置少量寄存器的内容
5	系统开销	较大	较小
6	健壮性	一个进程崩溃不会影响到其他进程 多进程要比多线程健壮	一个线程崩溃会使整个进程死掉

7	执行过程	每个独立的进程有程序运行的入口、顺序执行序列和程序出口	线程不能独立执行，必须依存在应用程序中，由应用程序控制多个线程的执行
8	通信	进程间通信需要同步和互斥等辅助手段，保证数据一致性	线程间可以直接读写同一进程中的数据段进行通信

2. Linux 理论上最多可以创建多少个进程？一个进程可以创建多少线程，和什么有关

3. 冯诺依曼结构有哪几个模块？分别对应现代计算机的哪几个部分？

4. 【热门】进程之间的通信方法有哪几种

- 《Unix 网络编程 卷 2：进程间通信》
- 进程之间究竟有哪些通信方式？如何通信？：https://blog.csdn.net/m0_37907797/article/details/103188294
- 进程间 8 种通信方式详解：https://blog.csdn.net/violet_echo_0908/article/details/51201278
- 进程中的五种通信方式介绍：https://blog.csdn.net/mulinsen77/article/details/88591695
- 进程间通信的 8 种方式：https://blog.csdn.net/qq_41333582/article/details/83988370
- 进程间通信的方式——信号、管道、消息队列、共享内存：https://www.cnblogs.com/luo77/p/5816326.html
- linux 进程间通信：https://www.bilibili.com/video/BV1tJ41117ty
- 凉了！张三同学没答好「进程间通信」，被面试官挂了....：https://juejin.cn/post/6856317967026798599

1、匿名管道（pipe）

- 进程间通信之管道（pipe、fifo）：https://www.cnblogs.com/MrListening/p/5858358.html
- 《Unix 网络编程 卷 2》“第 4 章 管道和 FIFO”“4.3 管道”
- Linux 管道 pipe 的实现原理：https://segmentfault.com/a/1190000009528245
- Linux 的进程间通信：管道：https://segmentfault.com/a/1190000018411174?utm_source=sf-related
- Linux 进程间通信——使用匿名管道：https://blog.csdn.net/ljianhui/article/details/10168031

Linux 中的竖线 `|` 其实就是一个管道

将前一个命令 `ps auxf` 的输出作为后一个命令 `grep mysql` 的输入。

```
ps auxf | grep mysql
```

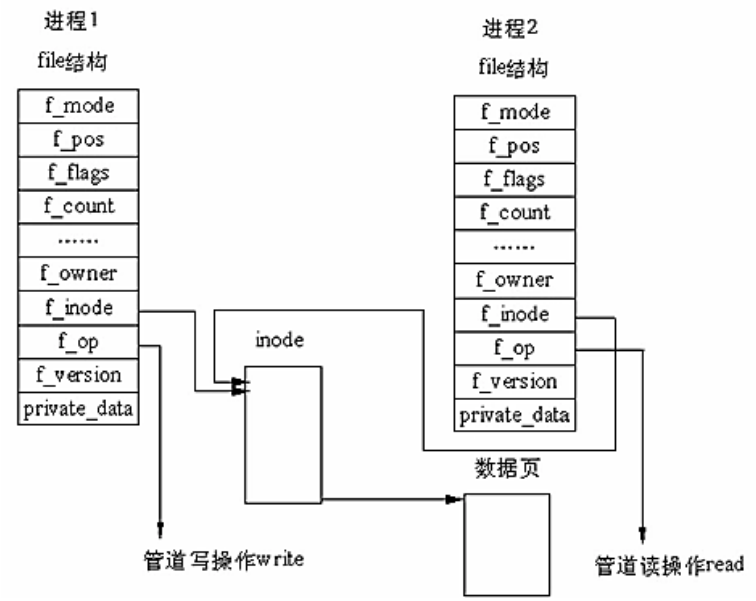
管道的特点

1. 半双工：支持两个方向，但是同一时刻只能在一个方向上传输
   - 双方需要通信的时候，需要建立两个管道
2. 只能在具有亲缘关系的进程（父子进程）间使用
3. 是一种文件系统，但并不占用磁盘或外部存储空间，而是在内存中，管道实际上是内存缓冲区（设计为环形的数据结构，以便循环利用）
4. 生命周期跟随进程

5. 面向字节流
6. 内部提供同步机制

管道的实现

管道的实现借助了文件系统的 file 结构和 VFS 的索引节点 inode：将两个 file 结构指向同一个临时的 VFS 索引节点，该节点又指向一个物理页面。



管道的底层实现

管道的函数

pipe 函数创建管道：

```
#include <unistd.h>
int pipe(int fd[2]); // 返回：若成功则为 0，若出错则为 -1
```

该函数返回两个文件描述符：fd[0] 和 fd[1]。前者打开来读，后者打开来写。

标准 I/O 函数提供了 popen 函数，创建一个管道并启动另一个进程，该进程要么从管道读出标准输入，要么往管道写入标准输出。pclose 函数关闭由 popen 创建的标准 I/O 流，等待其中的命令终止，然后返回 shell 的终止状态。

```
#include <stdio.h>
// 返回：若成功则为文件指针，若出错则为 NULL
FILE *popen(const char *command, const char *type);
// 返回：若成功则为 shell 的终止状态，若出错则为 -1
int pclose(FILE *stream);
```

创建管道的流程

①（图 4-3）一个进程（将成为父进程）创建管道后，调用 fork 派生一个自身的副本

②（图 4-4）父进程关闭管道的读出端（fd[0]），子进程关闭写入端（fd[1]），这时父子进程间就有了一个单向数据流

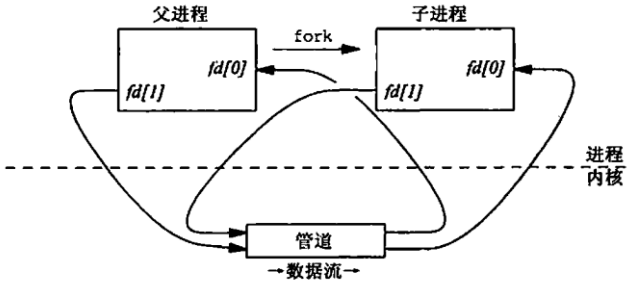


图4-3 单个进程内的管道，刚刚fork后

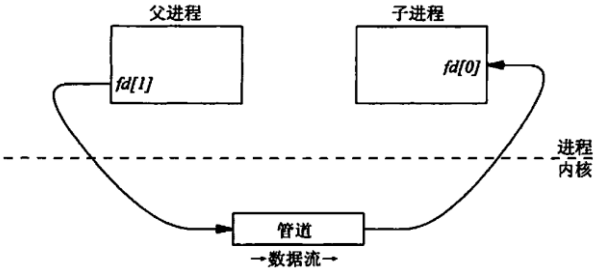
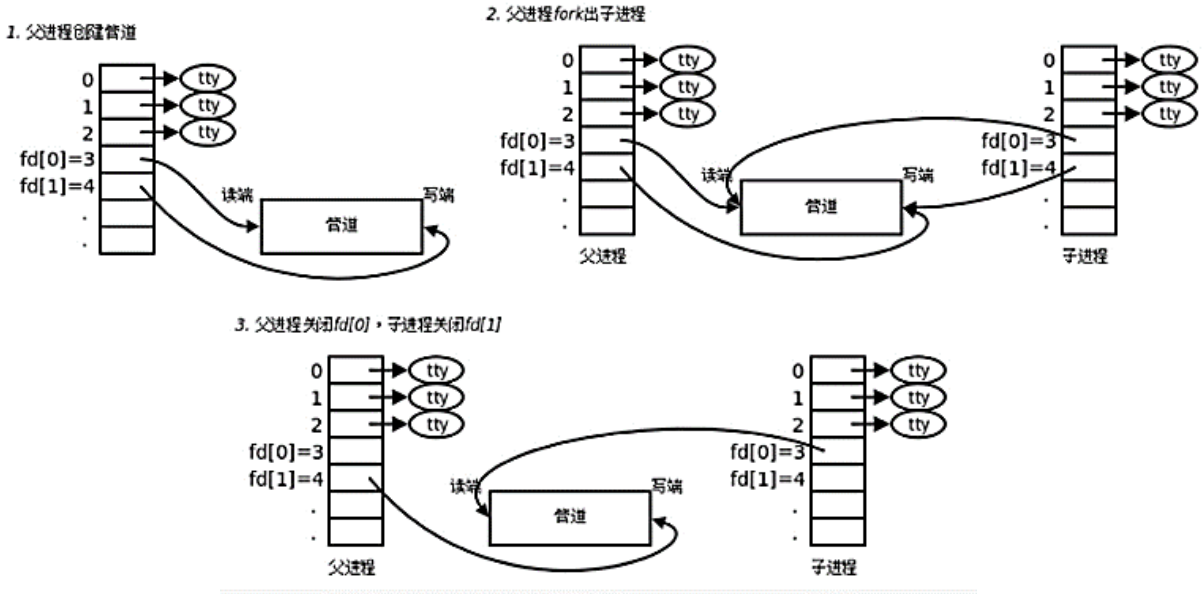


图4-4 两个进程间的管道

下图展示创建过程的底层实现：（1）创建（2）fork（3）关闭



管道创建过程的底层实现

比如命令 `who | sort | lp` 将会创建三个进程和两个管道：

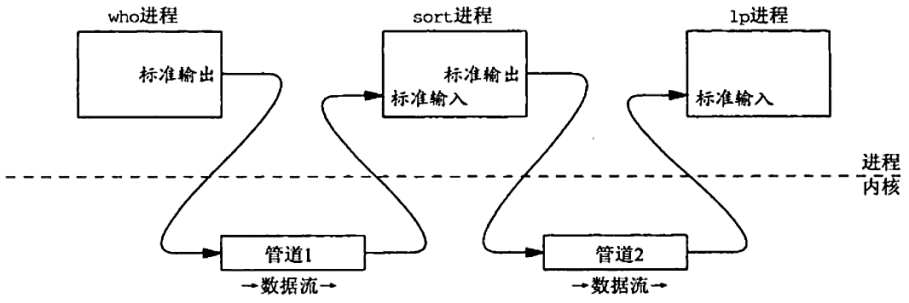


图4-5 某个shell管道线中三个进程间的管道

如果需要一个双向的数据流，必须创建两个管道，两个方向各一个，步骤如下：

1. 创建管道 1（fd1[0]、fd1[1]）和管道 2（fd2[0]、fd2[1]）
2. fork
3. 父进程关闭管道 1 的读出端（fd1[0]）
4. 父进程关闭管道 2 的写入端（fd2[1]）
5. 子进程关闭管道 1 的写入端（fd1[1]）
6. 子进程关闭管道 2 的读出端（fd2[0]）

产生的管道布局如图 4-6 所示：

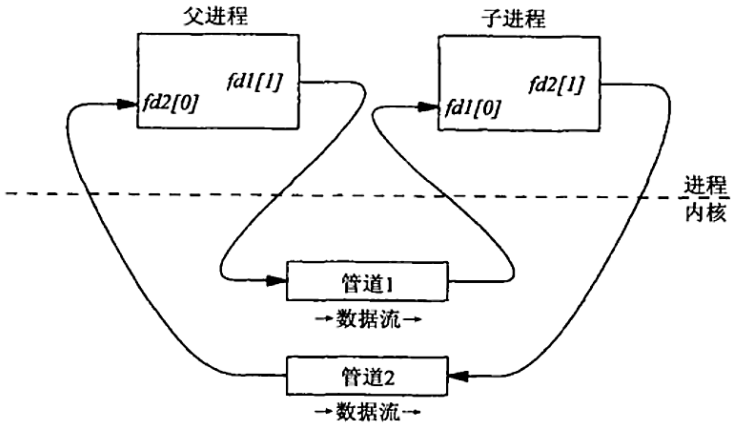


图4-6 提供一个双向数据流的两个管道

图 4-1 是一个客户-服务器的例子。图 4-7 是使用两个管道实现该例子。
其中客户为父进程，服务器为子进程。

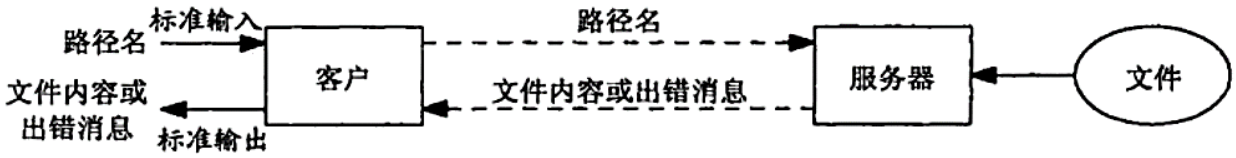


图4-1 客户-服务器例子

图4-7 使用两个管道实现图4-1

管道的读写

通过 pipe_read() 和 pipe_write() 对管道进行读写。

- 管道写函数：通过将字节复制到 VFS 索引节点指向的物理内存而写入数据
  - 当写进程向管道中写入时，它利用标准的库函数 write()，系统根据库函数传递的文件描述符，可找到该文件的 file 结构
  - file 结构中指定了写入函数的地址，内核调用该函数完成写操作
    - 写入函数首先锁定内存，然后从写进程的地址空间中复制数据到内存
    - 当数据写入内存之后，内存被解锁，而所有休眠在索引节点的读取进程会被唤醒
  - 管道和 FIFO 的 write 操作的原子性相当重要。
- 管道读函数：通过复制物理内存中的字节而读出数据。
  - 管道的读取过程和写入过程类似，但是，进程可以在没有数据或内存被锁定时，立即返回错误信息，而不是阻塞该进程，这依赖于文件或管道的打开模式
  - 反之，进程可以休眠在索引节点的等待队列中，等待写入进程写入数据

当所有的进程完成了管道操作之后，管道的索引节点被丢弃，而共享数据页也被释放

四种情况处理

1. 所有指向管道写端的文件描述符都关闭了，此时有进程从管道的读端读数据：
   - 那么管道中剩余的数据都被读取后，再次 read 会返回 0，就像读到文件末尾一样
2. 如果存在指向管道写端的文件描述符未关闭，而持有管道写端的进程也未向管道中写数据，这时有进程从管道读端读数据：
   - 那么管道中剩余的数据都被读取后，再次 read 会阻塞，直到管道中有数据可读了才读取数据并返回
3. 如果所有指向管道读端的文件描述符都关闭了，这时有进程指向管道的写端写数据：
   - 那么该进程会收到信号 SIGPIPE，通常会导致进程异常终止
4. 如果存在指向管道读端的文件描述符没关闭，而持有管道读端的进程也没有从管道中读数据，这时有进程向管道写端写数据：
   - 那么在管道被写满时再 write 会阻塞，直到管道中有空位置了才写入数据并返回。

2、命名管道（fifo）

- Linux 进程间通信——使用命名管道：https://blog.csdn.net/ljianhui/article/details/10202699
- 《Unix 网络编程 卷 2》“4.6 FIFO”、“4.8 单个服务器，多个客户”

命名管道 fifo 和匿名管道 pipe 的行为类似，只是有如下不同：

		匿名管道 pipe	命名管道 FIFO
1	调用函数	创建并打开管道需要调用 pipe	创建并打开 FIFO 需要在调用 mkfifo 后再调用 open
2	名字	管道没有名字	FIFO 以文件路径作为名字
3	进程关系	管道需要通信的两个进程有亲缘关系	FIFO 不需要两个进程有亲缘关系
4	关闭	在所有进程最终都关闭它后自动消失	只有调用 unlink 才从文件系统中删除

FIFO 通过 mkfifo 函数创建：

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode) // 返回：若成功则为 0，若出错则为 -1
```

其中，pathname 是一个普通的 Unix 文件路径名，也是该 FIFO 的名字。

创建出一个 FIFO 后，必须或者打开来读，或者打开来写，但不能既读又写（和管道一样是半双工）。可以是 open 函数，也可以是某个标准 I/O 打开函数，比如 fopen。

FIFO 的 write 总是往末尾添加数据，read 总是从开头返回数据。如果对 pipe 和 FIFO 调用 lseek，则返回 ESPIPE 错误。

图 4-17 展示了使用两个 FIFO 的客户-服务器例子。其中父进程是客户，子进程是服务器。

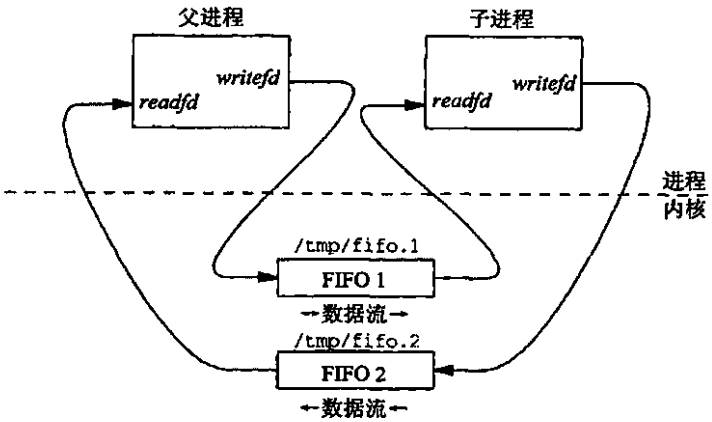


图4-17 使用两个FIFO的客户-服务器例子

FIFO 的真正优势表现在：服务器可以是一个长期运行的进程（例如守护进程），而且其与客户可以无亲缘关系。下图是一个单服务器多客户的例子：

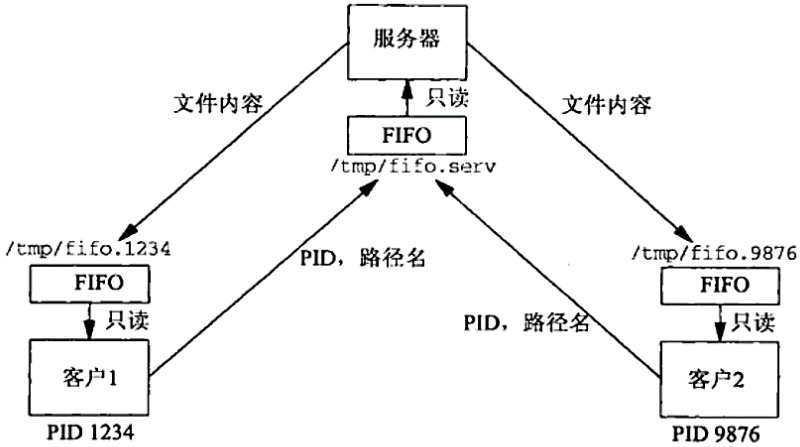


图4-22 单个服务器，多个客户

3、消息队列（message queue）

- Linux 进程间通信——使用消息队列：https://blog.csdn.net/ljianhui/article/details/10287879
- 进程间通信的方式（三）：消息队列：https://zhuanlan.zhihu.com/p/37891272
- 消息队列设计精要：https://tech.meituan.com/2016/07/01/mq-design.html

定义

消息队列可以认为是一个消息链表。每个消息都是一个记录，它由发送者赋予一个优先级。

- 有足够写权限的线程可以往队列中放置消息
- 有足够读权限的线程可以从队列中取走消息

一个 Posix 消息队列的典型布局如下：

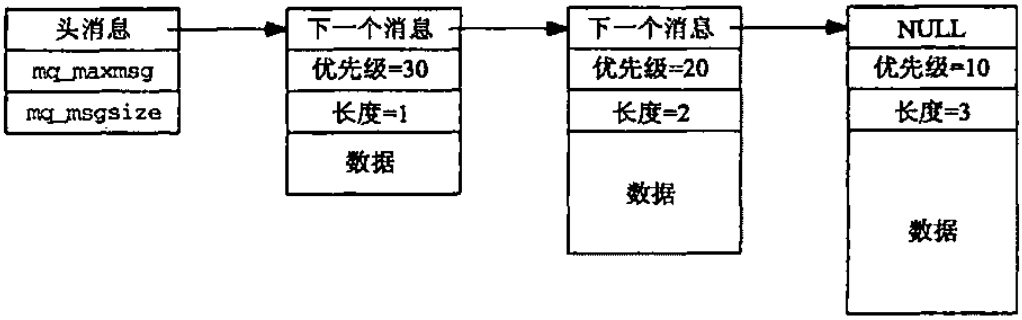


图5-1 含有三个消息的某个Posix消息队列的可能布局

消息队列与管道和 FIFO 的不同

	消息队列	管道和 FIFO
1	在某个进程往一个队列写入消息之前，并不需要另外某个进程在该队列上等待消息的到达	除非读出者已经存在，否则先有写入者没有意义
2	消息队列具有随内核的持续性	当一个管道或 FIFO 最后一次关闭时，仍在该管道或 FIFO 上的数据将被丢弃
3	队列中的每个消息包含三个属性： 优先级、数据长度、数据本身	是字节流模型，没有消息边界，也没有与每个消息关联的类型
4	接受程序可以通过消息类型有选择地接收数据	只能默认地接收数据

消息队列的函数（以 Posix 为例）

mq_open 函数创建一个新的消息队列或者打开一个已经存在的消息队列。mq_open 的返回值称为消息队列描述符。

```
#include <mqueue.h>
// 返回：若成功则为消息队列描述符，若出错则为-1
mqd_t mq_open(const char *name, int oflag, ...
              /* mode_t mode, struct mq_attr* attr */);
```

oflag 参数是 O_RDONLY、O_WRONLY 或 O_RDWR 之一，可能按位或上 O_CREAT、O_EXCL 或 O_NONBLOCK。详见下表。

打开或创建 Posic IPC 对象所用的各种 oflag 的值

说明	mq_open	sem_open	shm_open
只读 只写 读写	O_RDONLY O_WRONLY O_RDWR		O_RDONLY
若不存在则创建 排他性创建	O_CREAT O_EXCL	O_CREAT O_EXCL	O_CREAT O_EXCL
非阻塞模式 若已存在则截短	O_NONBLOCK		 O_TRUNC

mq_close 关闭一个已打开的消息队列。其功能与关闭一个已打开文件的 close 函数类似：

- 调用进程可以不再使用该描述符，但其消息队列并不从系统中删除。

一个进程终止时，它的所有打开着的消息队列都关闭，就像调用了 mq_close 一样

```
#include <mqueue.h>
int mq_close(mqd_t mqdes);
```

mq_unlink 从系统中删除某个 name（mq_open 中的第一个参数）。

```
#include <mqueue.h>
int mq_unlink(const char *name);
```

每个消息队列都有一个保存其打开着描述符的引用计数器（就像文件一样），因此本函数能够实现类似于 unlink 函数删除一个文件的机制：

- 当以消息队列的引用计数仍大于 0 时，其 name 就能删除，但是该队列的析构（不同于从系统中删除其名字）要到最后一个 mq_close 发生时才行。
- 一个消息队列的名字在系统中的存在本身也占用其引用计数器的一个引用数，mq_unlink 从系统中删除该名字意味着同时将其引用计数减 1，若变为 0 则真正拆除该队列。
- mq_close 也会将当前消息队列的引用计数减 1。

每个消息队列有 4 个属性，mq_getattr 返回所有这些属性，mq_setattr 设置其中某个属性。

```
#include <mqueue.h>
// 二者均返回：若成功则为 0，若出错则为-1
int mq_getattr(mqd_t mqdes, struct mq_attr *attr);
int mq_setattr(mqd_t mqdes, const struct mq_attr *attr, struct mq_attr *oattr);
```

mq_attr 结构含有如下属性：

```
struct mq_attr {
    long mq_flags;   // message queue flag: 0, O_NONBLOCK
    long mq_maxmsg;  // max num of messages allowed on queue
    long mq_msgsize; // max size of a message (in bytes)
    long mq_curmsgs; // num of messages currently on queue
};
```

mq_send 和 mq_receive 分别向队列中放置消息和取走消息。

- mq_receive 总是返回所指定队列中最高优先级的最早消息，而且该优先级能随该消息的内容和长度一起返回。
- 这两个函数的前 3 个参数与 write 和 read 的前 3 个参数类似。
  - mqdes：要发送消息的消息队列描述符
  - *ptr：要发送的数据
  - len：消息长度

- prio：消息的优先级，它是一个小于 MQ_PRIO_MAX 的无符号整数，Posix 要求这个上限至少为 32。

```
#include <mqueue.h>
// 返回：若成功则为 0，若出错则为-1
int mq_send(mqd_t mqdes, const char *ptr, size_t len, usigned int prio);
// 返回：若成功则为消息中字节数，若出错则为-1
ssize_t mq_receive(mqd_t mqdes, char *ptr, size_t len, usigned int *priop);
```

mq_notify 为指定队列建立或删除异步事件通知。Posix 消息队列允许异步事件通知，以告知何时有一个消息放置到了某个空消息队列中。这种通知有两种方式：

- 产生一个信号
- 创建一个线程来执行一个指定的函数

```
#include <mqueue.h>
// 返回：若成功则为 0，若出错则为-1
int mq_notify(mqd_t mqdes, const struct sigevent *notification);
```

消息队列描述符（mqd_t 变量）不是“普通”描述符，不能用在 select 或 poll 中，但是可以伴随一个管道和 mq_notify 函数使用它们。

4、信号量（semaphore）

- Linux 进程间通信——使用信号量：https://blog.csdn.net/ljianhui/article/details/10243617
- 《Unix 网络编程 卷 2：进程间通信》“第 10 章 Posix 信号量”

什么是信号量

信号量（semaphore）是一种用于提供不同进程间或一个给定进程的不同线程间同步手段的原语。

为了防止出现因多个程序同时访问一个共享资源而引发的一系列问题，我们需要一种方法，它可以通过生成并使用令牌来授权，在任一时刻只能有一个执行线程访问代码的临界区域。

- 临界区域是指执行数据更新的代码需要独占式地执行。
- 而信号量就可以提供这样的一种访问机制，让一个临界区同一时间只有一个线程在访问它，也就是说，**信号量是用来协调进程对共享资源的访问的**。

信号量是一个特殊的变量，程序对其访问都是原子操作，且只允许对它进行等待（P）和发送（V）操作。

最简单的信号量是只能取 0 和 1 的变量，这也是信号量最常见的一种形式，叫做二进制信号量（二值信号量），而可以取多个正整数的信号量被称为通用信号量（计数信号量）。

System V 信号量是由内核来维护的；Posix 信号量不必在内核中维护，并且是由可能与文件系统中的路径名对应的名字来标识的。

有三种类型的信号量，都用于进程或线程间的同步：

- Posix 有名信号量：使用 Posix IPC 名字标识

- Posix 基于内存的信号量（无名信号量）：存放在共享内存区中
- System V 信号量：在内核中维护

图 10-1 展示了一个二值信号量。

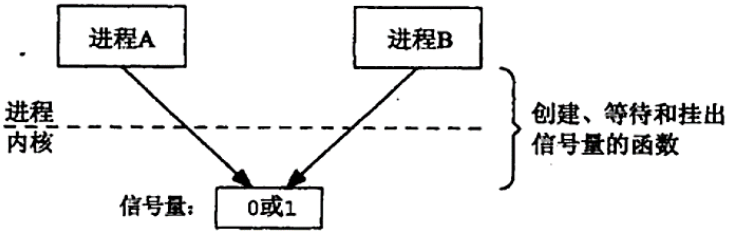


图10-1 由两个进程使用的一个二值信号量

图 10-2 是一个 Posix 有名二值信号量的更为符合实际的图示，其不必在内核中维护。

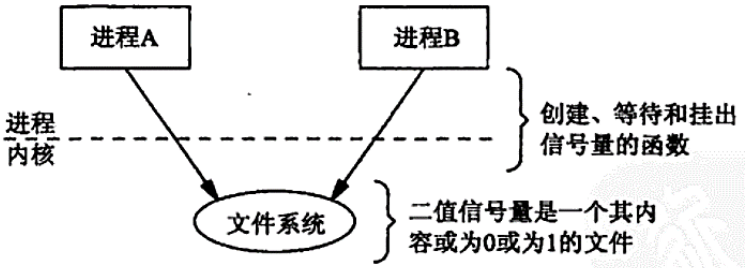


图10-2 由两个进程使用的一个Posix有名二值信号量

图 10-7 展示了一个 Posix 基于内存的信号量：

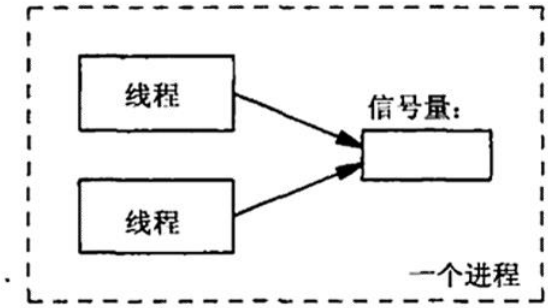


图10-7 由一个进程内的两个线程共享的基于内存的信号量

图 10-8 展示了某个共享内存区中由两个进程共享的一个 Posix 基于内存的信号量。图中画出了该共享内存区同时属于这两个进程的地址空间。

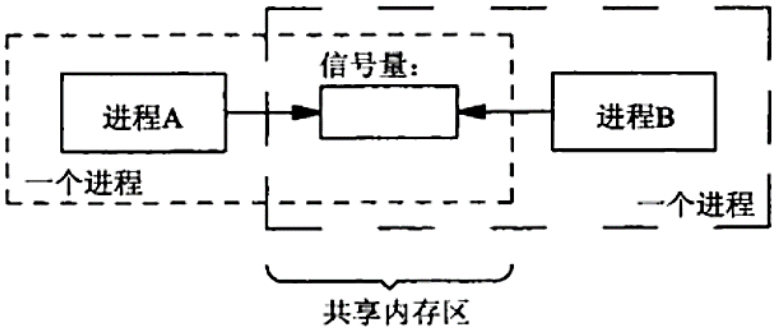


图10-8 由两个进程共享、处于共享内存区中的基于内存的信号量

进程对信号量的操作

一个进程在某个信号量上可以执行三种操作：

① 创建（create）：还要求调用者指定初始值，对于二值信号量，通常是 1，但也可以是 0

② 等待（wait）：该操作会测试这个信号量的值，如果其值≤ 0，则等待（阻塞），一旦其值> 0，就将其减 1。伪代码如下：

```
while (semaphore_value <= 0)
    ; /* wait: block the thread or process */
semaphore_value--;
/* we have the semaphore */
```

- 访问同一信号量的其他线程或进程，在 while 语句中**测试该信号量的值**和**之后将它减 1**（若> 0）这两个步骤必须作为一个原子操作完成。
- **等待 wait 也称为 P 操作**（荷兰语 proberen，意为尝试），也称为递减（down）或**上锁（lock）**。

③ 挂出（post）：将信号量的值加 1。伪代码如下：

```
semaphore++;
```

- 如果有一些进程阻塞着等待该信号量的值变为> 0，其中一个进程现在就可能被唤醒。与刚刚等待伪代码一样，访问同一信号量的其他进程，挂出操作也必须是原子的。
- **挂出 post 也称为 V 操作**（荷兰语 verhogen，意为增加），也称为递增（up）、**解锁（unlock）**或发信号（signal）。

上面的两段伪代码并未假定是二值信号量，任意初值非负的信号量都适用。这样的信号量称为计数信号量（counting semaphore），通常初始化为某个值 N，指示可用的资源（比如缓冲区）数。

二值信号量可用于互斥目的，就就像互斥锁一样：

解决互斥问题时使用互斥锁和信号量的对比

互斥锁	信号量
初始化互斥锁：	初始化信号量为 1：
`pthread_mutex_lock(&mutex);`	`sem_wait(&sem);`
临界区：	临界区：
`pthread_mutex_unlock(&mutex);`	`sem_post(&sem);`

- 初始化信号量为 1，sem_wait 调用等待其值变为大于 0，然后将它减 1
- sem_post 调用将其值加 1（从 0 到 1），然后唤醒阻塞在 sem_wait 调用中等待该信号量的任何线程

信号量还有一个互斥锁没有的特性：互斥锁必须总是由锁住它的线程解锁，信号量的挂出却不必由执行过它的等待操作的同一线程执行。

[博客] 信号量只能进行两种操作，等待和发送信号，即 P(sv)和 V(sv)：

- P(sv)：如果 sv 的值大于 0，就给它减 1；如果它的值等于 0，就挂起该进程的执行
- V(sv)：如果有其他进程因等待 sv 而被挂起，就让它恢复运行，如果没有进程因等待 sv 而挂起，就给它加 1

例如，有两个进程共享信号量 sv，一旦其中一个进程执行了 P(sv) 操作，它将得到信号量，并可以进入临界区，使 sv 减 1。而第二个进程将被阻止进入临界区，因为当它试图执行 P(sv) 时，sv 为 0，它会被挂起以等待第一个进程离开临界区域并执行 V(sv) 释放信号量，这时第二个进程就可以恢复执行。

用信号量解决生产者-消费者问题

以生产者-消费者问题为例（参见生产者-消费者问题），使用两个二值信号量来解决。
为简单起见，在本例中，假设该缓冲区只能容纳一个条目（因此二值信号量就够用了）。

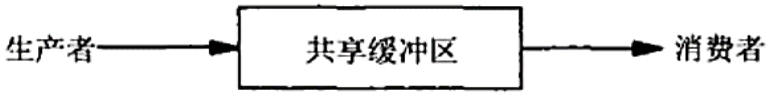


图10-4 使用一个共享缓冲区的简单生产者-消费者问题

下表给出了生产者和消费者程序的伪代码：

生产者	消费者
把信号量 get 初始化为 0 把信号量 put 初始化为 1	

for (; ;) {
sem_wait(&put);
把数据放入缓冲区
sem_post(&get);
}
	for (; ;) {
	sem_wait(&get);
	处理缓冲区中的数据
	sem_post(&put);
	}

- 信号量 put 控制生产者是否可以往共享缓冲区中放置一个条目
- 信号量 get 控制消费者是否可以从共享缓冲区中取走一个条目

按照时间顺序发生如下步骤：

1. 生产者初始化缓冲区和两个信号量（get 为 0，put 为 1）
2. 假设消费者接着运行，它阻塞在 sem_wait(&get) 调用中（因为 get 为 0）
3. 一段时间后生产者接着运行，当它调用 sem_wait 后，put 的值从 1 减为 0。
   - 于是生产者向缓冲区中放置一个条目，然后调用 sem_post，把 get 的值从 0 加为 1。
   - 此时消费者线程阻塞在 get 上等待其变为正数，该消费者被标记为 ready_to_run（准备好运行）。
   - 假设生产者继续运行，则会阻塞在 for 循环顶部的 sem_wait(&put) 调用中（因为 put 为 0），生产者必须等待消费者腾空缓冲区
4. 消费者从 sem_wait 调用中返回，将 get 的值从 1 减为 0
   - 接着消费者会处理缓冲区中的数据，然后调用 sem_post，将 put 的值从 0 加到 1。

- 此时生产者线程阻塞在 put 上等待其变为正数，该生产者被标记为 ready_to_run（准备好运行）。
- 假设消费者继续运行，则会阻塞在 for 循环顶部的 sem_wait(&get) 调用中（因为 get 为 0），消费者必须等待生产者向缓冲区中放入数据

5. 生产者从 sem_wait 调用中返回，把数据放入缓冲区中，之后重复上述步骤 2 ~ 4。

有名信号量和基于内存的信号量的函数差别

下图展示了 Posix 中有名（named）信号量和基于内存的（memory-based）信号量（也叫无名（unnamed）信号量）中函数的差别：

	有名信号量	基于内存的信号量（无名信号量）
1	sem_open()	sem_init()
2	sem_wait()	
3	sem_trywait()	
4	sem_post()	
5	sem_get_value()	
6	sem_close()	sem_destroy()
7	sem_unlink()	

sem_open，sem_close 和 sem_unlink 函数

sem_open 创建一个新的或打开一个已存在的有名信号量。有名信号量既可用于线程间的同步，又可用于进程间的同步。

```
#include <semaphore.h>
// 返回：若成功则为指向信号量的指针，若出错则为 SEM_FAILED
sem_t *sem_open(const char *name, int oflag, ...
                /* mode_t mode, unsigned int value */);
```

sem_close 关闭一个使用 sem_open 打开的有名信号量。

- 一个进程终止时，内核对其上仍然打开着的所有有名信号量自动执行这样的信号量关闭操作。不论该进程是自愿终止的（调用 exit 或_exit）还是非自愿终止的（向它发送一个 Unix 信号），这种自动关闭都会自动发生。
- 关闭一个信号量并没有将它从系统中删除，也就是说，Posix 有名信号量至少是随内核持续的：即使当前没有进程打开着某个信号量，它的值依然保持。

```
#include <semaphore.h>
// 返回：若成功则为 0，若出错则为-1
int sem_close(sem_t *sem);
```

sem_unlink 从系统中删除一个有名信号量。

- 每个信号量有一个引用计数器记录当前的打开次数（就像文件一样），sem_unlink 类似文件 I/O 函数的 unlink 函数：

- 当引用计数还是大于 0 时，name 就能从文件系统中删除
- 然而其信号量的析构（不同于将它的名字从文件系统中删除）却要等到最后一个 sem_close 发生时为止

```
#include <semaphore.h>
// 返回：若成功则为 0，若出错则为-1
int sem_unlink(const char *name);
```

sem_wait 和 sem_trywait 函数

sem_wait 函数测试所指定信号量的值：

- 如果该值大于 0，那就将它减 1 并立即返回
- 如果该值等于 0，调用线程就被投入睡眠中，直到该值变为大于 0，这时再将它减 1，函数随后返回
- “测试并减 1”的操作必须是原子的

```
#incluse <semaphore.h>
// 均返回：若成功则为 0，若出错则为-1
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
```

sem_trywait 和 sem_wait 的差别是：

- 当所指定信号量的值已经是 0 时，sem_trywait 并不将调用线程投入睡眠，而是返回一个 EAGAIN 错误。
- 如果被某个信号中断，sem_wait 就可能过早地返回，返回错误为 EINTR。

sem_post 和 sem_getvalue 函数

一个线程使用完某个信号量时，应该调用 sem_post 函数，将指定信号量的值加 1，然后唤醒正在等待该信号量变为正数的任意线程。

sem_getvalue 返回由 valp 指向的整数中所指定信号量的当前值。若该信号量当前已上锁，返回 0 或负数，其绝对值就是等待该信号量解锁的线程数。

```
#include <semaphore.h>
int sem_post(sem_t *sem);
int sem_getvalue(sem_t *sem, int *valp);
```

信号量、互斥锁和条件变量的差异

	信号量	互斥锁	条件变量
1	信号量的挂出不必由执行过其等待操作的同一线程执行	互斥锁必须总是由给它上锁的线程解锁	

	任何一个线程都可以挂出一个信号，即使当时没有线程在等待该信号量变为正数也没关系		
2		互斥锁要么被锁住，要么被解开（二值状态，类似二值信号量）	
3	由于信号量有一个与之关联的状态（计数值），因此信号量的挂出操作总是被记住		当向一个条件变量发送信号时（pthread_cond_signal），如果没有线程等待在该条件变量上（pthread_cond_wait），那么该信号将丢失
4	在各种同步技巧中，能够从信号处理程序中安全调用的唯一函数是 sem_post		

这些差异不能视为对信号量的偏袒，所有的同步原语（互斥锁、条件变量、读写锁、信号量、记录上锁）都有其各自的位置：

- 互斥锁是为上锁而优化的
- 条件变量是为等待而优化的
- 信号量既可用于上锁，也可用于等待，因而可能导致更多的开销和更高的复杂性

5、信号（signal）

- Linux 进程间通信——使用信号：https://blog.csdn.net/ljianhui/article/details/10128731
- 《Linux 高性能服务器编程》“第 10 章 信号”

信号是由用户、系统或者进程发送给目标进程的信息，以通知目标进程某个状态的改变或系统异常。**信号是开销最小的。**

信号可以在任何时候发送给某一进程，而无须知道该进程的状态。

- 如果该进程并未处于执行状态，则该信号就由内核保存起来，知道该进程恢复执行并传递给他为止。
- 如果一个信号被进程设置为阻塞，则该信号的传递被延迟，直到其阻塞被取消时才被传递给进程。

Linux 信号可由如下条件产生：

- 对于前台进程，用户可以通过输入特殊的终端字符来给它发送信号。比如输 Ctrl+C 通常会给进程发送一个中断信号。
- 系统异常。比如浮点异常和非法内存段访问。
- 系统状态变化。比如 alarm 定时器到期将引起 SIGALRM 信号。
- 运行 kill 命令或调用 kill 函数。

服务器程序必须处理（或至少忽略）一些常见的信号，以免异常终止。

Linux 下，一个进程给其他进程发送信号的 API 是 kill 函数。

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

目标进程在收到信号时，需要定义一个接收函数来处理之。信号处理函数的原型如下：

```
#include <signal.h>
typedef void (*__sighandler_t) ( int );
```

信号处理函数只带有一个整型参数，该参数用来指示信号类型。信号处理函数应该是可重入的，否则很容易引发一些竞态条件。所以在信号处理函数中严禁调用一些不安全的函数。
除了用户自定义信号处理函数外，<bits/signum.h> 头文件中还定义了信号的两种其他处理方式——SIG_IGN 和 SIG_DFL：

```
#include <bits/signum.h>
#define SIG_DFL ((__sighandler_t) 0)
#define SIG_IGN ((__sighandler_t) 1)
```

SIG_IGN 表示忽略目标信号，SIG_DFL 表示使用信号的默认处理方式。信号的默认处理方式有如下几种：结束进程（Term）、忽略信号（Ign）、结束进程并生成核心转储文件（Core）、暂停进程（Stop），以及继续进程（Cont）。
Linux 的可用信号都定义在 <bits/signum.h> 头文件中，其中包括标准信号和 POSlX 实时信号。

6、共享内存（shared memory）

- Linux 进程间通信——使用共享内存：https://blog.csdn.net/ljianhui/article/details/10253345
- 《Unix 网络编程 卷 2：进程间通信》“第四部分 共享内存区”“第 12 章 共享内存区介绍”和“第 13 章 Posix 共享内存区”

共享内存的概念

顾名思义，**共享内存就是允许两个不相关的进程访问同一个逻辑内存。**共享内存是在两个正在运行的进程之间共享和传递数据的一种非常有效的方式，是可用 IPC 形式中最快的。

- 不同进程之间共享的内存通常安排为同一段物理内存。
- 进程可以将同一段共享内存连接到它们自己的地址空间中，所有进程都可以访问共享内存中的地址，就好像它们是由用 C 语言函数 malloc 分配的内存一样。
- 而如果某个进程向共享内存写入数据，所做的改动将立即影响到可以访问同一段共享内存的任何其他进程。

特别提醒：**共享内存并未提供同步机制**，也就是说，在第一个进程结束对共享内存的写操作之前，并无自动机制可以阻止第二个进程开始对它进行读取。
所以往该共享内存区存放信息或从中取走信息的进程间通常需要某种形式的同步：

- 互斥锁、条件变量、读写锁、记录锁、信号量。

一旦这样的内存区映射到共享它的进程的地址空间，这些进程间数据的传递就不再涉及内核。

- 这里说的“不再涉及内核”的含义是：进程不再通过执行任何进入内核的系统调用来彼此传递数据。显然，内核必须建立允许各个进程共享该内存区的内存映射关系，然后一直管理该内存区（处理页面故障等）。

考虑用来传递各种类型消息的一个示例客户-服务器文件复制程序中涉及的通常步骤。

① 服务器从输入文件读。该文件的数据由内核读入自己的内存空间，然后从内核复制到服务器进程。

② 服务器往一个管道、FIFO 或消息队列以一条消息的形式写入这些数据。这些 IPC 形式通常需要把这些数据从进程复制到内核。

③ 客户从该 IPC 通道读出这些数据，这通常需要把这些数据从内核复制到进程。

④ 最后，将这些数据从由 write 函数的第二个参数指定的客户缓冲区复制到输出文件。

图 12-1 展示了客户与服务器之间通过内核桥接的数据转移。这里通常需要总共四次数据复制。而且这四次复制是在内核和某个进程间进行的，往往开销很大（比纯粹在内核中或单个进程内复制数据的开销大）。

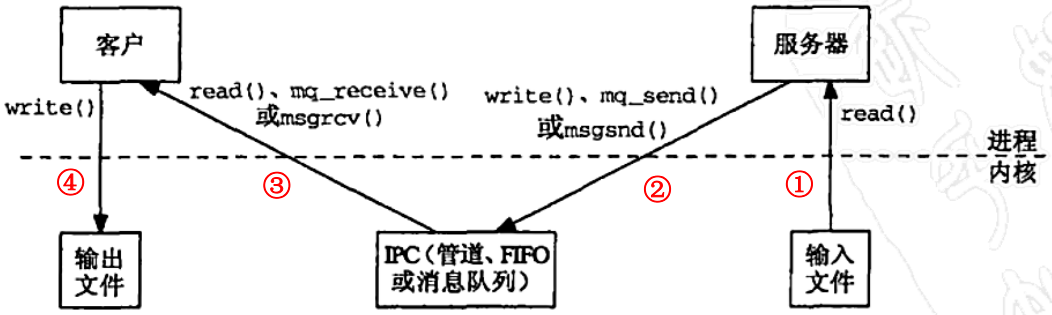


图12-1 从服务器到客户的文件数据流

这些 IPC 形式（管道、FIFO 和消息队列）的问题在于：

- 两个进程要交换信息时，这些信息必须经由内核传递。

通过让两个或多个进程共享一个内存区，共享内存区这种 IPC 形式提供了绕过上述问题的办法。当然，这些进程必须协调或同步对该共享内存区的使用。

- 共享一个公共的内存区跟共享一个硬盘文件类似，例如本书所有文件上锁例子中都使用的那个序列号文件。

前面的客户-服务器例子现在涉及的步骤如下。图 12-2 展示了这个情形。

① 服务器使用（譬如说）一个信号量取得访问某个共享内存区对象的权力。

② 服务器将数据从输入文件读入到该共享内存区对象。

- read 函数的第二个参数所指定的数据缓冲区地址指向这个共享内存区对象。

③ 服务器读入完毕时，使用一个信号量通知客户。

④ 客户将这些数据从该共享内存区对象写出到输出文件中。

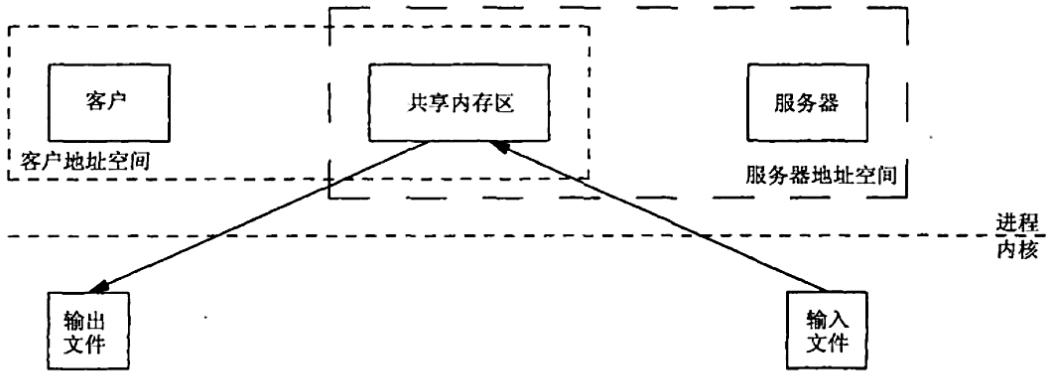


图12-2 使用共享内存区将文件数据从服务器复制到客户

本图中数据只复制两次：一次从输入文件到共享内存区，另一次从共享内存区到输出文件。我们画了一个包围客户和该共享内存区对象的虚框，又画了另一个包围服务器和该共享内存区对象的虚框，目的是强调**该共享内存区对象同时出现在客户和服务器的地址空间中**。

使用共享内存区所涉及的概念对于 Posix 接口和 System V 接口都类似。

mmap、munmap 和 msync 函数

mmap 函数把一个文件或一个 Posix 共享内存区对象映射到调用进程的地址空间。

```
#include <sys/mman.h>
// 返回：若成功则为被映射区的起始地址，若出错则为 MAP_FAILED
void *mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);
```

图 12-6 展示了内存映射文件的例子。

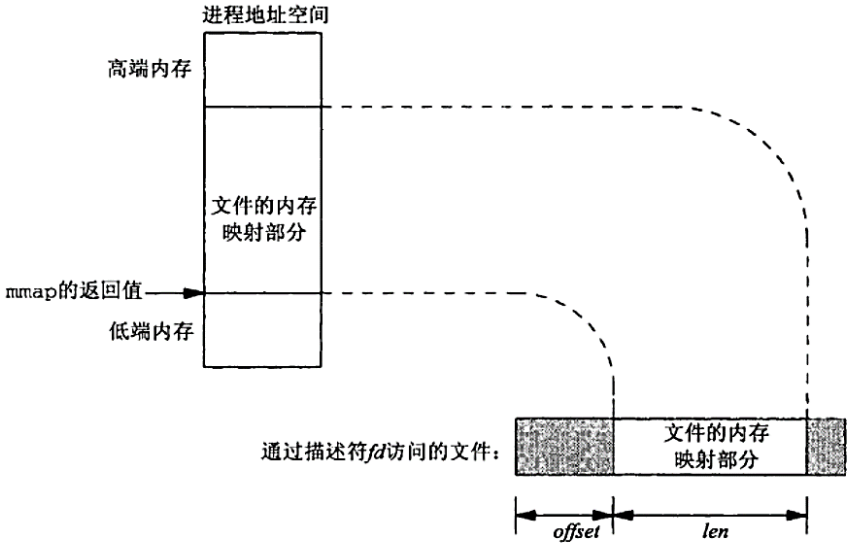


图12-6 内存映射文件的例子

使用 mmap 函数有三个目的：

1. 使用普通文件以提供内存映射 IO
2. 使用特殊文件以提供匿名内存映射
3. 使用 shm_open 以提供无亲缘关系进程间的 Posix 共享内存区

父子进程之间共享内存区的方法之一是：**父进程在调用 fork 前先指定 MAP_SHARED 调用 mmap。**

- Posix.1 保证父进程中的内存映射关系存留到子进程中，而且父进程所作的修改子进程能看到，反过来也一样。
- mmap 成功返回后，fd 参数可以关闭。该操作对于由 mmap 建立的映射关系没有影响。

munmap 从某个进程的地址空间删除一个映射关系。

```
#include <sys/mman.h>
// 返回：若成功则为 0，若出错则为-1
int munmap(void *addr, size_t len);
```

在图 12-6 中，内核的虚拟内存算法保持内存映射文件（一般在硬盘上）与内存映射区（在内存中）的同步，前提是它是一个 MAP_SHARED 内存区。

这就是说，如果我们修改了处于内存映射到某个文件的内存区中某个位置的内容，那么内核将在稍后某个时刻相应地更新文件。然而有时我们希望确信硬盘上的文件内容与内存映射区中的内容一致，于是调用 msync 来执行这种同步。

```
#include <sys/mman.h>
// 返回：若成功则为 0，若出错则为-1
int msync(void *addr, size_t len, int flags);
```

Posix 共享内存区

Posix.1 提供了两种在无亲缘关系进程间共享内存区的方法。

- **内存映射文件**（memory-mapped file）：由 open 函数打开，由 mmap 函数把得到的描述符映射到当前进程地址空间中的一个文件。
- **共享内存区对象**（shared-memory object）：由 shm_open 打开一个 Posix.1 IPC 名字（也许是在文件系统中的一个路径名），所返回的描述符由 mmap 函数映射到当前进程的地址空间。

这两种技术都需要调用 mmap，差别在于作为 mmap 的参数之一的描述符的获取手段：通过 open 还是 shm_open。

图 13-1 展示了这个差别。Posix 把两者合称为内存区对象（memory object）。

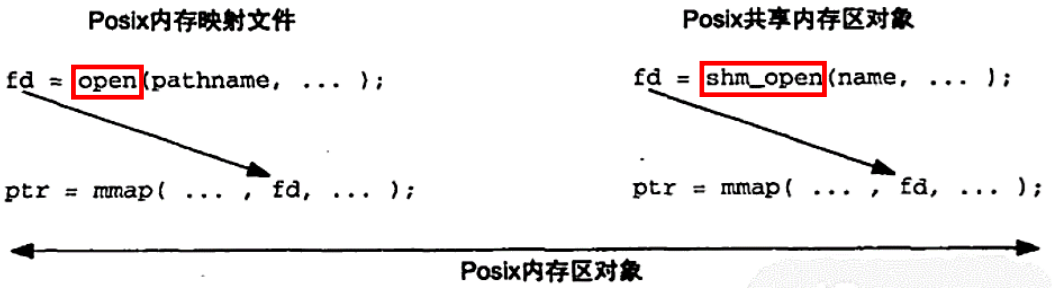


图13-1 Posix内存区对象：内存映射文件和共享内存区对象

shm_open 和 shm_unlink 函数

Posix 共享内存区涉及以下两个步骤要求。

- 指定一个名字参数调用 shm_open，以创建一个新的或打开一个已存在的共享内存区对象。传递给 shm_open 的名字参数随后由希望共享该内存区的任何其他进程使用。
- 调用 mmap 把这个共享内存区映射到调用进程的地址空间。

shm_open 的返回值是一个整数描述符，它随后用作 mmap 的第五个参数。

shm_unlink 函数删除一个共享内存区对象的名字。跟所有其他 unlink 函数一样，删除一个名字不会影响对于其底层支撑对象的现有引用，直到对于该对象的引用全部关闭为止。删除一个名字仅仅防止后续的 open、mq_open 或 sem_open 调用取得成功。

- 删除文件系统中一个路径名的 unlink，
- 删除一个 Posix 消息队列的 mq_unlink，
- 删除一个 Posix 有名信号量的 sem_unlink

```
#include <sys/mman.h>
// 返回：若成功则为非负描述符，若出错则为-1
int shm_open(const char *name, inf oflag, mode_t mode);
// 返回：若成功则为 0，若出错则为-1
int shm_unlink(const char *name);
```

fstruncate 和 fstat 函数

处理 mmap 的时候，普通文件或共享内存区对象的大小都可以通过调用 ftruncate 修改。

```
#include <unistd.h>
// 返回：若成功则为 0，若出错则为-1
int ftruncate(int fd, off_t length);
```

当打开一个已存在的共享内存区对象时，我们可调用 fstat 来获取有关该对象的信息。

```
#include <sys/types.h>
#include <sys/stat.h>
// 返回：若成功则为 0，若出错则为-1
int fstat(int fd, struct stat *buf);
```

7、套接字（socket）

套接口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同设备及其间的进程通信。

5. 生产者-消费者问题

- 《现代操作系统》“第 2 章 进程与线程”“2.3 进程间通信”“2.3.4 睡眠与唤醒”
- 经典并发同步模式：生产者-消费者设计模式：https://zhuanlan.zhihu.com/p/73442055

生产者-消费者（producer-consumer）问题，也称为有界缓冲区（bounded-buffer）问题。

场景描述：

- 两个进程共享一个公共的固定大小的缓冲区
- 生产者：将数据写入缓冲区
- 消费者：从缓冲区中读出数据

问题 1：缓冲区已满，但生产者还想向其中写入新数据

- 解决方法：让生产者睡眠，待消费者从缓冲区中读出部分数据后再唤醒生产者

问题 2：缓冲区为空，但消费者还想从中读出数据

- 解决方法：让消费者睡眠，待生产者向缓冲区中写入部分数据后再唤醒消费者

6. 线程之间的通信方式有哪些？

7. 进程调度方法详细介绍

8. 进程的执行过程是什么样的，执行一个进程需要做哪些工作

9. 进程的状态以及转换图

10. 进程之间的同步方式有哪些？

11. 线程是如何同步的（尤其是如果项目中用到了多线程，很大可能会结合讨论）

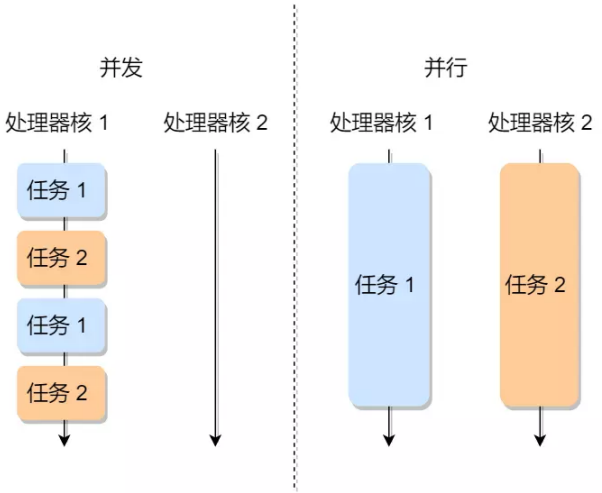
12. 同一个进程内的线程会共享什么资源？

13. 什么时候用多进程，什么时候用多线程

14. 并发和并行的区别

- 并发和并行的区别：http://c.biancheng.net/view/95.html
- 并发和并行的区别：https://segmentfault.com/a/1190000021945345
- 还在疑惑并发和并行？：https://laike9m.com/blog/huan-zai-yi-huo-bing-fa-he-bing-xing,61/

并发（concurrency）	并行（parallelism）
并发强调的是程序的结构，指能够让多个任务在逻辑上交织执行的程序设计	并行强调的是程序运行时的状态，指物理上同时执行
多个任务在重叠的时间段内以无特定顺序运行。	多个任务在逻辑上同时运行的情况，例如在多核处理器上。
在同一时间点，任务并不会同时运行。	在同一时间点，任务一定同时运行。
并发是指同时管理很多事情，这些事情可能只做了一半就被暂停去做别的事情了	并行的关键是同时做很多事情



15. 【热门】协程了解吗

16. 协程的底层是怎么实现的，怎么使用协程？

17. 操作系统的内存管理说一下

18. 实现一个 LRU 算法

参考“高频代码题解.docx”“0146 LRU 缓存机制”。

19. 死锁的必要条件和原因，死锁的检测和解除，死锁的避免和预防

- 面试杂谈 - 死锁的四大条件与处理策略：https://cloud.tencent.com/developer/article/1493418
- 死锁产生的原因及四个必要条件：https://zhuanlan.zhihu.com/p/25677118
- 死锁的四个必要条件：https://blog.csdn.net/rabbit_in_android/article/details/50530960

死锁有 4 个必要条件

只要系统发生死锁，这些条件必然成立；而只要任何一个条件不满足，就不会发生死锁。

1. **互斥**：一个资源每次只能被一个进程使用。
2. **占有且等待（请求与保持）**：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
3. **不可剥夺（不可抢占）**：进程已获得的资源，在使用完之前，不能强行剥夺。
4. **循环等待**：若干进程之间形成一种头尾相接的循环等待资源关系。

死锁产生的原因

1. 系统资源不足
2. 进程运行推进的顺序不合适

3. 资源分配不当

死锁的检测和解除（恢复）

死锁检测和解除有可能使系统获得较好的资源利用率和吞吐量，但在实现上难度也最大。

- **死锁检测**：不须实现采取任何限制性措施，而是允许系统在运行过程发生死锁，但可通过系统设置的检测机构及时检测出死锁的发生，并精确地确定于死锁相关的进程和资源，然后采取适当的措施，从系统中将已发生的死锁清除掉。
- **死锁解除（恢复）**：与死锁检测相配套的一种措施。当检测到系统中已发生死锁，需将进程从死锁状态中解脱出来。常用方法有：
  - 利用抢占恢复：临时将某个资源从它的当前所有者那里转移给另一个进程
  - 利用回滚恢复：周期性地对进程进行检查点检查，将该进程复位到一个更早的状态
  - 通过杀死进程恢复：一种方法是杀掉环中的一个进程。如果走运的话，其他进程将可以继续。如果这样做行不通的话，就需要继续杀死别的进程直到打破死锁环。另一种方法是选一个环外的进程作为牺牲品以释放该进程的资源。

死锁的避免

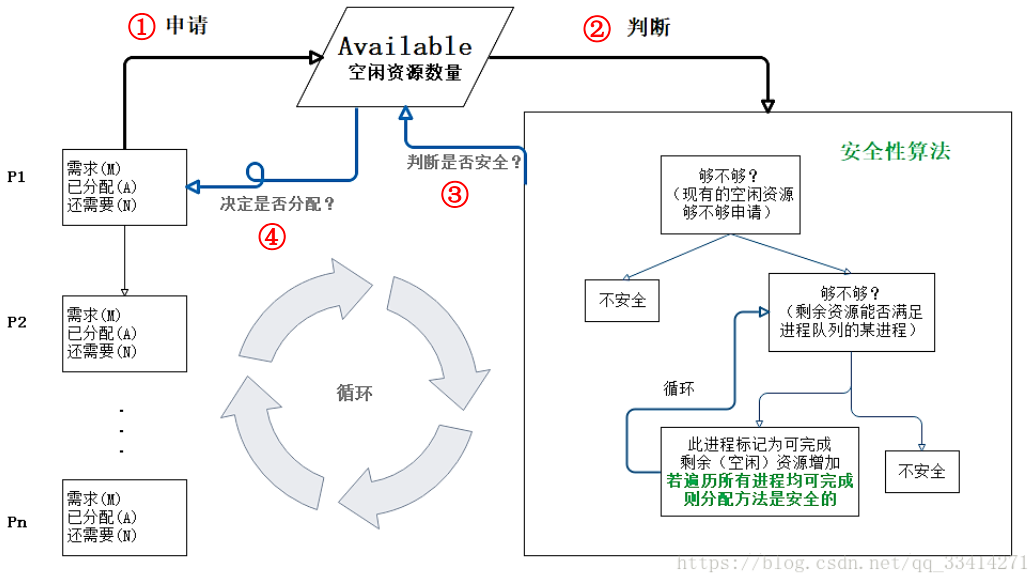
- 一句话+一张图说清楚——银行家算法：https://blog.csdn.net/qq_33414271/article/details/80245715

资源轨迹图；安全状态和不安全状态。

Dijkstra 提出了一种能够避免死锁的调度算法：银行家算法。

- 算法判断满足请求后是否会进入不安全状态。如果不安全，就拒绝请求；如果仍然安全，就予以分配。

死锁的避免允许前三个必要条件，但通过明智的选择，确保永远不会到达死锁点，因此死锁避免比死锁预防允许更多的并发。



银行家算法

死锁的预防

通过设置某些限制条件，破坏死锁的四个条件中的一个或几个条件，来预防发生死锁。但由于所施加的限制条件往往太严格，因而导致系统资源利用率和系统吞吐量降低。

- 破坏**互斥**条件：如果资源不被一个进程独占，那么死锁肯定不会产生。
- 破坏**占有且等待**条件：只要禁止已持有资源的进程再等待其他资源便可以消除死锁。
  - 一种方法是，规定所有进程在开始执行前请求所需的全部资源。如果所需的全部资源可用，那么就将它们分配给这个进程，于是该进程肯定能够运行结束。如果有一个或多个资派正被使用，那么就不进行分配，进程等待。
  - 另一种方法是，要求当一个进程请求资源时，先暂时释放其当前占用的所有资源，然后再尝试一次获得所需的全部资源。
- 破坏**不可剥夺**条件：虚拟化。
- 破坏**循环等待**条件：
  - 一种方法是，保证每一个进程在任何时刻只能占用一个资源，如果要请求另外一个资源，它必须先释放第一个资源。
  - 另一种方法是，将所有资源统一编号，进程可以在任何时刻提出资源请求，但是所有请求必须按照资源编号的顺序（升序）提出。

20. 什么是饥饿

21. 如果要你实现一个 mutex 互斥锁你要怎么实现？

22. 文件读写使用的系统调用

23. 孤儿进程和僵尸进程分别是什么，怎么形成的？

24. 说一下 PCB / 说一下进程地址空间

25. 内核空间和用户空间是怎样区分的

26. 异常和中断的区别

27. 一般情况下在 Linux / Windows 平台下栈空间的大小

28. 虚拟内存的了解，页表，TLB

29. 服务器高并发的解决方案

30. 在执行 malloc 申请内存的时候，操作系统是怎么做的？/ 内存分配的原理说一下 / malloc 函数底层是怎么实现的？/ 进程是怎么分配内存的？

31. 什么是字节序？怎么判断是大端还是小端？有什么用？

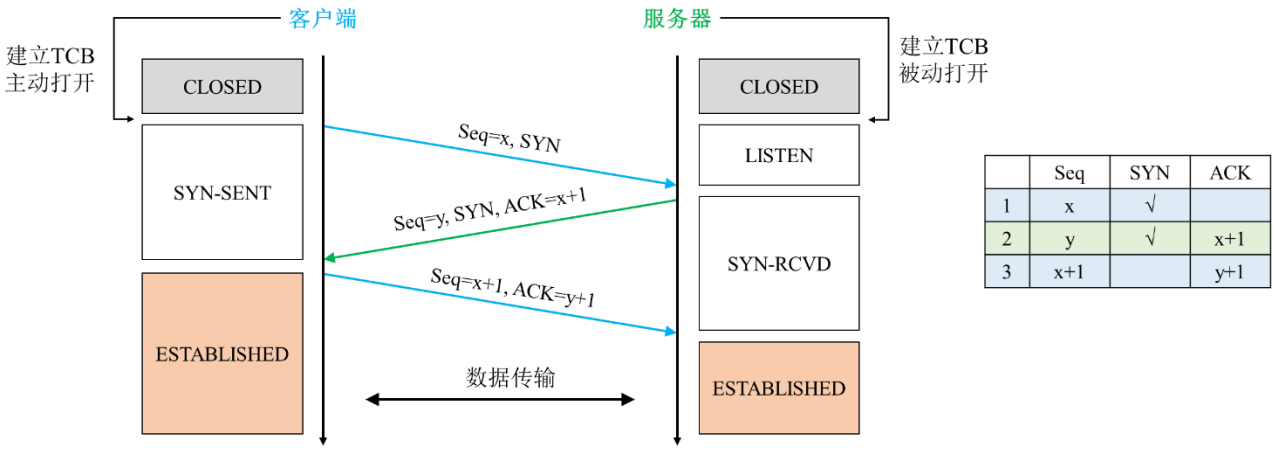
计算机网络

1. 建立 TCP 服务器的各个系统调用
2. 继上一题，说明 socket 网络编程有哪些系统调用？其中 close 是一次就能直接关闭的吗，半关闭状态是怎么产生的？
3. 对路由协议的了解与介绍
4. UDP 如何实现可靠传输
5. TCP 和 UDP 的区别
6. TCP 和 UDP 相关的协议与端口号
7. TCP（UDP，IP）等首部的认识（http 请求报文构成）
8. 网页解析的过程与实现方法
9. 【热门】在浏览器中输入 URL 后执行的全部过程
10. 网络层分片的原因与具体实现
11. 【热门】TCP 的三次握手与四次挥手的详细介绍

- 《TCP/IP 详解 卷 1》“第 13 章 TCP 连接管理”：“13.2 TCP 连接的建立与终止”，第 443~451 页
- 《Unix 网络编程 卷 1》“第 2 章 传输层：TCP、UDP 和 SCTP”：“2.6 TCP 连接的建立和终止”，第 45~50 页

- 《Linux 高性能服务器编程》“第 3 章 TCP 协议详解”：“3.3 TCP 连接的建立和关闭”，第 55~62 页
- 两张动图-彻底明白 TCP 的三次握手与四次挥手：**https://blog.csdn.net/qzcsu/article/details/72861891**
- 面试官，不要再问我三次握手和四次挥手：https://juejin.cn/post/6844903958624878606
- TCP 协议 • 笔试面试知识整理：https://hit-alibaba.github.io/interview/basic/network/TCP.html

TCP 三次握手



	Seq	SYN	ACK
1	x	√	
2	y	√	x+1
3	x+1		y+1

TCP 建立连接：三次握手

TCP 建立连接过程为三次握手。

初始时客户端和服务器均处于 CLOSED（关闭）状态。

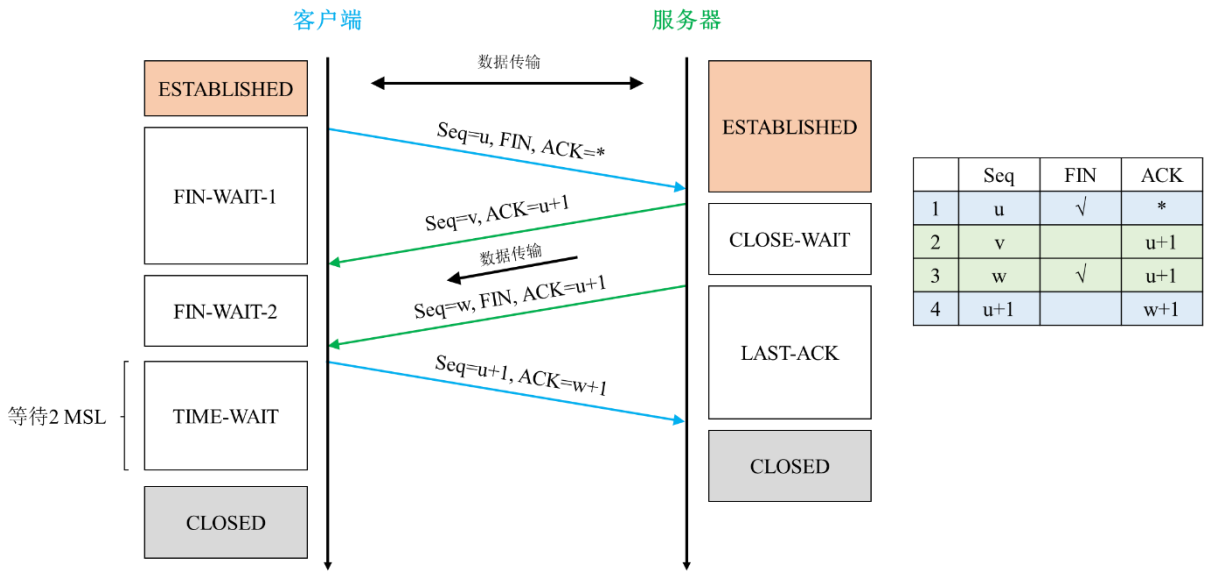
服务器进程创建传输控制块 TCB，等待接收客户端进程的连接请求，进入 LISTEN（监听）状态。

1. 客户端进程创建传输控制块 TCB，向服务器发出连接请求报文，进入 SYN-SENT（同步发送）状态
   - 客户端发送①客户端初始序列号 Seq=x；②同步序列报文段 SYN
2. 服务器收到请求报文，若同意连接，发出确认报文，进入 SYN-RCVD（同步收到）状态
   - 服务器发送①服务器初始序列号 Seq=y；②SYN；③确认号 ACK=x+1
3. 客户端收到确认报文，再给服务器发送确认报文，进入 ESTABLISHED（连接建立）状态
   - 客户端返回①Seq=x+1；②ACK=y+1
   - 服务器收到客户端的确认报文后，也进入 ESTABLISHED（连接建立）状态，此时双方就可以通信了。

报文速记：

- 三次都携带本方序列号 Seq
- 前两次有 SYN，后两次有 ACK
- 确认号 ACK 都是对方上次的序列号 Seq+1

TCP 四次挥手



TCP 关闭连接：四次挥手

TCP 关闭连接的过程为四次挥手。

初始时客户端和服务器都处于 ESTABLISHED（连接建立）状态。

1. 客户端停止发送数据，发送连接释放报文，进入 FIN-WAIT-1（终止等待 1）状态
   - 客户端发送①客户端序列号 seq=u；②终止报文段 FIN；③确认号 ack=*。
2. 服务器收到连接释放报文，发送确认报文，进入 CLOSE-WAIT（关闭等待）状态
   - 服务器发送①服务器序列号 seq=v；②确认号 ack=u+1。
   - 此时客户端往服务器的方向已释放，连接处于半关闭状态，客户端没有数据要发送给服务器，但是如果服务器要发送数据，客户端仍要接收服务器最后发送的数据。
   - 客户端收到服务器的确认报文，进入 FIN-WAIT-2（终止等待 2）状态，等待服务器发送连接释放报文，在这段时间内还要继续接收服务器发送的数据。
3. 服务器发送完最后的数据，向客户端发送连接释放报文，进入 LAST-ACK（最后确认）状态
   - 服务器发送①服务器序列号 seq=w；②终止报文段 FIN；③确认号 ack=u+1。
4. 客户端收到服务器的连接释放报文，回复确认报文，进入 TIME-WAIT（时间等待）状态
   - 客户端发送①客户端序列号 seq=u+1；②确认号 ack=w+1。
   - 此时 TCP 连接还没有释放，需经过 2*MSL（最大报文段寿命）时间后，客户端撤销了相应的传输控制块 TCB，才进入 CLOSED（关闭）状态。
   - 服务器一旦收到该确认报文，立刻进入 CLOSED（关闭）状态，撤销 TCB 后，此次 TCP 连接结束。

报文速记：

- 四次都携带本方序列号 Seq 和确认号 ACK

- 第 1、3 次有 FIN
- ACK 都是对方上次的序列号 Seq+1

12. TCP 握手以及每一次握手客户端和服务器端处于哪个状态

见上题。

13. 为什么使用三次握手，两次握手可不可以？

- TCP 为什么是三次握手，而不是两次或四次？ - 知乎：https://www.zhihu.com/question/24853633
- https://www.zhihu.com/question/24853633/answer/1279799682
- 近 40 张图解被问千百遍的 TCP 三次握手和四次挥手面试题：https://mp.weixin.qq.com/s/tH8RFmjrveOmgLvk9hmrkw
- https://www.zhihu.com/question/24853633/answer/573627478
- RFC 793: Transmission Control Protocol：https://www.rfc-editor.org/rfc/rfc793.html#page-32
- TCP 为什么三次握手而不是两次握手（正解版）：https://blog.csdn.net/lengxiao1993/article/details/82771768

两种常见的说法是：

- 为了防止已失效的连接请求报文段突然又传送到了服务端，因而产生错误。（谢希仁《计算机网络》）
- 为了解决“网络中存在延迟的重复分组”问题。（Tanenbaum《计算机网络》）

在 RFC 793 的第 28、29、32 页回答了三次握手的原因：

> [p.28] A three way handshake is necessary **because sequence numbers are not tied to a global clock in the network, and TCPs may have different mechanisms for picking the ISN's.** The receiver of the first SYN has no way of knowing whether the segment was an old delayed one or not, unless it remembers the last sequence number used on the connection (which is not always possible), and so it must ask the sender to verify this SYN. The three way handshake and the advantages of a clock-driven scheme are discussed in [3].
>
> [p.29] The three-way handshake **reduces the possibility of false connections.** It is the implementation of **a trade-off between memory and messages to provide for this checking.**
>
> [p.32] The principle reason for the three-way handshake is **to prevent old duplicate connection initiations from causing confusion.**
>
> To deal with this, a special control message, reset, has been devised.
>
> - If the receiving TCP is in a non-synchronized state (i.e., SYN-SENT, SYN-RECEIVED), it returns to LISTEN on receiving an acceptable reset.
> - If the TCP is in one of the synchronized states (ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK, TIME-WAIT), it aborts the connection and informs its user.
>
> We discuss this latter under "half-open" connections below.

两次握手的过程：

1. A 发送①序列号 Seq=a；②同步信号 SYN；
2. B 发送①序列号 Seq=b；②同步信号 SYN；③确认信号 ACK

（如果四次握手，那么实际上是将三次握手中的 ACK 和 SYN 拆开发送。）

这时我们就能回答为什么使用三次握手：

1. 防止旧的重复连接初始化造成混乱（主要）

- 如果一个旧的 SYN 报文比最新的 SYN 报文早到达了服务端，那么此时服务端会返回一个 SYN + ACK 报文给客户端
- 如果是两次握手，无法判断当前连接是否是历史连接
- 如果是三次握手，客户端（发送方）在准备发送第三次报文时，有足够的上下文来判断当前连接是否是历史连接：
  - 如果是历史连接（序列号过期或超时），则第三次握手发送 RST 报文，中止历史连接
  - 如果不是历史连接，则第三次握手发送 ACK 报文，通信双方成功建立连接

2. 同步双方的序列号

TCP 连接握手，实际上握的是通信双方数据原点的序列号。TCP 需要序列号 Seq 来做可靠重传或接收，而避免连接复用时无法分辨出 Seq 是延迟的或是旧连接的 Seq，因此需要三次握手来约定确定双方的 ISN（初始序列号）。

TCP 协议的通信双方都必须维护一个序列号，序列号是可靠传输的一个关键因素，它的作用如下：

- 接收方可以去除重复数据
- 接收方可以根据数据包的序列号按序接收
- 可以标识发送出去的数据包中，哪些是已经被对方收到的。

可见序列号在 TCP 连接中具有重要作用：

- 当客户端发送携带初始序列号的 SYN 报文时，需要服务端返回一个 ACK 应答报文，表示客户端的 SYN 报文已被服务端成功接收；
- 当服务端发送初始序列号给客户端的时候，依然也要得到客户端的应答回应，这样一来一回，才能确保双方的初始序列号能被可靠的同步。

两次握手只保证了一方的初始序列号能被对方成功接收，没办法保证双方的初始序列号都能被确认接收。具体来说，客户端和服务器对客户端的序列号达成了共识，但是并没有对服务器的序列号达成共识。也即，服务器无法知道客户端是否已经接收到自己的同步信号，如果这个同步信号丢失了，客户端和服务器将无法对服务器的初始序列号达成一致。

拓展内容：

- TCP 的设计者将 SYN 这个同步标志位 SYN 设计成占用一个字节的编号（FIN 标志位也是），既然是一个字节的数据，按照 TCP 对有数据的 TCP 字段必须确认的原则，所以在这里客户端必须给服务器一个确认，以确认客户端已经接收到服务器的同步信号。

- SYN 位字段会消耗一个序列号，消耗一个序列号也意味着使用重传进行可靠传输。故 SYN 和应用程序字节（还有 FIN）是被可靠传输的。不消耗序列号的 ACK 则不是。

  （《TCP/IP 详解 卷 1》第 439 页）

3. 避免资源浪费

（其实第 3 点我认为和第 1 点是一个意思。）

如果只有两次握手，当客户端的 SYN 请求连接在网络中阻塞，客户端没有接收到服务器的 ACK 报文，就会重新发送 SYN，由于没有第三次握手，服务器不清楚客户端是否收到了自己发送的建立连接的 ACK 确认信号，所以每收到一个 SYN 就只能先主动建立一个连接。因此如果客户端的 SYN 阻塞了，就会重复发送多次 SYN 报文，那么服务器在收到请求后就会建立多个冗余的无效链接，造成不必要的资源浪费。

顺便补充一下三次握手中丢包的情形：

1. 第一个包，即 A 发给 B 的 SYN 中途被丢，没有到达 B
- A 会周期性超时重传，直到收到 B 的确认
2. 第二个包，即 B 发给 A 的 SYN +ACK 中途被丢，没有到达 A
- B 会周期性超时重传，直到收到 A 的确认
3. 第三个包，即 A 发给 B 的 ACK 中途被丢，没有到达 B
- A 发完 ACK，单方面认为 TCP 为 ESTABLISHED 状态，而 B 显然认为 TCP 为 Active 状态：
  - 假定此时双方都没有数据发送，B 会周期性超时重传，直到收到 A 的确认，收到之后 B 的 TCP 连接也为 ESTABLISHED 状态，双向可以发包。
  - 假定此时 A 有数据发送，B 收到 A 的数据+ ACK，自然会切换为 ESTABLISHED 状态，并接受 A 的数据。
  - 假定 B 有数据发送，则数据事实上是发送不了的，而是一直周期性超时重传 SYN + ACK，直到收到 A 的 ACK 后才能发送数据。

14. 四次挥手可以改成三次握手吗

TCP 为什么是四次挥手，而不是三次？：https://www.zhihu.com/question/63264012

可以，将第二次和第三次的 ACK 和 FIN 合起来发送。

通常将这两次分开的原因是，服务端可能还有数据要发送，因此先发送 ACK 报文确认断开连接的请求，这样客户端就不好因为没有收到应答而继续重复发送断开连接的请求。

服务端发送完数据后，再发送 FIN 报文，保证数据通信的完整可靠。服务端之后进入 LAST-ACK 状态。

15. TCP 三次握手时的第一次的 seq 序号是怎样产生的

16. TIME_WAIT 的意义（为什么要等于 2MSL）

17. 服务器出现大量 close_wait 连接的原因以及解决方法

close_wait 状态是在 TCP 四次挥手的时候服务端收到 FIN 但是没有发送自己的 FIN 时出现的，出现大量 close_wait 状态的原因有两种：

- 服务器内部业务处理占用了过多时间，都没能处理完业务；或者还有数据需要发送；或者服务器的业务逻辑有问题，没有执行 close() 方法
- 服务器的父进程派生出子进程，子进程继承了 socket，收到 FIN 的时候子进程处理了该信号，但父进程没有处理，导致 socket 的引用不为 0 无法回收

解决方法：

- 停止应用程序
- 修正代码中的 bug

18. 客户端出现大量 time_wait 的原因以及解决方法

- 服务产生大量 TIME_WAIT 如何解决：https://www.jianshu.com/p/41f7e468f312
- 大量的 TIME_WAIT 状态连接怎么处理？：https://cloud.tencent.com/developer/article/1675933

客户端的 time_wait 的状态出现在 TCP 四次握手中客户端收到服务端的 FIN 信号后，需要等待 2 MSL 时间才变为 CLOSED，在这 2 MSL 时间内都为 TIME-WAIT 状态。

通常原因：

- 大量短连接的存在：后台服务器经常会调用客户端的 redis、mysql 和其他 HTTP 和 RPC 服务。

解决方法：

- 客户端+服务端：
  - HTTP 请求头部的 connection 设置为 keep-alive（而不是 close），使连接保活一段时间
- 客户端：
  - 缩减 time_wait 的时间（比如降低 MSL 时间或者使用 1 MSL）
- 服务端：
  - 允许 time_wait 状态的 socket 重用；

- 通常 redis 等客户端会有连接池，合理设置相关的连接调用参数（连接数、连接重用时间、连接空闲数）

19. 长连接和短连接的区别

- TCP 的长连接和短连接：https://developer.aliyun.com/article/37987
- HTTP 长连接和短连接：https://www.cnblogs.com/0201zcr/p/4694945.html
- http 的长连接和短连接（史上最通俗！）：https://www.jianshu.com/p/3fc3646fad80
- 网络连接中的长连接和短链接是什么意思？：https://www.zhihu.com/question/22677800

定义：

- 短连接：一次读写完成后，任何一方都可以发起 close 操作，通常由客户端发起
- 长连接：一次读写完成后，连接保持，不关闭（服务端保活机制）

优缺点：

- 长连接：可以省去较多的 TCP 建立和关闭的操作，减少浪费，节约时间。
  - 缺点：随着客户端连接的增加，服务器的资源开销也越来越大
    - 采取相应的缓解策略：关闭长时间无读写事件的连接，限制客户端的最大长连接数
- 短连接：管理较为简单，存在的连接都是有用的连接，不需要额外的控制手段。
  - 缺点：如果客户请求频繁，将在 TCP 的建立和关闭操作上浪费时间和带宽。

使用场景：

- 长连接：多用于操作频繁，点对点通讯，且连接数不能太多的情况
  - 数据库的连接用长连接，如果用短连接，频繁的通信会造成 socket 错误，并且频繁创建 socket 也会浪费资源。
- 短连接：多用于操作较少，连接数很多的情况
  - Web 网站的 HTTP 服务用短连接，如果用长连接，成千上万的连接会占用大量资源

长短轮询和长短连接的区别：

- 决定的方式不同：
  - 一个 TCP 连接是否为长连接，是通过设置 HTTP 的 Connection Header 来决定的，而且是需要两边都设置才有效
  - 一种轮询方式是否为长轮询，是根据服务端的处理方式来决定的，与客户端无关
- 实现的方式不同：
  - 连接的长短是通过协议来规定和实现的
  - 轮询的长短是服务器通过编程的方式手动挂起请求来实现的

20. 超时重传机制（不太高频）

21. TCP 怎么保证可靠性？

22. 流量控制的介绍，采用滑动窗口会有什么问题（死锁可能，糊涂窗口综合征）？

- 计算机网络微课堂 p60 5.4 TCP 的流量控制：https://www.bilibili.com/video/BV1c4411d7jb?p=60

23. TCP 滑动窗口协议

24. 拥塞控制和流量控制的区别

25. 【热门】TCP 拥塞控制，算法名字？

- 《TCP/IP 详解 卷 1：协议》“第 16 章 TCP 拥塞控制”，第 537~583 页
- 计算机网络微课堂 p61 5.5 TCP 的拥塞控制：https://www.bilibili.com/video/BV1c4411d7jb?p=61
- TCP 流量控制、拥塞控制：https://zhuanlan.zhihu.com/p/37379780

37. **消息摘要算法列举一下，介绍 MD5 算法，为什么 MD5 是不可逆的，有什么办法可以加强消息摘要算法的安全性让它不那么容易被破解呢？**

38. **单条记录高并发访问的优化**

39. **介绍一下 ping 的过程，分别用到了哪些协议**

40. **一个 ip 配置多个域名，靠什么识别？**

41. **服务器攻击（DDos 攻击）**

42. **DNS 的工作过程和原理**

43. **OSA 七层协议和五层协议，分别有哪些**

44. **IP 寻址和 MAC 寻址有什么不同，怎么实现的**

数据库

1. **关系型和非关系型数据库的区别（低频）**

2. **什么是非关系型数据库（低频）**

3. **说一下 MySQL 执行一条查询语句的内部执行过程？**

4. **数据库的索引类型**

- mysql 四种索引类型：https://my.oschina.net/zhangqie/blog/1618391
- 聚集索引和非聚集索引（整理）：https://www.cnblogs.com/aspnethot/articles/1504082.html

逻辑分类：

- 主键索引：在关系表中定义主键时会自动创建主键索引。每张表中的主键索引只能有一个，要求主键中的每个值都唯一，即不可重复，也不能有空值。

```
alter table table_name add primary key (col1, col2, ...)
```

- 唯一索引：数据列不能有重复，但可以有空值。一张表可以有多个唯一索引，但是每个唯一索引只能有一列。（比如身份证、卡号）

```
alter table table_name add unique (col1, col2, ...)
```

- 普通索引：可以重复，可以为空值。一张表可以有多个普通索引。

```
alter table table_name add index (col1, col2, ...)
```

- 全文索引：可以加快模糊查询。可以在 varchar、char、text 类型的列上创建。

```
alter table table_name add fulltext (col1, col2, ...)
```

物理分类：

- 聚集索引 / 聚簇索引 / 主索引：数据在物理存储中的顺序和在索引中的逻辑顺序相同。
  - 比如以 ID 建立聚集索引，数据库中 ID 从小到大排列，物理存储中该数据的内存地址也从小到大排列。
  - 一般是表中的主键索引，若没有主键索引，则以第一个非空的唯一索引作为聚集索引
  - 一张表只能有一个聚集索引，但是该索引可以有多个列（组合索引）
  - 聚集索引对于经常要搜索范围值的列特别有效，找到聚集索引中包含第一个值的行后，可以确保后续索引值的行在物理上相邻
  - 当索引值唯一时，使用聚集索引查找特定的行也很有效率
- 非聚集索引 / 非聚簇索引 / 辅助索引：数据在物理存储中的顺序和在索引中的逻辑顺序不同。
  - 由于非聚集索引无法定位数据所在行，因此需要扫描两遍索引树。第一遍扫描非聚集索引的索引树，确定该数据的主键 ID，第二遍到主键索引（聚集索引）中寻找对应数据
  - 聚集索引的叶结点就是数据结点，而非聚集索引的叶结点仍然是索引结点，只不过有一个指针指向对应的数据块

5. MySQL 怎么建立索引，怎么建立主键索引，怎么删除索引

- 建立索引：alter table add 或者 create index。
- 删除索引：alter table drop 或者 drop index。

```
1  alter table table_name add primary key (col1, col2, ...) # 添加主键索引
2  alter table table_name add index (col1, col2, ...)       # 添加普通索引
3  alter table table_name add unique (col1, col2, ...)      # 添加唯一索引
4
5  create index index_name on table_name (col1, col2, ...)   # 创建普通索引
6  create unique index_name on table_name (col1, col2, ...)  # 创建唯一索引
7
```

```
8   alter table table_name drop index index_name  # 删除普通索引
9   alter table table_name drop primary key        # 删除主键索引
10
11  drop index index_name on table_name  # 删除索引
```

6. 主键、外键和索引的区别

- 关于数据库主键和外键（终于弄懂啦）：https://blog.csdn.net/harbor1981/article/details/53449435
- MySQL 数据库的主键和外键详解 3：https://zhuanlan.zhihu.com/p/114834741

	主键	外键	索引
定义	主键是表中一列或多列的组合，其值能唯一地标识表中的每一行。	若一个公共关键字在某个关系中是主关键字，则该公共关键字称为另一个关系的外键。[1]	见上上题
示例	自增整数类型 全局唯一 GUID 类型		
作用	保证表的实体完整性 加快数据库的操作速度	保持数据一致性和完整性 和其他表建立联系	索引用于提高查询排序速度
个数	一张表只能有一个主键	一张表可以有多个外键	一张表可以有多个唯一索引

[1] 以该公共关键字作主关键字的表被称为主表，具有此外键的表被称为主表的从表。外键又称作外关键字。

外键举例：

student 表

id	class_id	name	age	sex
1	1	张三	16	男
2	1	李四	17	男
3	2	王五	18	女
4	1	赵六	18	男
(Auto)	(NULL)	(NULL)	(NULL)	(NULL)

class 表

id	class_name
1	高三一班
2	高三二班
(Auto)	(NULL)

- 在 student 表中，通过 class_id 字段可以把数据与另一张表 class 关联起来，这里的 class_id 就是外键。
- 由于一个班级可以有多个学生，在关系模型中，这两个表的关系称为“一对多”，即 class 表的一个记录可以对应 student 表的多个记录。
- 为了表达这种一对多的关系，我们需要在 student 表中加入一列 class_id，让它的值与 class 表的某条记录相对应。这样，我们就可以根据 class_id 这个列直接定位出 student 表的一条记录应该对应到 class 的哪条记录。

外键并不是通过列名实现的，而是通过定义外键约束实现的：

```
1   // 定义外键约束
2   ALTER TABLE student
3   ADD CONSTRAINT fk_class_id  // 外键约束的名称 fk_class_id 可以任意
4   FOREIGN KEY (class_id)      // 指定 class_id 作为外键
5   REFERENCES class (id);      // 指定这个外键关联到 class 表的 id 列（即 class 表的主键）
6
```

```
// 删除外键约束
ALTER TABLE student
DROP FOREIGN KEY fk_class_id;
```

由于外键约束会降低数据库的性能，大部分互联网应用程序为了追求速度，并不设置外键约束，而是仅靠应用程序自身来保证逻辑的正确性。这种情况下，class_id 仅仅是一个普通的列，只是它起到了外键的作用而已。

7. 【热门】索引的优缺点，什么时候使用索引，什么时候不能使用索引

- 浅谈索引的优缺点和建立索引的原则：**https://www.jianshu.com/p/107e5bdf4148**
- 数据库中索引的优缺点（转）：https://www.cnblogs.com/guojin705/archive/2011/06/21/2086429.html
- 数据库的索引的优缺点：https://programtip.com/zh/art-70024

索引的优点：

- 可以大大加快数据的检索速度，这也是创建索引的最主要的原因
- 通过创建唯一索引，可以保证数据库表中每一行数据的唯一性
- 可以加速表和表之间的连接，特别是在实现数据的参照完整性方面很有意义
- 可以减少分组和排序子句进行数据检索的时间
- 可以在查询过程中使用优化隐藏器，提高系统性能

索引的缺点：

- 创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加
- 索引需要占用物理空间，除了数据表占数据空间之外，每个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间会更大
- 对表中的数据进行增加、删除和修改时，索引也要动态维护，降低了数据的维护速度

什么时候使用索引：

- 经常需要搜索的列：可以加快搜索的速度
- 作为主键的列：强制该列的唯一性和组织表中数据的排列结构
- 经常用在连接的列：这些列主要是一些外键，可以加快连接的速度
- 经常需要根据范围进行搜索的列：因为索引已经排序，其指定的范围是连续的
- 经常需要排序的列：因为索引已经排序，可以加快排序查询速度
- 经常使用在 WHERE 子句中的列：加快条件的判断速度

什么时候不能使用索引：

- 在查询中很少使用或者参考的列：因为这些列很少用到，有无索引并不能提高查询速度。相反，由于增加了索引，反而降低了系统的维护速度，增大了空间需求。

- 只有很少数据值的列：由于这些列的取值很少，例如人事表的性别列，在查询的结果中，结果集的数据行占了表中数据行的很大比例，即需要在表中搜索的数据行的比例很大，增加索引并不能明显加快检索速度。
- 定义为 text, image 和 bit 数据类型的列：因为这些列要么数据量很大，要么取值很少。
- 当修改性能远远大于检索性能时：因为修改性能和检索性能是互相矛盾的。当增加索引时，会提高检索性能，但是会降低修改性能；当减少索引时，会提高修改性能，降低检索性能。因此，当修改性能远远大于检索性能时，不应该创建索引。

8. 【热门】索引的底层实现

- 深入理解 MySQL 索引底层实现原理：https://zhuanlan.zhihu.com/p/77383599

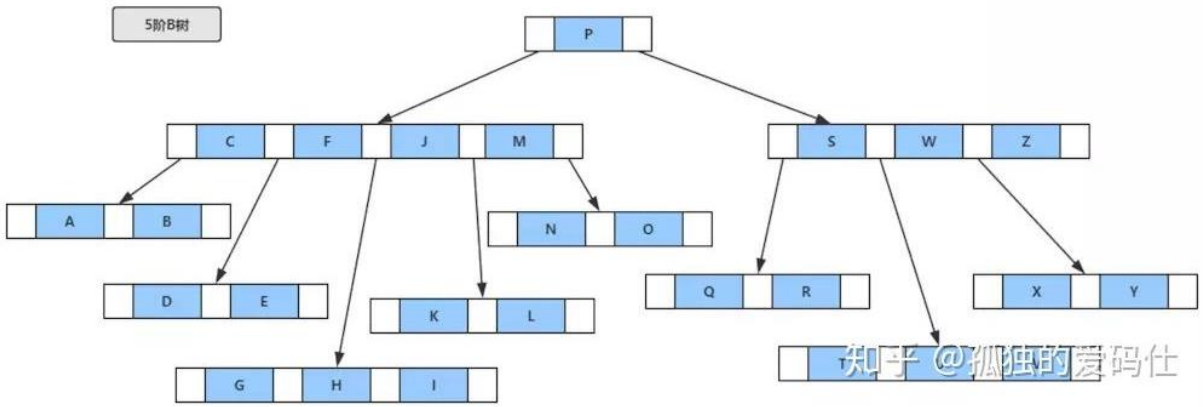
索引是数据结构

MySQL 官方对索引的定义为：索引（Index）是帮助 MySQL 高效获取数据的数据结构。提取句子主干，就可以得到索引的本质：**索引是数据结构。**

- 数据库查询是数据库的最主要功能之一。我们都希望查询数据的速度能尽可能的快，因此数据库系统的设计者会从查询算法的角度进行优化。
- 最基本的查询算法当然是顺序查找，这种复杂度为 O(n)的算法在数据量很大时显然是糟糕的，好在计算机科学的发展提供了很多更优秀的查找算法，例如二分查找、二叉树查找等。
- 每种查找算法都只能应用于特定的数据结构之上（例如二分查找要求被检索数据有序，而二叉树查找只能应用于二叉查找树上），但是数据本身的组织结构不可能完全满足各种数据结构（例如理论上不可能同时将两列都按顺序进行组织）
- 所以，在数据之外，数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构上实现高级查找算法。
- 这种数据结构，就是索引。

B 树

B 树事实上是一种平衡的多叉查找树，也就是说最多可以开 m 个叉（m>=2），称之为 m 阶 B 树，下图是一棵 5 阶 B 树。



总的来说，m 阶 B 树满足以下条件：

- 每个节点最多有 m 棵子树。
- 若根节点不是叶节点，则至少有 2 个子节点
  - 极端情况下一棵树就一个根节点，同时是根、叶、树
- 非根非叶的节点至少有 $\lceil \frac{m}{2} \rceil$ 个子树，图中 5 阶 B 树，每个节点至少有 3 个子树，也就是至少有 3 个叉。
- 有 n 个子节点的非叶节点有 n-1 个键
  - 非叶节点中的信息包括 $[n, A_1, K_1, A_2, K_2, \cdots, K_{n-1}, A_{n-1}]$，其中 n 表示该节点中保存的节点个数，K 为关键字且 $K_i < k_{i+1}$，A 为指向子树的根节点的指针。
- 所有叶子节点都在同一层
  - 换句话说，从根到叶子的每一条路径都有相同的长度
  - 这些叶子节点包含的指针为空，表示找不到指定的值

B 树的查询过程和二叉排序树比较类似，从根节点依次比较每个结点，因为每个节点中的关键字和左右子树都是有序的，所以只要比较节点中的关键字，或者沿着指针就能很快地找到指定的关键字，如果查找失败，则会返回叶子节点，即空指针。

- B 树搜索的伪代码如下：

```
1  BTree_Search(node, key) {
2      if(node == null) return null;
3      foreach(node.key) {
4          if(node.key[i] == key) return node.data[i];
5           if(node.key[i] > key)  return BTree_Search(point[i]->node);
6      }
7      return BTree_Search(point[i+1]->node);
8  }
9  data = BTree_Search(root, my_key);
```

B 树的特点：

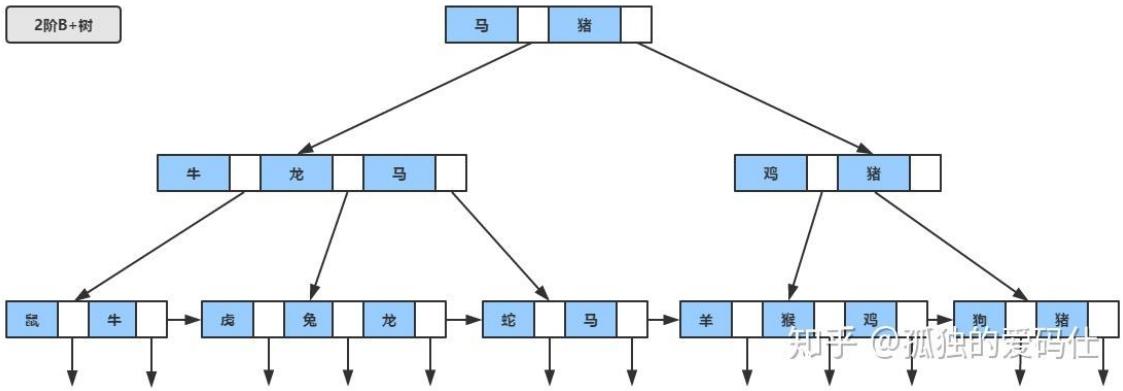
- 关键字集合分布在整颗树中。
- 任何一个关键字出现且只出现在一个节点中。
- 搜索有可能在非叶子节点结束。
- 其搜索性能等价于在关键字集合内做一次二分查找。
- B 树在插入删除新的数据记录会破坏 B 树的性质，因为在插入删除时，需要对树进行一个分裂、合并、转移等操作以保持 B 树性质。

B+树

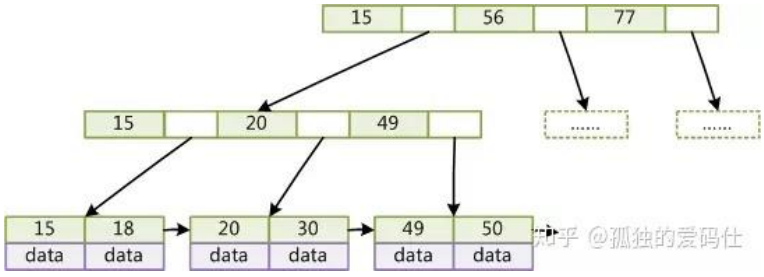
作为 B 树的加强版，B+树与 B 树的差异在于：
（详见下题）

B+树的特性如下：

- 所有关键字都存储在叶子节点上，且链表中的关键字恰好是有序的。
- 查找时不可能返回非叶子节点。
- 非叶子节点相当于叶子节点的索引，叶子节点相当于是存储（关键字）数据的数据层。
- 更适合文件索引系统。



一般在数据库系统或文件系统中使用的 B+树结构都在经典 B+树的基础上进行了优化，增加了顺序访问指针。



如上图所示，在 B+树的每个叶子节点增加一个指向相邻叶子节点的指针，就形成了带有顺序访问指针的 B+树。

做这个优化的目的是为了提高区间访问的性能，例如图中要查询 key 为从 18 到 49 的所有数据记录，当找到 18 后，只需顺着节点和指针顺序遍历就可以一次性访问到所有数据节点，极大提高了区间查询效率。

MySQL 为什么使用 B/B+树

红黑树等数据结构也可以用来实现索引，但是文件系统以及数据库系统普遍采用 B 树或者 B+树。

一般来说，索引本身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式存储在磁盘上。这样的话，索引查找过程中就要产生磁盘 I/O 消耗，相对于内存存取，I/O 存取的消耗要高几个数量级，所以**评价一个数据结构作为索引的优劣最重要的指标就是在查找过程中磁盘 I/O 操作次数的渐进复杂度**。

换句话说，索引的结构组织要尽量减少查找过程中磁盘 I/O 的存取次数。

下面先介绍内存和磁盘存取原理，然后再结合这些原理分析 B/B+树作为索引的效率。

（该部分略）

先从 B 树分析，根据定义，可知检索一次最多需要访问 h 个节点，h 为树高。

数据库系统的设计者巧妙利用了磁盘预读原理，将一个节点的大小设为等于一个页，这样每个节点只需要一次 I/O 就可以完全载入。

B 树中一次检索最多需要 (h-1) 次 I/O（根节点常驻内存），渐进复杂度为 O(h) = O(logdN) * O(h) = O(logdN)。一般实际应用中，出度 d 是非常大的数字，通常超过 100，因此 h 非常小（通常不超过 3），这里的出度 d 表示树的度，即树中各个节点的度的最大值。

因此用 B 树作为索引结构效率是非常高的。

为什么不用红黑树？

而红黑树这种结构，h 明显要大的多。由于逻辑上很近的节点（父子）在物理上可能很远，无法利用空间局部性，所以红黑树的 I/O 渐进复杂度也为 O(h)，效率明显比 B+树差很多。

上文还说过，B+树更适合外存索引，原因和内节点出度 d 有关。从上面分析可以看到，d 越大索引的性能越好，而出度的上限取决于节点内 key 和 data 的大小：

$$d_{\max} = \left\lfloor \frac{\text{page\_size}}{\text{key\_size} + \text{data\_size} + \text{point\_size}} \right\rfloor$$

由于 B+树内节点去掉了 data 域，因此可以拥有更大的出度，性能也更好。

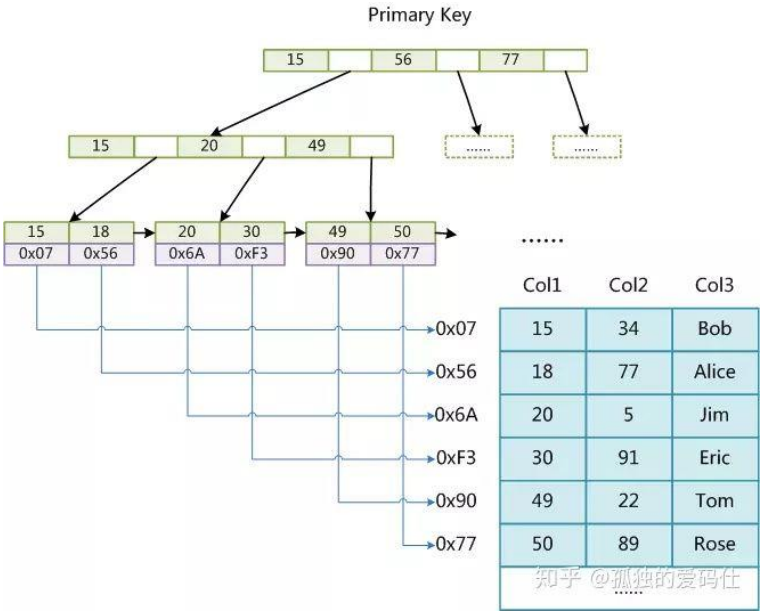
为什么不用 B 树？

- B+树把所有数据都存放在叶子节点上，内部节点保存索引，因此中间节点可以存放更多指针，树可以更矮更胖，减少了查询时的磁盘 I/O 次数，提高了查询效率。
- 叶子节点之间有指针连接，可以进行范围查询，方便区间访问

MyISAM 索引实现

MyISAM 使用 B+树作为索引结构，叶节点的 data 域存放数据记录的地址。

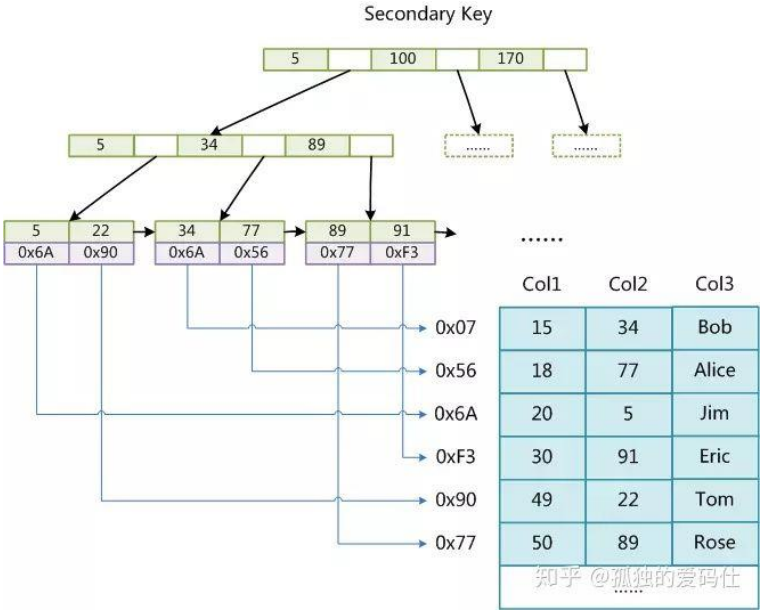
下图是 MyISAM 表的主索引（Primary key）示意图。表一共有三列，以 Col1 为主键。



MyISAM 的主索引示意图

可以看出 MyISAM 的索引文件仅仅保存数据记录的地址。在 MyISAM 中，主索引和辅助索引（Secondary Key）在结构上没有任何区别，只是主索引要求 key 是唯一的，而辅助索引的 key 可以重复。

如果在 Col2 上建立一个辅助索引，则该索引的结构如下图所示：



MyISAM 的辅助索引示意图

同样是一棵 B+树，data 域保存数据记录的地址。

MyISAM 中索引检索的算法：首先按照 B+树搜索算法搜索索引，如果指定的 Key 存在，则取出其 data 域的值，然后以 data 域的 值为地址，读取相应数据记录。

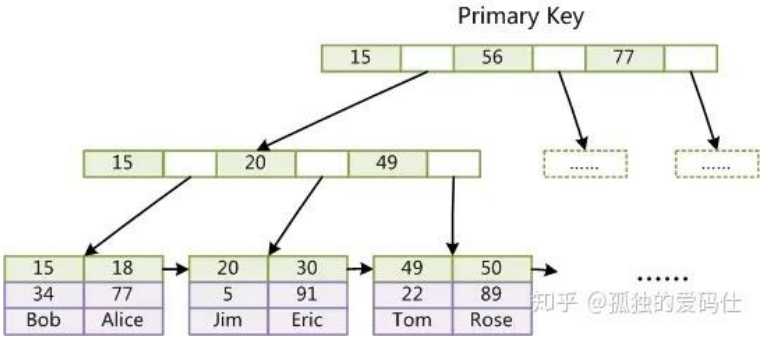
MyISAM 的采用“非聚集索引”，与 InnoDB 的“聚集索引”相区分。

InnoDB 的索引实现

虽然 InnoDB 也使用 B+树作为索引结构，但具体实现方式却与 MyISAM 截然不同。

第一个区别是，InnoDB 的数据文件本身就是索引文件。

从上文知道，MyISAM 索引文件和数据文件是分离的，索引文件仅保存数据记录的地址。而在 InnoDB 中，表数据文件本身就是按 B+树组织的一个索引结构，这棵树的叶节点 data 域保存了完整的数据记录。这个索引的 key 是数据表的主键，因此 InnoDB 表数据文件本身就是主索引。

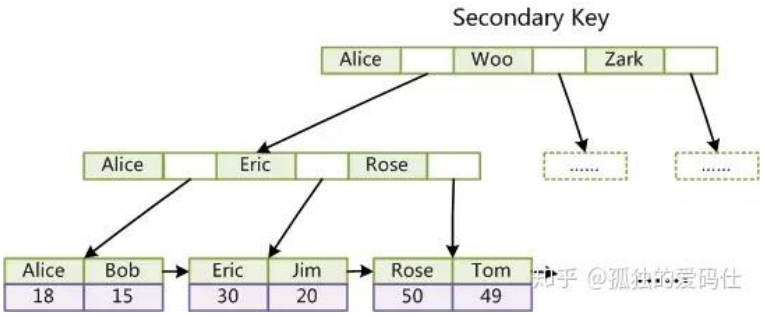


InnoDB 的主索引示意图

第二个区别是，InnoDB 的辅助索引中，data 域存储相应记录主键的值，而不是地址。

换句话说，InnoDB 的所有辅助索引都引用主键作为 data 域。

下图是定义在 Col3 上的一个辅助索引：



InnoDB 的辅助索引示意图

这里以英文字符的 ASCII 码作为排序的大小比较准则。

聚集索引这种实现方式使得按主键的搜索十分高效，但是辅助索引搜索需要检索两遍索引：首先检索辅助索引获得主键，然后用主键到主索引中检索获得记录。

了解不同存储引擎的索引实现方式对于正确使用和优化索引都非常有帮助。

例如，在 InnodDB 中：

- 不建议使用过长的字段作为主键：
  - 因为 InnoDB 所有辅助索引都引用主索引，过长的主索引会令辅助索引变得过大。
- 不建议使用非单调的字段作为主键

- 因为 InnoDB 数据文件本身是一棵 B+树，非单调的主键会造成在插入新记录时数据文件为了维持 B+树的特性而频繁的分裂调整，十分低效，而使用自增字段作为主键则是一个很好的选择。

9. 【热门】B 树和 B +树的区别（红黑树）

- B 树 - 维基百科：https://zh.wikipedia.org/wiki/B 树
- B-tree – Wikipedia：https://en.wikipedia.org/wiki/B-tree
- 对 B+树，B 树，红黑树的理解：https://www.jianshu.com/p/86a1fd2d7406
- 面试官问你 B 树和 B+树，就把这篇文章丢给他：https://blog.ouyangsihai.cn/mian-shi-guan-wen-ni-b-shu-he-b-shu-jiu-ba-zhe-pian-wen-zhang-diu-gei-ta.html
- database - What are the differences between B trees and B+ trees? ：https://stackoverflow.com/questions/870218/what-are-the-differences-between-b-trees-and-b-trees
- 平衡二叉树、B 树、B+树、B*树 理解其中一种你就都明白了：https://zhuanlan.zhihu.com/p/27700617
- B+树 - 维基百科，自由的百科全书：https://zh.wikipedia.org/wiki/B%2B 树
- b 树和 b+树的区别：https://juejin.cn/post/6844903908985274381
- B+树详解 | Ivanzz：https://ivanzz1001.github.io/records/post/data-structure/2018/06/16/ds-bplustree

B+树是 B 树的加强版，与 B 树的差别有：

		B+树	B 树
1	关键字数量	非叶子树的指针个数与关键字个数相同 有 n 个节点的子树包含 n 个关键字，关键字仅起到索引作用	n 个子节点只有 n-1 个关键字
2	存储方式	内部节点仅存储索引不存储数据 叶子节点存储了所有关键字和数据	数据存储在每个节点中
3	有序性	叶子节点根据关键字从小到大顺序连接，形成一个有序链表	
4	查询结果	通过索引找到叶子节点中的数据才能结束，也即走完了从根到叶的完整路径	找到符合条件的节点即返回
5	适用场景	数据库索引	文件索引

B+树的优点：

- 方便全表扫描，只需扫描一遍叶子节点（叶子节点保存了所有的数据）
- 方便区间查询（叶子节点间有指针连接，形成有序链表，数据紧密性和缓冲命中率更高）
- 查询效率更稳定（所有关键字数据地址都存储在叶子节点，每次查找次数相同）

B 树的优点：

- 如果频繁访问距离根结点较近的节点，效率更高（每个节点都保存数据）

10. 索引最左前缀 / 最左匹配

11. 【热门】Mysql 的优化（索引优化，性能优化）

12. 说一下事务是怎么实现的

13. MySQL 数据库引擎介绍，InnoDB 和 MyISAM 的特点与区别

- 深入理解 MySQL 索引底层实现原理：https://zhuanlan.zhihu.com/p/77383599
- MyISAM 和 InnoDB 的区别：https://blog.csdn.net/daiyudong2020/article/details/104468714
- MySQL 存储引擎——MyISAM 与 InnoDB 区别：https://blog.csdn.net/xifeijian/article/details/20316775
- mysql 中 innodb 和 myisam 对比及索引原理区别：https://blog.csdn.net/qq_27607965/article/details/79925288
- Mysql 中 MyISAM 和 InnoDB 的区别有哪些？https://www.zhihu.com/question/20596402
- MySQL 存储引擎——MyISAM 与 InnoDB 区别：https://segmentfault.com/a/1190000008227211

		MyISAM	InnoDB
6	默认引擎	MySQL 5.5 版本之前	MySQL 5.5 版本及之后
7	**外键**	×	√
8	**事务**	×	**√（4 个事务隔离级别）**
9	**行锁**	**×（表锁）**	**√（也会退化为表锁）**
10	全文索引	√	×（5.7 版本后支持）
11	主键	可以没有	必须有（不设会内置）
12	索引	非聚集索引 只能缓存索引	聚集索引 能缓存索引，也能缓存数据
13	数据存储	索引文件和数据文件分离 （.myi 索引文件和.myd 数据文件）	数据文件按主键聚集 （.idb 数据文件）
14	索引实现方式	见“【热门】索引的底层实现”最后两小节	
15	并发	读写互相阻塞，但读不会阻塞读	读写阻塞与事务隔离级别有关
16	查询速度	更快 [1]	
17	适用场景	• 不需要事务支持 • 并发较低（只能表锁） • 数据修改少（阻塞方式），读为主 • 数据一致性要求较低	需要事务支持 高并发（行级锁） 数据更新较为频繁 数据一致性要求较高 硬件设备内存较大（可缓存索引+数据）

MySQL 里还有：

- MEMORY 存储引擎：将表中数据保存在内存里，适合数据量较小且访问频繁的场景
- MERGE 存储引擎

[1] 为什么 MyISAM 查询更快：

- InnoDB 除了索引还要缓存数据块，MyISAM 只缓存索引，因此加载索引更快
- InnoDB 寻址先到块再到行，MyISAM 直接寻址到文件 offset
- InnoDB 要维护 MVCC（多版本并非控制）一致，即使场景没有也要检查和维护

14. 数据库中事务的 ACID（四大特性都要能够举例说明，理解透彻，比如原子性和一致性的关联，隔离性不好会出现的问题）

15. 什么是脏读，不可重复读和幻读？

16. 【热门】数据库的隔离级别，MySQL 和 Oracle 的隔离级别分别是什么

Redis

Linux

1. 【热门】Linux 的 I/O 模型介绍以及同步异步阻塞非阻塞的区别
2. 文件系统的理解（EXT4，XFS，BTRFS）
3. EPOLL 的介绍和了解
4. IO 复用的三种方法（select,poll,epoll）深入理解，包括三者区别，内部原理实现？
5. Epoll 的 ET 模式和 LT 模式（ET 的非阻塞）
6. 查询进程占用 CPU 的命令（注意要了解到 used，buf，代表意义）
7. linux 的其他常见命令（kill，find，cp 等等）
8. Linux 执行 ls 时操作系统做了什么
9. shell 脚本用法
10. 硬连接和软连接的区别
11. 文件权限怎么看（rwx）
12. 文件的三种时间（mtime, atime，ctime），分别在什么时候会改变
13. Linux 监控网络带宽的命令，查看特定进程的占用网络资源情况命令
14. Linux 中线程的同步方式有哪些？
15. 怎么修改一个文件的权限
16. 查看文件内容常用命令

17. 怎么找出含有关键字的前后 4 行

18. Linux 的 GDB 调试

19. coredump 是什么，怎么才能 coredump

20. tcpdump 常用命令

21. crontab 命令

22. 查看后台进程

场景题、算法和数据结构

1. leetcode hot100 至少刷两遍，剑指 offer 至少刷两遍，重中之重！！

2. 介绍熟悉的设计模式（单例，简单工厂模式）

3. 写单例模式，线程安全版本

4. 写三个线程交替打印 ABC

5. 二维码登录的实现过程

6. 不使用临时变量实现 swap 函数

7. 实现一个 strcpy 函数（或者 memcpy），如果内存可能重叠呢

8. 实现快排

9. 快排存在的问题，如何优化

10. 实现一个堆排序

11. 实现一个插入排序

12. 希尔排序，手撕

13. 反转一个链表

14. 学生选课

15. 春晚红包

16. 【热门】Top K 问题（可以采取的方法有哪些，各自优点？）

17. 8G 的 int 型数据，计算机的内存只有 2G，怎么对它进行排序？（外部排序）

18. 自己构建一棵二叉树，使用带有 null 标记的前序遍历序列

19. 介绍一下 b 树和它的应用场景有哪些

20. 介绍一下 b+树和它的应用场景有哪些

21. 介绍一下红黑树和它的应用场景有哪些

22. 怎么写 sql 取表的前 1000 行数据

23. N 个骰子出现和为 m 的概率

24. 海量数据问题（可参考左神的书）

25. 一致性哈希

26. Dijkstra 算法

27. 如何实现一个动态数组

28. 最小生成树算法说一下

29. 海量数据的 bitmap 使用原理

30. 布隆过滤器原理与优点

31. 布隆过滤器处理大规模问题时的持久化，包括内存大小受限、磁盘换入换出问题

32. 实现一个队列，并且使它支持多线程，队列有什么应用场景

智力题

1. 100 层楼，只有 2 个鸡蛋，想要判断出那一层刚好让鸡蛋碎掉，给出策略

2. 毒药问题，1000 瓶水，其中有一瓶可以无限稀释的毒药，要快速找出哪一瓶有毒，需要几只小白鼠

3. 先手必胜策略问题：100 本书，每次能够拿 1-5 本，怎么拿能保证最后一次是你拿

4. 放 n 只蚂蚁在一条树枝上，蚂蚁与蚂蚁之间碰到就各自往反方向走，问总距离或者时间

5. 瓶子换饮料问题：1000 瓶饮料，3 个空瓶子能够换 1 瓶饮料，问最多能喝几瓶

6. 在 24 小时里面时针分针秒针可以重合几次

7. **有一个天平，九个砝码，一个轻一些，用天平至少几次能找到轻的？**
8. **有十组砝码每组十个，每个砝码重 10g，其中一组每个只有 9g，有能显示克数的秤最少几次能找到轻的那一组砝码？**
9. **生成随机数问题：给定生成 1 到 5 的随机数 Rand5()，如何得到生成 1 到 7 的随机数函数 Rand7()**
10. **赛马：有 25 匹马，每场比赛只能赛 5 匹，至少要赛多少场才能找到最快的 3 匹马？**
11. **烧 香 / 绳子 / 其他 确定时间问题：有两根不均匀的香，燃烧完都需要一个小时，问怎么确定 15 分钟的时长？**
12. **掰巧克力问题 N_M 块巧克力，每次掰一块的一行或一列，掰成 1_1 的巧克力需要多少次？（1000 个人参加辩论赛，1V1，输了就退出，需要安排多少场比赛）**

大数据

1. **介绍一下 Hadoop**
2. **说一下 MapReduce 的运行机制**
3. **介绍一下 kafka**
4. **为什么 kafka 吞吐量高？介绍一下零拷贝**
5. **介绍一下 spark**

6. 介绍一下 spark-streaming
7. spark 的 transformation 和 action 有什么区别
8. spark 常用的算子说几个
9. 如何保证 kafka 的消息不丢失
10. kafka 如何选举 leader
11. 说下 spark 中的宽依赖和窄依赖
12. 说下 spark 中 stage 是依照什么划分的
13. spark 的内存管理是怎样的
14. spark 的容错机制是什么样的

HR 面

1. 自我介绍
2. 项目中遇到的最大难点
3. 项目中的收获
4. 可以实习的时间，实习时长
5. 哪里人
6. 说一下自己的性格
7. 你的优缺点是什么

8. **有什么兴趣爱好，画的怎么样 / 球打的如何 / 游戏打的怎么样**

9. **看过最好的一本书是什么**

10. **学习技术中有什么难点**

11. **怎么看待加班**

12. **觉得深圳怎么样（或者其他地点）**

13. **遇见过最大的挫折是什么，怎么解决的**

14. **职业规划**

15. **目前的 offer 情况**

16. **你最大的优势和劣势是什么**

17. **介绍在项目里面充当的角色**

18. **介绍一下本科获得的全国赛奖项的情况**

19. **最有成就感的事情 / 最骄傲的一件事情**

20. **在实验室中担任什么角色，参加的 XXX 能聊聊吗**

21. **用两个词来形容自己**

反问