C++性能优化简介（一）

一、程序性能优化简介

1、程序性能优化简介

在计算机发展的早期阶段，硬件资源相对而言是非常昂贵的，CPU运行时间与内存容量给程序开发人员设置了极大限制。因此，早期的程序对运行性能和内存空间占用的要求是非常严格的，很多开发人员为了减少1%的CPU运行时间，为减少几十个甚至几个字节而不懈努力。随着计算机技术的快速发展，硬件资源变得相对便宜。但如果认为软件开发时，程序的性能优化不再重要，硬件将解决性能问题也是片面的。计算机硬件的发展解决了部分软件的性能问题，但随着硬件计算能力的提高，用户对软件功能的要求也越来越高，软件功能也变得越来越复杂，给用户的界面和操作体验也越来越智能和友好。但复杂的用户需求带来软件性能上的要求是硬件不能完全解决的。众多实际项目经验证明，如果在开发软件时不重视性能优化，最终实现了软件的功能要求，但软件的运行效率低下，最终也不能给用户带来很好的效益。但另一方面，计算机硬件越来越便宜，而优秀的软件开发工程师则越来越昂贵，在软件开发过程中无限制的性能优化同样会导致软件开发过程中人力成本的大幅增加。因此，软件开发过程中的性能优化必须在便宜的计算机硬件和昂贵的优秀工程师之间找到一个平衡点。

2、程序性能优化的流程

应用程序性能优化的流程如下：

名字	内容	大小（KB）	类型
复杂文档100.doc	100页，100图片，16表格，60文本框，其它	3082	doc
复杂文档100.odt	100页，100图片，16表格，60文本框，其它	1249	odt
复杂文档100.lwp	100页，100图片，16表格，60文本框，其它	1246	lwp
简单文档120.doc	120页，纯文本	672	doc
简单文档120.odt	120页，纯文本	19	odt
简单文档120.lwp	120页，纯文本	13	lwp

对于同一种文档类型，每一种文档类型包含两个基准文件，分别有不同的文档内容。
（2）基准用例

基准用例是性能基准测试时需要执行的一系列用例。基准用例的选择有一定原则，既要尽可能全面地覆盖应用程序的主要功能，又不能像功能测试用例那样复杂，因此，基准用例应该是用户日常操作经常遇到的情形。

不同的基准用例在性能基准中的地位并不相同，每一个基准用例都需要一个权值来表明它对整体性能基准的贡献度。权值的定义依据具体情况各有不同，一个比较实用的定义公式如下：
权值=用例频率X用例重要性。

用例频率是用户一定时间内执行该用例的平均次数。理想的用例频率应该通过用户行为数据反馈获得。例如通用文字处理软件的用户一天内可能会执行“打开文档”用例5次，执行“保存文档”的用例15次。

用例重要性是一个修正系数，反映用例没有完成前对用户工作的影响程度。文字处理软件打开一个文档时有“异步打开”的功能，即程序会首先读入文档的部分内容并显示给用户，然后在后台继续读入文档的后续内容。对于“异步打开”功能可以定义两个用例，“第一页显示”（从用户选择打开文档到文档的第一页显示出来的过程）和“全部读入完毕”（从用户选择打开文档到文档的所有页的内容已经加载完毕的过程）。“第一页显示”用例的重要性为1，表示不执行完本用例，用户不能继续工作；“全部读入完毕”用例的重要性为0.5，表示本用例不会显著影响用户的工作，文档在后台加载，用户前台已经可以编辑，除非用户需要编辑的内容还没有加载出来。
文字处理程序的部分基准用例如下：

名字	结果值（秒）	权值	分值
冷启动	10.5	1	10.5
热启动	2.1	6	12.6
关闭	2.3	1.4	3.22
新建文档	0.8	3	2.4
第一页显示（简单doc文档）	0.5	5	2.5
全部读入完毕（简单doc文档）	2	2.5	5
保存文档（简单doc文档）	0.2	15	3
第一页显示（复杂doc文档）	0.8	5	4
全部读入完毕（复杂doc文档）	4.8	2.5	12
保存文档（复杂doc文档）	0.8	15	12
插入图片	0.1	10	1
翻页	0.02	100	2
粘贴	0.03	50	1.5
打印预览	0.8	5	4
查找	0.01	20	0.2
总分			71.5

性能基准可以反映应用程序的总体性能，定义良好的性能基准用途如下：
（1)应用程序性能的绝对指标。任何想要了解产品性能的人，无论是管理层还是客户，都可以通过产品性能报告了解产品的性能。

（2）通过比较不同版本的基准结果，提前发现性能下降的问题和验证性能提升的设计结果。软件开发过程中通常都会进行每日构建，性能基准也可以在每日构建的基础上每日运行，及时发现性能问题，而不是在产品即将发布时进行性能优化。

（3)比较不同厂商的类似软件的性能。横向的比较需要性能基准，可以找出自己软件产品的性能薄弱环节，集中力量进行优化。

三、程序性能分析方法

1、性能分析方法简介

拥有定义良好的性能基准后，可以轻易发现应用程序存在的性能问题。发现性能问题后需要对性能问题进行分析，程序的性能分析过程包括：性能问题分类、查找性能瓶颈、进行性能优化。

2、性能问题分类

一个操作执行太慢，需要首先分类是IO操作密集引起的问题还是CPU相关的计算密集型问题。正确的分类将直接影响进一步的问题分析。

区别IO相关还是CPU相关问题的简单方法是隔离IO影响后，看性能是否得到改善，例如同时在机械硬盘和SSD硬盘上测试，如果性能显著提高，则是IO相关的问题。

对于文字处理软件，冷启动需要10.5秒，热启动需要2.1秒，因此冷启动的主要问题在IO。无论是冷启动还是热启动，应用程序都是完全退出后再重新启动，执行的代码流程完全一样，唯一区别在于IO：冷启动后操作系统会缓存很多动态库的代码页在内存。

3、查找性能瓶颈

对性能问题分类后，可以使用性能分析工具在代码层次查找性能瓶颈，性能分析工具有监测工具和注入工具两类。
监测工具如下：

- perfmon，Windows工具，可以监测所有的性能指标。
- FileMon，Windows工具，监测IO操作。
- ProcessExplorer，Windows工具，监测进程相关的所有操作。
- sysstat，Linux工具，监测所有的性能指标。
- iostat，Linux工具，监测IO操作。
- vmstat，Linux工具，监测内存变化。

注入工具如下：

- IBM rational quantify，Windows工具，针对C应用程序代码注入，可以计算函数调用次数、时间等。
- Valgrind，Linux工具，针对C应用程序代码注入，可以计算函数调用次数、事件、内存分配、内存泄漏检测等。
- IBM rational purify，Windows工具，针对C++应用程序代码注入，可以进行内存分析。
- WinDbg，Windows工具，调试工具。
- GDB，Linux工具，调试工具。
- Dependency walker，Windows工具，分析动态链接库之间的动态、静态依赖关系。
- ldd，Linux工具，分析共享对象间的依赖关系。

4、查找性能优化机会

代码层次的性能优化设计的改动通常局限在有限的函数调用内，相对比较容易完成。进一步的性能提升的机会需要在设计层次进行查找。设计层面的性能分析需要性能优化者对软件的整体架构有比较深入的了解，需要具体问题具体分析。

四、程序性能优化方法

性能问题分析完成后，需要进行性能优化。根据性能分析结果的不同，优化方法也各有不同。

1、针对IO瓶颈的性能优化

每次IO操作大概在10 ms量级，100次就需要1秒左右，因此尽量避免不必要的IO操作。具体做法如下：
（1）预先顺序读文件避免随机访问。
（2）合并多个小文件为单个大文件。
（3）优化动态库文件的加载。
（4）交错IO时间和CPU时间。

2、针对计算密集的性优化

计算密集的性能问题主要有内存分配性能、字符串操作、共享变量的互斥锁保护等，具体优化方法如下：
（1）去除冗余代码。
（2）字符串操作优化。
（3）减少内存分配、释放操作，例如使用内存池。
（4）减少不必要的互斥锁操作。
（5）根据性能需求选择数据结构。
（6）延迟工作，按需执行。
（7）减少跨进程的调用。
（8）使用高性能的函数库。

3、C++语言特性相关的性能优化

C++语言特性相关的性能优化包括内联函数、引用、编译优化选项等。

4、用户体验的性能优化

有些设计不能真正提升性能，但让用户体验到了性能提升。如：
（1）流式播放设计，用户不需要等到视频文件下载完成再播放，可以边下载边播放。
（2）线程化设计，对于需要较长时间完成的操作，可以设计为非阻塞式的，用户可以在等待时间完成其它操作任务。

5、设计层面的性能优化

设计层面的性能优化需要根据软件整体架构具体问题具体分析。

C性能优化（二）——C对象模型

一、C++对象模型与性能优化

对象模型是面向对象程序设计语言的重要方面，会直接影响面向对象语言编写程序的运行机制以及对内存的使用机制，因此了解对象模型是进行程序性能优化的基础。只有深入理解C++对象模型，才能避免程序开发过程中一些不易发现的内存错误，从而改善程序性能，提高程序质量。

二、C++程序的内存分布

1、程序内存分布简介

通常，计算机程序由代码和数据组成，因此代码和数据也是影响程序所需内存的主要因素。代码是程序运行的指令，比如数学运算、比较、跳转以及函数调用，其大小通常由程序的功能和复杂度决定，正确地使用程序编写技巧以及编程语言的特性可以优化所生成的代码的大小；数据是代码要处理的对象。

程序占用的内存区通常分为五种：全局/静态数据区、常量数据区、代码区、栈、堆。
程序的代码存储在代码区中，而程序的数据则根据数据种类的不同存储在不同的内存区中。C++语言中，数据有不同的分类方法，例如常量和变量，全局数据和局部数据，静态数据和非静态数据。此外，程序运行过程中动态产生和释放的数据也要存放在不同的内存区。
不同内存区存储的数据如下：
（1）全局/静态数据区存储全局变量以及静态变量（包括全局静态变量和局部静态变量）。
（2）常量数据区存储程序中的常量字符串等。
（3）栈中存储自动变量或者局部变量，以及传递函数参数等，而堆是用户程序控制的存储区，存储动态产生的数据。
不同类型的数据在内存存储位置的示例如下：

```
#include <stdio.h>
#include <stdlib.h>

using namespace std;

int g_GolbalVariable = 100;

int main()
{
    int localVariable = 1;
    static int staticLocalVariable = 200;
    const int constLocalVariable = 100;
```

```
    char* pLocalString1 = "pLocalString1";
    const char* pLocalString2 = "pLocalString2";
    int* pNew = new int[5]; // 16字节对齐
    char* pMalloc = (char*)malloc(1);

    printf( "GolbalVariable: 0x%x\n", &g_GolbalVariable);
    printf( "Static Variable: 0x%x\n", &staticLocalVariable);
    printf( "LocalString1: 0x%x\n", pLocalString1);
    printf( "const LocalString2: 0x%x\n", pLocalString2);
    printf( "const LocalVariable: 0x%x\n", &constLocalVariable);

    printf( "New: 0x%x\n", pNew);
    printf( "Malloc: 0x%x\n", pMalloc);

    printf( "LocalVariable: 0x%x\n", &localVariable);

    return 0;
}
```

上述代码定义了8个变量，一个全局变量`GolbalVariable`，一个静态局部变量`staticLocalVariable`，六个局部变量。在`RHEL 7.3`系统使用`GCC`编译器编译运行，程序输出结果如下：

> GolbalVariable: 0x60105c
> Static Variable: 0x601060
> LocalString1: 0x4009a0
> const LocalString2: 0x4009ae
> const LocalVariable: 0xdbd23ef8
> New: 0x182b010
> Malloc: 0x182b030
> LocalVariable: 0xdbd23efc

全局变量、静态变量和局部静态变量存储在全局/静态数据区。

字符串常量存储在常量数据区，`pLocalString1`指向的字符串`"pLocalString1"`的长度是13字节，加上结束符`'\0'`，共计14个字节，存储在`0x4009a0`开始的14个字节内存空间；存储`pLocalString2`的字符串`"pLocalString2"`时，从`0x4009ae`地址开始，因此，没有进行内存对齐处理。程序中的其它字符串常量，如`printf`中的格式化串通常也存储在常量数据区。

通过`new`、`malloc`获得的内存是堆的内存。通过`new`申请5个`int`所需的内存，但由于内存边界需要字节对齐（堆上分配内存时按16字节对齐），因此申请5个int共计20个字节，但占据32字节的内存。通过`malloc`申请1个字节的内存，申请1个字节时会从32字节后开始分配。
内存对齐虽然会浪费部分内存，但由于CPU在对齐方式下运行较快，因此内存对齐对于程序性能是有益的。C++语言中`struct`、`union`、`class`在编译时也会对成员变量进行内存对齐处理，开发人员可以使用`#progma pack()`或者编译器的编译选项来控制对`struct`、`union`、`class`的成员变量按多少字节对齐，或者关闭对齐。

2、全局/静态数据区、常量数据区

全局/静态存储区、常量数据区在程序编译阶段已经分配好，在整个程序运行过程中始终存在，用于存储全局变量、静态变量，以及字符串常量等。其中字符串常量存储的区域是不可修改的内存区域，试图修改字符串常量会导致程序异常退出。

```
char* pLocalString1 = "hello world";
pLocalString1[0] = 'H';// 试图修改不可修改的内存区
```

全局/静态数据区除了全局变量，还有静态变量。C语言中可以定义静态变量，静态变量在第一次进入作用域时被初始化，后续再次进入作用域时不必初始化。C++语言中，可以定义静态变量，也可以定义类的静态成员变量，类的静态成员变量用来在类的多个对象间共享数据。类的静态成员变量存储在全局/静态数据区，并且只有一份拷贝，由类的所有对象共享。如果通过全局变量在类的多个对象间共享数据则会破坏类的封装性。

```
#include <stdio.h>
#include <stdlib.h>

class A
{
public:
    int value;
    static int nCounter;
    A()
    {
        nCounter++;
    }
    ~A()
    {
        nCounter--;
    }
};
int A::nCounter = 0;

int main()
{
    A a;
    A b;
    printf("number of A: %d\n", A::nCounter);
    printf("non-static class member: 0x%x\n", &a.value);
    printf("non-static class member: 0x%x\n", &b.value);
    printf("static class member: 0x%x\n", &a.nCounter);
    printf("static class member: 0x%x\n", &b.nCounter);

    return 0;
}
```

上述代码，类A定义了一个静态成员变量`nCounter`用于对类A的对象进行计数，类A也定义了一个成员变量`value`，在`RHEL 7.3`系统使用`GCC`编译器编译运行，程序输出结果如下：

> number of A: 2
> non-static class member: 0x99a457c0
> non-static class member: 0x99a457b0
> static class member: 0x601048
> static class member: 0x601048

对象a和对象b中的`value`成员变量的地址不同，而静态成员变量`nCounter`的地址相同。类A的每一个对象会有自己的`value`存储空间，在栈上分配；类A的所有对象共享一个`nCounter`的存储空间，在全局/静态数据区分配。

3、堆和栈

在C/C++语言中，当开发人员在函数内部定义一个变量，或者向某个函数传递参数时，变量和参数存储在栈中。当退出变量的作用域时，栈上的存储单元会被自动释放。当开发人员通过`malloc`申请一块内存或使用`new`创建一个对象时，申请的内存或对象所占的内存在堆上分配。开发人员需要记录得到的地址，并在不再需要时负责释放内存空间。

```
#include <stdio.h>
#include <stdlib.h>

using namespace std;

int g_GolbalVariable = 100;

int main()
{
    int localVariable = 1;
    static int staticLocalVariable = 200;
    const int constLocalVariable = 100;
    char* pLocalString1 = "pLocalString1";
    const char* pLocalString2 = "pLocalString2";
    int* pNew = new int[5]; // 16字节对齐
    char* pMalloc = (char*)malloc(1);

    printf( "GolbalVariable: 0x%x\n", &g_GolbalVariable);
    printf( "Static Variable: 0x%x\n", &staticLocalVariable);
    printf( "LocalString1: 0x%x\n", pLocalString1);
    printf( "const LocalString2: 0x%x\n", pLocalString2);
    printf( "const LocalVariable: 0x%x\n", &constLocalVariable);

    printf( "New: 0x%x\n", pNew);
    printf( "Malloc: 0x%x\n", pMalloc);

    printf( "LocalVariable: 0x%x\n", &localVariable);

    return 0;
}
```

上述代码中，通过`new`在堆上申请5个int的所需的内存空间，将获得的地址记录在栈上的变量`pNew`中；通过`malloc`在堆上申请1字节的内存空间，将获得的地址记录在栈上的变量`pMalloc`中。

```
int* pNew = new int[5]; // 16字节对齐
char* pMalloc = (char*)malloc(1);
```

在`main`函数结束时，`pNew`和`pMalloc`自身是栈上的内存单元，会被自动释放，但`pNew`和`pMalloc`所指向的内存是堆上的，虽然指向堆空间的`pNew`和`pMalloc`指针变量已经不存在，但相应的堆空间内存不会被自动释放，造成内存泄露。通过`new`申请的堆内存空间需要使用`delete`进行释放，使用`malloc`获得的堆空间内存需要使用`free`进行释放。

既然栈上的内存空间不存内存泄露的问题，而堆上的内存容易引起内存泄露，为什么要使用堆上的内存呢?因为很多应用程序需要动态管理地管理数据。此外，栈的大小有限制，占用内存较多的对象或数据只能分配在堆空间。
栈和堆的区别如下：

（1）大小
通常，程序使用栈的大小是固定的，由编译器决定，开发人员可以通过编译器选项指定栈的大小，但通常栈都不会太大。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    char buf[8 * 1024 * 1024];
    printf("%x\n", buf);

    return 0;
}
```

`RHEL 7.3`系统中默认的栈大小为`8MB`，在`RHEL 7.3`系统使用`GCC`编译器编译运行，程序会运行时出错，原因是栈溢出。

堆的大小通常只受限于系统有效的虚拟内存的大小，因此可以用来分配创建一些占用内存较大的对象或数据。

（2）效率
栈上的内存是系统自动分配的，压栈和出栈都有相应的指令进行操作，因此效率较高，并且分配的内存空间是连续的，不会产生内存碎片；堆上的内存是由开发人员来动态分配和回收的。当开发人员通过`new`或`malloc`申请堆上的内存空间时，系统需要按照一定的算法在堆空间中寻找合适大小的空闲堆，并修改相应的维护堆空闲空间的链表，然后返回地址给程序。因此，效率币栈要低，此外还容易产生内存碎片。

如果程序在堆上申请5个100字节大小的内存块，然后释放其中不连续的两个内存块，此时当需要在堆上申请一个150字节大小的内存块时，则无法充分利用刚刚释放的两个小内存块。由此可见，连续创建和删除占用内存较小的对象或数据时，很容易在堆上造成内存碎片，使得内存的使用效率降低。

4、C++对象创建方式

从C对象模型角度看，对象就是内存中的一块区域。根据C标准，一个对象可以通过定义变量创建，或者通过`new`操作符创建，或者通过实现来创建。如果一个对象通过定义在某个函数内的变量或者需要的临时变量来创建，是栈上的一个对象；如果一个对象是定义在全局范围内的变量，则对象存储在全局/静态数据区；如果一个对象通过`new`操作符创建，存储在堆空间。

对面向对象的C程序设计，程序运行过程中的大部分数据应该封装在对象中，而程序的行为也由对象的行为决定。因此，深入理解C对象的内部结构，从而正确地设计和使用对象，对于设计开发高性能的C++程序很重要。

三、C++对象的生命周期

1、C++对象生命周期简介

对象的生命周期是指对象从创建到销毁的过程，创建对象时要占用一定的内存空间，而对象要销毁后要释放对应的内存空间，因此整个程序占用的内存空间也会随着对象的创建和销毁而动态的发生变化。深入理解对象的生命周期会帮助分析程序对内存的消耗情况，从而找到改进方法。

对象的创建有三种方式，不同方式所创建对象的生命周期各有不同，创建对象的三种方式如下：
（1）通过定义变量创建对象

（2）通过`new`操作符创建对象

（3）通过实现创建对象

2、通过定义变量创建对象

通过定义变量创建对象时，变量的作用域决定了对象的生命周期。当进入变量的作用域时，对象被创建；退出变量的作用域时，对象被销毁。全局变量的作用域时整个程序，被声明为全局对象的变量在程序调用`main`函数前被创建，当程序退出`main`函数后，全局对象才会被销毁。静态变量作用域不是整个程序，但静态变量存储在全局/静态数据区，在程序开始时已经分配好，因此声明为静态变量的对象在第一次进入作用域时会被创建，直到程序退出时被销毁。

```
#include <stdio.h>
#include <stdlib.h>


class A
{
public:
    A()
    {
        printf("A Created\n");
    }
    ~A()
    {
        printf("A Destroyed\n");
    }
};

class B
{
public:
    B()
    {
        printf("B Created\n");
    }
    ~B()
    {
        printf("B Destroyed\n");
    }
};

A globalA;

void test()
{
    printf("test()------------------------->\n");
    A localA;
    static B localB;
    printf("test()<-------------------------\n");
}

int main()
{
    printf("main()------------------------->\n");
    test();
    test();
    static B localB;
    printf("main()<-------------------------\n");
```

```
    return 0;
}
```

上述代码中定义了一个A的全局对象`globalA`，一个A的局部对象`localA`，一个B的静态局部对象`localB`，`localA`和`localB`的作用域为`test`函数。
在`RHEL 7.3`系统使用`GCC`编译器编译运行结果如下：

```
A Created
main()------------------------->
test()------------------------->
A Created
B Created
test()<-------------------------
A Destroyed
test()------------------------->
A Created
test()<-------------------------
A Destroyed
B Created
main()<-------------------------
B Destroyed
B Destroyed
A Destroyed
```

根据程序运行结果，全局对象`globalA`在`main`函数开始前被创建，在`main`函数退出后被销毁；静态对象`localB`在第一次进入作用域时被创建，在`main`函数退出后被销毁，如果程序从来没有进入到其作用域，则静态对象不会被创建；局部对象在进入作用域时被创建，在退出作用域时被销毁。

3、通过`new`操作符创建对象

通过`new`创建的对象会一直存在，直到被`delete`销毁。即使指向对象的指针被销毁，但还没有调用`delete`，对象仍然会一直存在，占据这堆空间，直到程序退出，因此会造成内存泄露。

```
#include <stdio.h>
#include <stdlib.h>

class A
{
public:
    A()
    {
        printf("A Created\n");
    }
    ~A()
    {
        printf("A Destroyed\n");
    }
};

A* createA()
{
    return new A();
}

void deleteA(A* p)
```

```
{
    delete p;
    p = NULL;
}

int main()
{
    A* pA = createA();
    pA = createA();

    deleteA(pA);
    return 0;
}
```

上述代码中，`createA`函数使用`new`操作符创建了一个A对象，并将返回地址记录在`pA`指针变量中；然后再次使用`createA`函数创建了一个A对象，将返回地址记录在`pA`指针变量中，此时`pA`指针将指向第二次创建的A对象，第一次创建的A对象已经没有指针指向。使用`deleteA`销毁对象时，销毁的是第二次创建的A对象，第一次创建的A对象会一直存在，直到程序退出，并且即使在程序退出时，第一次创建的A对象的析构函数仍然不会被调用，最终造成内存泄露。

4、通过实现创建对象

通过实现创建对象通常是指一些隐藏的中间临时变量的创建和销毁。中间临时变量的生命周期很短，不易被开发人员察觉，通常是造成性能下降的瓶颈，特别是占用内存多、创建速度慢的对象。
中间临时对象通常是通过拷贝构造函数创建的。

```
#include <stdio.h>
#include <stdlib.h>

class A
{
public:
    A()
    {
        printf("A Created\n");
    }
    A(const A& other)
    {
        printf("A Created with copy\n");
    }
    ~A()
    {
        printf("A Destroyed\n");
    }
};

A getA(A a)
{
    printf("before\n");
    A b;
    return b;
}

int main()
{
    A a;
```

```
    a = getA(a);
    return 0;
}
```

在 RHEL 7.3 系统使用 GCC 编译器编译运行结果如下：

```
A Created
A Created with copy
before
A Created
A Destroyed
A Destroyed
A Destroyed
```

getA 函数的参数和返回值都是通过值传递的，在调用 getA 是需要把实参复制一份，压入 getA 函数的栈中（对于某些C编译器，getA 函数的返回值也要拷贝一份放在栈中，在 getA 函数调用结束时，参数出栈就会返回给调用者）。因此，在调用 getA 函数时，需要构造一个a的副本，调用一次拷贝构造函数，创建了一个临时变量。

中间临时对象的创建和销毁是隐式的，因此如果中间临时对象的创建和销毁在循环内或是对象构造需要分配很多资源，会造成资源在短时间内被频繁的分配和释放，甚至可能造成内存泄露。
上述代码 getA 函数的问题可以通过传递引用的方式解决，即 getA（A& a），不用构造参数的临时对象。

实际的C工程实践中，会有大量其它类型的隐式临时对象存在，如重载+和重载++等操作符，对对象进行算术运算时也会有临时对象，操作符重载本质上也是函数，因此要尽量避免临时对象的出现。

当一个派生类实例化一个对象时，会先构造一个父类对象，同样，在销毁一个派生类对象时也会销毁其父类对象。派生类对象的父类对象是隐含的对象，其生命周期和派生类对象绑定在一起。如果构造父类对象的开销很大，则所有子类的构造都会开销很大。

```
#include <stdio.h>
#include <stdlib.h>

class A
{
public:
    A()
    {
        printf("A Created\n");
    }
    ~A()
    {
        printf("A Destroyed\n");
    }
};

class B : public A
{
public:
    B(): A()
    {
        printf("B Created\n");
    }
    ~B()
    {
        printf("B Destroyed\n");
```

```
    }
};

int main()
{
    B b;
    return 0;
}
```

在 RHEL 7.3 系统使用 GCC 编译器编译运行结果如下：

```
A Created
B Created
B Destroyed
A Destroyed
```

根据运行结果，创建派生类对象时会先创建隐含的父类对象，销毁派生类对象时会在调用派生类析构函数后调用父类的析构函数。

四、C++对象的内存布局

1、C++对象内部结构简介

C对象的内部结构及实现和C编译器紧密相关，不同的编译器可能会有不同的实现方式。

2、C++简单对象

在一个C++对象中包含成员数据和成员函数，成员数据分为静态成员数据和非静态成员数据；成员函数分为静态成员函数、非静态成员函数和虚函数。

```
#include <stdio.h>
#include <stdlib.h>

class SimpleObject
{
public:
    static int nCounter;
    double value;
    char flag;
    SimpleObject()
    {
        printf("SimpleObject Created\n");
    }
    virtual ~SimpleObject()
    {
        printf("SimpleObject Destroyed\n");
    }

    double getValue()
    {
        return value;
    }
    static int getCount()
    {
        return nCounter;
    }
    virtual void test()
```

```
    {
        printf("virtual void test()\n");
    }
};

int main()
{
    SimpleObject object;
    printf("Obejct start address: 0x%X\n", &object);
    printf("Value address: 0x%X\n", &object.value);
    printf("flag address: 0x%X\n", &object.flag);
    printf("Object size: %d\n", sizeof(object));

    return 0;
}
```

在 RHEL 7.3 系统使用 GCC 编译器编译运行结果如下：

> SimpleObject Created
> Obejct start address: 0x5728F3F0
> Value address: 0x5728F3F8
> flag address: 0x5728F400
> Object size: 24
> SimpleObject Destroyed

上述代码，静态成员数据 nCounter 存储在全局/静态数据区，由类的所有对象共享，并不作为对象占据的内存的一部分，因此 sizeof 返回的 SimpleObject 大小并不包括 nCounter 所占据的内存大小。非静态成员数据 value 和 flag 存储在对象占用的内存中，不论时全局/静态数据区，还是堆上、栈上。Value 是 double 类型，占据8个字节（64位），flag 是 char 类型，占据1个字节，但由于内存对齐，也会占用8字节。

SimpleObject 类对象的数据成员占用了16个字节，剩下的8字节是与虚函数相关的。如果将两个虚函数的 virtual 关键字去掉，则 sizeof（SimpleObject） 将得到16。

虚函数用于实现C语言的动态绑定特性，为了实现动态绑定特性，C编译器遇到含有虚函数的类时，会分配一个指针指向一个函数地址表，即虚函数表（virtual table），虚函数表指针占据了8个字节，并且占据的是类实例内存布局开始的8个字节。

C++简单对象占用的内存空间如下：

（1）非静态成员数据是影响对象占用内存大小的主要因素，随着对象数目的增加，非静态成员数据占用的内存空间会相应增加。

（2）所有的对象共享一份静态成员数据，因此静态成员数据占用的内存空间大小不会随着对象数目的增加而增加。

（3）静态成员函数和非静态成员函数不会影响对象内存的大小，虽然其实现会占用相应的内存空间，同样不会随着对象数目的增加而增加。

（4）如果类中包含虚函数，类对象会包含一个指向虚函数表的指针，虚函数的地址会放在虚函数表中。

在虚函数表中，不一定完全是指向虚函数实现的指针。当指定编译器打开 RTTI 开关时，虚函数表中的第一个指针指向的是一个 typeinfo 的结构，每个类只产生一个 typeinfo 结构的实例，当程序调用 typeid() 来获取类的信息时，实际是通过虚函数表中的第一个指针获取 typeinfo 结构体实例。

3、单继承

C++语言中，继承分为单继承和多继承。

```
#include <stdio.h>
#include <stdlib.h>

class SimpleObject
{
public:
    static int nCounter;
    double value;
    char flag;
    SimpleObject()
    {
        printf("SimpleObject Created\n");
    }
    virtual ~SimpleObject()
    {
        printf("SimpleObject Destroyed\n");
    }

    double getValue()
    {
        return value;
    }
    static int getCount()
    {
        return nCounter;
    }
    virtual void test()
    {
        printf("virtual void SimpleObject::test()\n");
    }
};
int SimpleObject::nCounter = 0;

class DerivedObject : public SimpleObject
{
public:
    double subValue;
    DerivedObject()
    {
        printf("DerivedObject Created\n");
    }
    virtual ~DerivedObject()
    {
        printf("DerivedObject Destroyed\n");
    }
    virtual void test()
    {
        printf("virtual void DerivedObject::test()\n");
    }

};

int main()
```

```
{
    DerivedObject object;
    printf("Obejct start address: 0x%X\n", &object);
    printf("Value address: 0x%X\n", &object.value);
    printf("flag address: 0x%X\n", &object.flag);
    printf("subValue address: 0x%X\n", &object.subValue);
    printf("SimpleObject size: %d\n"
           "DerivedObject size: %d\n",
           sizeof(SimpleObject),
           sizeof(DerivedObject));

    return 0;
}
```

在 RHEL 7.3 系统使用 GCC 编译器编译运行结果如下：

> SimpleObject Created
> DerivedObject Created
> Obejct start address: 0x96EA42D0
> Value address: 0x96EA42D8
> flag address: 0x96EA42E0
> subValue address: 0x96EA42E8
> SimpleObject size: 24
> DerivedObject size: 32
> DerivedObject Destroyed
> SimpleObject Destroyed

根据上述输出结果，构造一个派生类实例时首先需要构造一个基类的实例，基类实例在派生类实例销毁后被销毁。

SimpleObject 类大小是24个字节，DerivedObject 类的大小是32个字节，DerivedObject 增加了一个 double 类型的成员数据 subValue，需要占用8个字节。由于 DerivedObject 类也需要一个虚函数表，因此 DerivedObject 派生类与 SimpleObject 基类使用同一个虚函数表，DerivedObject 派生类在构造时不会再创建一个新的虚函数表，而是在 SimpleObject 基类的虚函数表中增加或修改，DerivedObject 实例的虚函数表中会存储 DerivedObject 相应的虚函数实现，如果 DerivedObject 没有提供某个虚函数实现，则存储基类 SimpleObject 的虚函数实现。

4、多继承

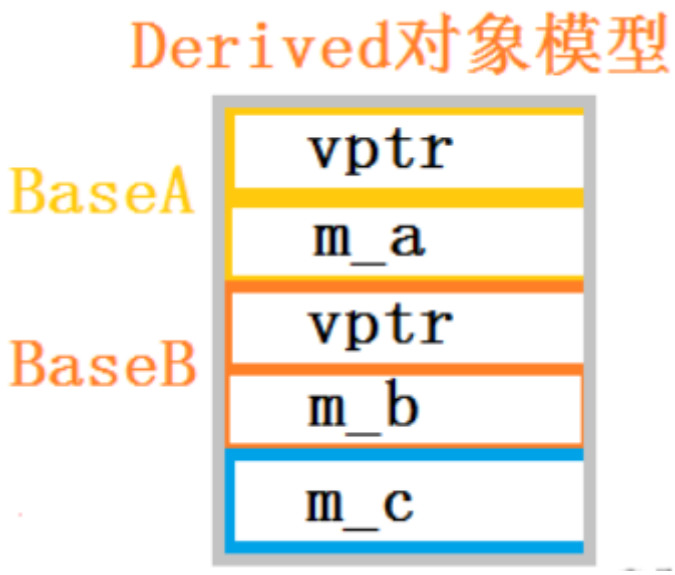
C语言提供多继承的支持，多继承中派生类可以有一个以上的基类。多继承是C语言中颇受争议的一项特性，多继承在提供强大功能的同时也带来了容易造成错误的诸多不便。因此，后续很多面向对象程序设计语言取消了多继承支持，而是提供了更清晰的接口概念。

C++语言中仍然通过继承实现接口，在面向接口的编程模型，如COM，都采用多继承实现。如果需要开发一个文字处理软件，要求有些文档即可以打印有可以存储，有些文档只可以打印或存储。考虑到程序的可扩展性，比较好的设计是将打印和存储分别定义为两个接口，在接口中定义相应的方法。当一个类实现了打印和存储接口时，其对象即可以打印也可以存储。如果只实现了打印或存储，则只具备相应的功能。

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

using namespace std;

class BaseA
```

```
{
public:
    BaseA(int a)
    {
        m_a = a;
    }
    virtual void funcA()
    {
        cout << "BaseA::funcA()" <<endl;
    }
private:
    int m_a;
};

class BaseB
{
public:
    BaseB(int b)
    {
        m_b = b;
    }
    virtual void funcB()
    {
        cout << "BaseB::funcB()" <<endl;
    }
private:
    int m_b;
};

class Derived : public BaseA, public BaseB
{
public:
    Derived(int a, int b, int c):BaseA(a),BaseB(b)
    {
        m_c = c;
    }
private:
    int m_c;
};

struct Test
{
    void* vptrA;
    int a;
    void* vptrB;
    int b;
    int c;
};

int main(int argc, char *argv[])
{
    cout << sizeof(Derived) << endl;
    Derived d(1,2,3);
    Test* pTest = (Test*)&d;
    cout << pTest->a <<endl;//1
    cout << pTest->b <<endl;//2
    cout << pTest->c <<endl;//3
    cout << pTest->vptrA <<endl;//
```

```
    cout << pTest->vptrB <<endl;//

    return 0;
}
```



创建派生类时，首先需要创建基类的对象。由于多继承一个派生类中有多个基类，因此，创建基类的对象时要遵循一定的顺序，其顺序由派生类声明时决定，如果将`Derived`类的声明修改为：

```
class Derived : public BaseB, public BaseA
```

基类对象`BaseB`会被首先创建，`BaseA`对象其次被创建。基类对象销毁的顺序与创建的顺序相反。

多继承会引入很多复杂问题，菱形继承时很典型的一种。菱形继承示例代码如下：

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

using namespace std;

class People
{
public:
    People(string name, int age)
    {
        m_name = name;
        m_age = age;
    }
    void print()
    {
        cout << "name: " << m_name
             << " age: " << m_age <<endl;
    }
private:
    string m_name;
    int m_age;
};

class Teacher : public People
{
    string m_research;
```

```
public:
    Teacher(string name, int age, string research):People(name + "_1", age + 1)
    {
        m_research = research;
    }
};

class Student : public People
{
    string m_major;
public:
    Student(string name, int age,string major):People(name + "_2", age + 2)
    {
        m_major = major;
    }
};

class Doctor : public Teacher, public Student
{
    string m_subject;
public:
    Doctor(string name, int age,string research, string major, string subject):
        Teacher(name, age,research),Student(name, age, major)
    {
        m_subject = subject;
    }
};

struct Test
{
    string name1;
    int age1;
    string research;
    string name2;
    int age2;
    string major;
    string subject;
};

int main(int argc, char *argv[])
{
    Doctor doc("Bauer", 30, "Computer", "Computer Engneering", "HPC");
    cout << "Doctor size: " << sizeof(doc) << endl;
    Test* pTest = (Test*)&doc;
    cout << pTest->name1 << endl;
    cout << pTest->age1 << endl;
    cout << pTest->research << endl;
    cout << pTest->name2 << endl;
    cout << pTest->age2 << endl;
    cout << pTest->major << endl;
    cout << pTest->subject << endl;

    return 0;
}
// output:
// Doctor size: 28
// Bauer_1
// 31
```

```
// Computer
// Bauer_2
// 32
// Computer Engneering
// HPC
```

上述代码中，底层子类对象的内存局部如下：


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

using namespace std;

class People
{
public:
    People(string name, int age)
    {
        m_name = name;
        m_age = age;
    }
    virtual void print()
    {
        cout << "name: " << m_name
             << " age: " << m_age <<endl;
    }
private:
    string m_name;
    int m_age;
};

class Teacher : public People
{
```

```
    string m_research;
public:
    Teacher(string name, int age, string research):People(name + "_1", age + 1)
    {
        m_research = research;
    }
};

class Student : public People
{
    string m_major;
public:
    Student(string name, int age,string major):People(name + "_2", age + 2)
    {
        m_major = major;
    }
};

class Doctor : public Teacher, public Student
{
    string m_subject;
public:
    Doctor(string name, int age,string research, string major, string subject):
        Teacher(name, age,research),Student(name, age, major)
    {
        m_subject = subject;
    }
    virtual void print()
    {

    }
};

struct Test
{
    void* vptr1;
    string name1;
    int age1;
    string research;
    void* vptr2;
    string name2;
    int age2;
    string major;
    string subject;
};

int main(int argc, char *argv[])
{
    Doctor doc("Bauer", 30, "Computer", "Computer Engneering", "HPC");
    cout << "Doctor size: " << sizeof(doc) << endl;
    Test* pTest = (Test*)&doc;
    cout << pTest->vptr1 << endl;
    cout << pTest->name1 << endl;
    cout << pTest->age1 << endl;
    cout << pTest->research << endl;
    cout << pTest->vptr2 << endl;
    cout << pTest->name2 << endl;
    cout << pTest->age2 << endl;
```

```
    cout << pTest->major << endl;
    cout << pTest->subject << endl;

    return 0;
}

// output:
// Doctor size: 28
// 0x405370
// Bauer_1
// 31
// Computer
// 0x40537c
// Bauer_2
// 32
// Computer Engneering
// HPC
```

虚继承是解决C多重继承问题的一种手段，虚继承的底层实现原理与C编译器相关，一般通过虚基类指针和虚基类表实现，每个虚继承的子类都有一个虚基类指针（占用一个指针的存储空间，4（8）字节）和虚基类表（不占用类对象的存储空间）（虚基类依旧会在子类里面存在拷贝，只是仅仅最多存在一份）；当虚继承的子类被当做父类继承时，虚基类指针也会被继承。

在虚继承情况下，底层子类对象的布局不同于普通继承，需要多出一个指向中间层父类对象的虚基类表指针`vbptr`。

`vbptr`是虚基类表指针（virtual base table pointer），`vbptr`指针指向一个虚基类表（virtual table），虚基类表存储了虚基类相对直接继承类的偏移地址；通过偏移地址可以找到虚基类成员，虚继承不用像普通多继承维持着公共基类（虚基类）的两份同样的拷贝，节省了存储空间。

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

using namespace std;

class People
{
public:
    People(string name, int age)
    {
        m_name = name;
        m_age = age;
    }
    void print()
    {
        cout << "this: " << this <<endl;
    }
private:
    string m_name;
    int m_age;
};

class Teacher : virtual public People
{
    string m_research;
public:
```

```
    Teacher(string name, int age, string research):People(name + "_1", age + 1)
    {
        m_research = research;
    }
    void print()
    {
        cout << "this: " << this <<endl;
    }
};

class Student : virtual public People
{
    string m_major;
public:
    Student(string name, int age,string major):People(name + "_2", age + 2)
    {
        m_major = major;
    }
    void print()
    {
        cout << "this: " << this <<endl;
    }
};

class Doctor : public Teacher, public Student
{
    string m_subject;
public:
    Doctor(string name, int age,string research, string major, string subject):
        People(name, age),Teacher(name, age,research),Student(name, age, major)
    {
        m_subject = subject;
    }
};

struct Test
{
    void* vbptr_left;
    string research;
    void* vbptr_right;
    string major;
    string subject;
    string name;
    int age;
};

int main(int argc, char *argv[])
{
    Doctor doc("Bauer", 30, "Computer", "Computer Engneering", "HPC");
    cout << "Doctor size: " << sizeof(doc) << endl;
    Test* pTest = (Test*)&doc;
    cout << pTest->vbptr_left << endl;
    cout << *(int*)pTest->vbptr_left << endl;
    cout << pTest->research << endl;
    cout << pTest->vbptr_right << endl;
    cout << *(int*)pTest->vbptr_right << endl;
    cout << pTest->major << endl;
    cout << pTest->subject << endl;
```

```
    cout << pTest->name << endl;
    cout << pTest->age << endl;

    return 0;
}

// output:
// Doctor size: 28
// 0x40539c
// 12
// Computer
// 0x4053a8
// 0
// Computer Engneering
// HPC
// Bauer
// 30
```

上述代码没有虚函数，在G++编译器打印结果如上，底层子类对象的内存布局如下：


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

using namespace std;

class People
{
public:
    People(string name, int age)
    {
        m_name = name;
        m_age = age;
    }
    virtual void print()
    {
        cout << "this: " << this <<endl;
    }
private:
    string m_name;
    int m_age;
```

```
};

class Teacher : virtual public People
{
    string m_research;
public:
    Teacher(string name, int age, string research):People(name + "_1", age + 1)
    {
        m_research = research;
    }
    void print()
    {
        cout << "this: " << this <<endl;
    }
    virtual void func1()
    {}
};

class Student : virtual public People
{
    string m_major;
public:
    Student(string name, int age,string major):People(name + "_2", age + 2)
    {
        m_major = major;
    }
    void print()
    {
        cout << "this: " << this <<endl;
    }
    virtual void func2()
    {}
};

class Doctor : public Teacher, public Student
{
    string m_subject;
public:
    Doctor(string name, int age,string research, string major, string subject):
        People(name, age),Teacher(name, age,research),Student(name, age, major)
    {
        m_subject = subject;
    }
    void print()
    {
        cout << "this: " << this <<endl;
    }
    virtual void func3()
    {}
};

struct Test
{
    void* vbptr_left;
    char* research;
    void* vbptr_right;
    char* major;
    char* subject;
```

```
    void* vptr_base;
    char* name;
    long age;
};

int main(int argc, char *argv[])
{
    Doctor doc("Bauer", 30, "Computer", "Computer Engneering", "HPC");
    cout << "Doctor size: " << sizeof(doc) << endl;
    Test* pTest = (Test*)&doc;
    cout << pTest->vbptr_left << endl;
    cout << std::hex << *(int*)pTest->vbptr_left << endl;
    cout << std::dec << *((int*)pTest->vbptr_left+8) << endl;
    cout << std::dec << *((int*)pTest->vbptr_left+16) << endl;
    cout << std::dec << *((int*)pTest->vbptr_left+24) << endl;

    cout << pTest->research << endl;
    cout << pTest->vbptr_right << endl;

    cout << pTest->major << endl;
    cout << pTest->subject << endl;
    cout << pTest->vptr_base << endl;

    cout << pTest->name << endl;
    cout << pTest->age << endl;

    return 0;
}
```

上述代码中，使用了虚继承，因此不同的C编译器实现原理不同。

对于`GCC`编译器，People对象大小为`char* + int + 虚函数表指针`，Teacher对象大小为`char*+虚基类表指针+A类型的大小`，Student对象大小为`char*+虚基类表指针+A类型的大小`，Doctor对象大小为`char* + int +虚函数表指针+char*+虚基类表指针+char*+虚基类表指针+char*`。中间层父类共享顶层父类的虚函数表指针，没有自己的虚函数表指针，虚基类指针不共享，因此都有自己独立的虚基类表指针。

`VC、GCC和Clang`编译器的实现中，不管是否是虚继承还是有虚函数，其虚基类指针都不共享，都是单独的。对于虚函数表指针，`VC`编译器根据是否为虚继承来判断是否在继承关系中共享虚表指针。如果子类是虚继承拥有虚函数父类，且子类有新加的虚函数时，子类中则会新加一个虚函数表指针；`GCC`编译器和`Clang`编译器的虚函数表指针在整个继承关系中共享的。
G编译器对于类的内存分布和虚函数表信息命令如下：

> g++ -fdump-class-hierarchy main.cpp
> cat main.cpp.002t.class

`VC++`编译器对于类的内存分布和虚函数表信息命令如下：

> cl main.cpp /d1reportSingleClassLayoutX

`Clang`编译器对于类的内存分布和虚函数表信息命令如下：

> clang -Xclang -fdump-record-layouts

5、构造与析构

C标准规定，每个类都必须有构造函数，如果开发人员没有定义，则C编译器会提供一个默认的构造函数，默认构造函数不带任何参数，也不会对成员数据进行初始化。如果类中定义了任何一种形式的构造函数，C编译器将不再生成默认构造函数。

除了构造函数，C标准规定，每个类都必须有拷贝构造函数，如果开发人员没有定义，则C++编译器会提供一个默认的拷贝构造函数，默认拷贝构造函数是浅拷贝，即按照对象的内存空间逐个字节进行拷贝，因此默认拷贝构造函数会带来隐含的内存问题。

```
#include <stdio.h>
#include <stdlib.h>

class SimpleObject
{
public:
    int n;
    SimpleObject(int n)
    {
        this->n = n;
        buffer = new char[n];
        printf("SimpleObject Created\n");
    }
    virtual ~SimpleObject()
    {
        if(buffer != NULL)
        {
            delete buffer;
            printf("SimpleObject Destroyed\n");
        }
    }
private:
    //SimpleObject(const SimpleObject& another);
private:
    char* buffer;
};

int main()
{
    SimpleObject a(10);
    SimpleObject b = a;
    printf("Object size: %d\n", a.n);

    return 0;
}
```

在`RHEL 7.3`系统使用`GCC`编译器编译运行时会异常退出。

`SimpleObject`在构造时分配了n个字节的缓冲区，在析构时释放缓冲区。但由于没有定义拷贝构造函数，C++编译器会提供一个浅拷贝的默认拷贝构造函数，`SimpleObject b = a`语句会通过浅拷贝构造一个`SimpleObject`对象b，对象b的`buffer`和对象a的`buffer`指向同一块内存空间，在对象a和对象b析构时，这块内存空间被释放了两次，造成程序崩溃。如果不想通过赋值或拷贝构造函数构造对象，可以将拷贝构造函数定义为`private`，此时`SimpleObject b = a`;会在编译时报错。

C性能优化（三）——C语言特性性能分析

一、C++语言特性性能分析简介

通常大多数开发人员认为，汇编语言和C语言比较适合编写对性能要求非常高的程序，C语言主要适用于编写复杂度非常高但性能要求并不是很高的程序。因为大多数开发人员认为，C语言设计时因为考虑到支持多种编程模式（如面向对象编程和范型编程）以及异常处理等，从而引入了太多新的语言特性。新的语言特性往往使得C编译器在编译程序时插入了很多额外的代码，会导致最终生成的二进制代码体积膨胀，而且执行速度下降。

但事实并非如此，通常一个程序的速度在框架设计完成时大致已经确定，而并非因为采用C语言才导致速度没有达到预期目标。因此，当一个程序的性能需要提高时，首先需要做的是用性能检测工具对其运行的时间分布进行一个准确的测量，找出关键路径和真正的性能瓶颈所在，然后针对性能瓶颈进行分析和优化，而不是主观地将性能问题归咎于程序所采用的语言。工程实践表明，如果框架设计不做修改，即使使用C语言或汇编语言重新改写，也并不能保证提高总体性能。

因此，遇到性能问题时，首先应检查和反思程序的总体架构，然后使用性能检测工具对其实际运行做准确的测量，再针对性能瓶颈进行分析和优化。

但C语言中确实有一些操作、特性比其它因素更容易成为程序的性能瓶颈，常见因素如下：
（1）缺页

缺页通常意味着要访问外部存储，因为外部存储访问相对于访问内存或代码执行，有数量级的差别。因此，只要有可能，应该尽量想办法减少缺页。

（2）从堆中动态申请和释放内存

C语言中的`malloc/free`和C语言中的`new/delete`操作时非常耗时的，因此要尽可能优先考虑从线程栈中获取内存。优先考虑栈而减少从动态堆中申请内存，不仅因为在堆中分配内存比在栈中要慢很多，而且还与尽量减少缺页有关。当程序执行时，当前栈帧空间所在的内存页肯定在物理内存中，因此程序代码对其中变量的存取不会引起缺页；如果从堆空间生成对象，只有指向对象的指针在栈上，对象本身则存储在堆空间中。堆一般不可能都在物理内存中，而且由于堆分配内存的特性，即使两个相邻生成的对象，也很有可能在堆内存位置上相距很远。因此，当访问两个对象时，虽然分别指向两个对象的指针都在栈上，但通过两个指针引用对象时很有可能会引起两次缺页。

（3）复杂对象的创建和销毁

复杂对象的创建和销毁会比较耗时，因此对于层次较深的递归调用需要重点关注递归内部的对象创建。其次，编译器生成的临时对象因为在程序的源码中看不到，更不容易察觉，因此需要重点关注。

（4）函数调用

由于函数调用有固定的额外开销，因此当函数体的代码量相对较少，并且函数被非常频繁调用时，函数调用时的固定开销容易成为不必要的开销。C语言的宏和C语言的内联函数都是为了在保持函数调用的模块化特征基础上消除函数调用的固定额外开销而引入的。由于C语言的宏在×××能优势的同时也给开发和调试带来不便，因此C语言中推荐使用内联函数。

二、构造函数与析构函数

1、构造函数与析构函数简介

构造函数和析构函数的特点是当创建对象时自动执行构造函数；当销毁对象时，析构函数自动被执行。构造函数是一个对象最先被执行的函数，在创建对象时调用，用于初始化对象的初始状态和取得对象被使用前需要的一些资源，如文件、网络连接等；析构函数是一个对象最后被执行的函数，用于释放对象拥有的资源。在对象的生命周期内，构造函数和析构函数都只会执行一次。

创建一个对象有两种方式，一种是从线程运行栈中创建，称为局部对象。销毁局部对象并不需要程序显示地调用析构函数，而是当程序运行出对象所属的作用域时自动调用对象的析构函数。

创建对象的另一种方式是从全局堆中动态分配，通常使用`new`或`malloc`分配堆空间。

```
Obejct* p = new Object();//1
// do something //2
delete p;//3
p = NULL;//4
```

执行语句1时，指针`p`所指向对象的内存从全局堆空间中获得，并将地址赋值给`p`，`p`本身是一个局部变量，需要从线程栈中分配，`p`所指向对象从全局堆中分配内存存放。从全局堆中创建的对象需要显示调用`delete`进行销毁，`delete`会调用指针p指向对象的析构函数，并将对象所占的全局堆内存空间返回给全局堆。执行语句3后，指针`p`指向的对象被销毁，但指针`p`还存在于栈中，直到程序退出其所在作用域。将`p`指针所指向对象销毁后，`p`指针仍指向被销毁对象的全局堆空间位置，此时指针`p`变成一个悬空指针，此时使用指针`p`是危险的，通常推荐将`p`赋值`NULL`。

在`Win32`平台，访问销毁对象的全局堆空间内存会导致三种情况：

（1）被销毁对象所在的内存页没有任何对象，堆管理器已经将所占堆空间进一步回收给操作系统，此时通过指针访问会引起访问违例，即访问了不合法内存，引起进程崩溃。

（2）被销毁对象所在的内存页存在其它对象，并且被销毁对象曾经占用的全局堆空间被回收后尚未分配给其它对象，此时通过指针`p`访问取得的值是无意义的，虽然不会立刻引起进程崩溃，但针对指针`p`的后续操作行为是不可预测的。

（3）被销毁对象所在的内存页存在其它对象，并且被销毁对象曾经占用的全局堆空间被回收后已经分配给其它对象，此时通过指针`p`取得的值是其它对象，虽然对指针`p`的访问不会引起进程崩溃，但极有可能引起对象状态的改变。

2、对象的构造过程

创建一个对象分为两个步骤，即首先取得对象所需的内存（从线程栈或全局堆），然后在内存空间上执行构造函数。在构造函数构建对象时，构造函数也分为两个步骤。第一步执行初始化（通过初始化参数列表），第二步执行构造函数的函数体。

```
class Derived : public Base
{
public:
    Derived(): id(1), name("UnNamed")    // 1
    {
        // do something      // 2
    }
private:
    int id;
    string name;
};
```

语句1中冒号后的代码即为初始化列表，每个初始化单元都是变量名（值）的模式，不同单元之间使用逗号分隔。构造函数首先根据初始化列表执行初始化，然后执行构造函数的函数体（语句2）。初始化操作的注意事项如下：

（1）构造函数其实是一个递归操作，在每层递归内部的操作遵循严格的次序。递归模式会首先执行父类的构造函数（父类的构造函数操作也相应包含执行初始化和执行构造函数函数体两个部分），父类构造函数返回后构造类自己的成员变量。构造类自己的成员变量时，一是严格按照成员变量在类中的声明顺序进行，与成员变量在初始化列表中出现的顺序完全无关；二是当有些成员变量或父类对象没有在初始化列表出现时，仍然在初始化操作中对其进行初始化，内建类型成员变量被赋值给一个初值，父类对象和类成员变量对象被调用其默认构造函数初始化，然后父类的构造函数和子成员变量对象在构造函数执行过程中也遵循上述递归操作，直到类的继承体系中所有父类和父类所含的成员变量都被构造完成，类的初始化操作才完成。

（2）父类对象和一些成员变量没有出现在初始化列表中时，其仍然会被执行默认构造函数。因此，相应对象所属类必须提供可以调用的默认构造函数，为此要求相应的类必须显式提供默认构造函数，要么不能阻止编译器隐式生成默认构造函数，定义除默认构造函数外的其它类型的构造函数将会阻止编译器生成默认构造函数。如果编译器在编译时，发现没有可供调用的默认构造函数，并且编译器也无法生成默认构造函数，则编译无法通过。

（3）对两类成员变量，需要强调指出（即常量型和引用型）。由于所有成员变量在执行函数体前已经被构造，即已经拥有初始值，因此，对于常量型和引用型变量必须在初始化列表中正确初始化，而不能将其初始化放在构造函数体内。

（4）初始化列表可能没有完全列出其子成员或父类对象成员，或者顺序与其在类中的声明顺序不同，仍然会保证严格被全部并且严格按照顺序被构建。即程序在进入构造函数体前，类的父类对象和所有子成员变量对象已经被生成和构造。如果在构造函数体内为其执行赋值操作，显然属于浪费。如果在构造函数时已经知道如何为类的子成员变量初始化，则应该将初始化信息通过构造函数的初始化列表赋予子成员变量，而不是在构造函数体内进行初始化，因为进入构造函数时，子成员变量已经初始化一次。

3、对象的析构过程

析构函数和构造函数一样，是递归的过程，但存在不同。一是析构函数不存在初始化操作部分，析构函数的主要工作就是执行析构函数的函数体；二是析构函数执行的递归与构造函数相反，在每一层递归中，成员变量对象的析构顺序也与构造函数相反。

析构函数只能选择类的成员变量在类中声明的顺序作为析构的顺序参考（正序或逆序）。因为构造函数选择了正序，而析构函数的工作与构造函数相反，因此析构函数选择逆序。又因为析构函数只能使用成员变量在类中的声明顺序作为析构顺序的依据（正序或逆序），因此构造函数也只能选择成员变量在类中的声明顺序作为构造的顺序依据，而不能采用初始化列表的顺序作为顺序依据。
如果操作的对象属于一个复杂继承体系的末端节点，其析构过程也将十分耗时。

在C++程序中，创建和销毁对象是影响性能的一个非常突出的操作。首先，如果是从全局堆空间中生成对象，则需要先进行动态内存分配操作，而动态内存的分配与回收是非常耗时的操作，因为涉及到寻找匹配大小的内存块，找到后可能还需要截断处理，然后还需要修改维护全局堆内存使用情况信息的链表。频繁的内存操作会严重影响性能的下降，使用内存池技术可以减少从全局动态堆空间申请内存的次数，提高程序的总体性能。当取得内存后，如果需要生成的内对象属于复杂继承体系的末端类，则构造函数的调用将会引起一连串的递归构造操作，在大型复杂系统中，大量的此类对象构造将会消耗CPU操作的主要部分。

由于对象的创建和销毁会影响性能，在尽量减少自己代码生成对象的同时，需要关注编译器在编译时临时生成的对象，尽量避免临时对象的生成。

如果在实现构造函数时，在构造函数体中进行了第二次的赋值操作，也会浪费CPU时间。

4、函数参数传递

减少对象创建和销毁的常见方法是在声明中将所有的值传递改为常量引用传递，如：

```
int func(Object obj);// 1
int func(const Object& obj);// 2
```

值传递验证示例如下：

```
#include <iostream>

using namespace std;

class Object
{
```

```
public:
    Object(int i = 1)
    {
        n = i;
        cout << "Object(int i = 1): " << endl;
    }
    Object(const Object& another)
    {
        n = another.n;
        cout << "Object(const Object& another): " << endl;
    }
    void increase()
    {
        n++;
    }
    int value()const
    {
        return n;
    }
    ~Object()
    {
        cout << "~Object()" << endl;
    }
private:
    int n;
};
void func(Object obj)
{
    cout << "enter func, before increase(), n = " << obj.value() << endl;
    obj.increase();
    cout << "enter func, after increase(), n = " << obj.value() << endl;
}
int main()
{
    Object a;   // 1
    cout << "before call func, n = " << a.value() << endl;
    func(a);    // 2
    cout << "after call func, n = " << a.value() << endl;// 3

    return 0;
}
// output:
//Object(int i = 1):        // 4
//before call func, n = 1
//Object(const Object& another):    // 5
//enter func, before increase(), n = 1  // 6
//enter func, after increase(), n = 2   // 7
//~Object() // 8
//after call func, n = 1    // 9
//~Object()
```

语句4的输出为语句1处的对象构造，语句5输出则是语句2处的`func(a)`函数调用，调用开始时通过拷贝构造函数生成对象a的复制品，紧跟着在函数内检查n的输出值输出语句6，输出值与`func`函数外部元对象a的值相同，然后复制品调用`increase`函数将n值加1，此时复制品的n值为2，并输出语句7。`func`函数执行完毕后销毁复制品，输出语句8。`main`函数内继续执行，打印原对象a的n值为1，输出语句9。

当函数需要修改传入参数时，应该用引用传入参数；当函数不会修改传入参数时，如果函数声明中传入参数为对象，则函数可以达到设计目的，但会生成不必要的复制品对象，从而引入不必要的构造和析构操作，应该使用常量引用传入参数。

构造函数的重复赋值对性能影响验证示例如下：

```
#include <iostream>
#include <time.h>

using namespace std;

class DArray
{
public:
    DArray(double v = 1.0)
    {
        for(int i = 0; i < 1000; i++)
        {
            d[i] = v + i;
        }
    }
    void init(double v = 1.0)
    {
        for(int i = 0; i < 1000; i++)
        {
            d[i] = v + i;
        }
    }
private:
    double d[1000];
};

class Object
{
public:
    Object(double v)
    {
        d.init(v);
    }
private:
    DArray d;
};

int main()
{
    clock_t start, finish;
    start = clock();
    for(int i = 0; i < 100000; i++)
    {
        Object obj(2.0 + i);
    }
    finish = clock();
    cout << "Used Time: " << double(finish - start) << "" << endl;

    return 0;
}
```

耗时为600000单位，如果通过初始化列表对成员变量进行初始化，其代码如下：

```
#include <iostream>
#include <time.h>

using namespace std;

class DArray
{
public:
    DArray(double v = 1.0)
    {
        for(int i = 0; i < 1000; i++)
        {
            d[i] = v + i;
        }
    }
    void init(double v = 1.0)
    {
        for(int i = 0; i < 1000; i++)
        {
            d[i] = v + i;
        }
    }
private:
    double d[1000];
};

class Object
{
public:
    Object(double v): d(v)
    {
    }
private:
    DArray d;
};

int main()
{
    clock_t start, finish;
    start = clock();
    for(int i = 0; i < 100000; i++)
    {
        Object obj(2.0 + i);
    }
    finish = clock();
    cout << "Used Time: " << double(finish - start) << "" << endl;

    return 0;
}
```

耗时为300000单位，性能提高约50%。

三、继承与虚函数

1、虚函数与动态绑定机制

虚函数是C++语言引入的一个重要特性，提供了动态绑定机制，动态绑定机制使得类继承的语义变得相对明晰。

（1）基类抽象了通用的数据及操作。对于数据而言，如果数据成员在各个派生类中都需要用到，需要将其声明在基类中；对于操作而语言，如果操作对于各个派生类都有意义，无论其语义是否会被修改和扩展，需要将其声明在基类中。

（2）某些操作，对于各个派生类而言，语义完全保持一致，而无需修改和扩展，则相应操作声明为基类的非虚成员函数。各个派生类在声明为基类的派生类时，默认继承非虚成员函数的声明和实现，如果默认继承基类的数据成员一样，而不必另外做任何声明，构成代码复用。

（3）对于某些操作，虽然对于各个派生类都有意义，但其语义并不相同，则相应的操作应该声明为虚成员函数。各个派生类虽然也继承了虚成员函数的声明和实现，但语义上应该对虚成员函数的实现进行修改或扩展。如果在实现修改、扩展虚成员函数的过程中，需要用到额外的派生类独有的数据时，则将相应的数据声明为派生类自己的数据成员。

当更高层次的程序框架（继承体系的使用者）使用此继承体系时，处理的是抽象层次的对象集合，对象集合的成员本质是各种派生类对象，但在处理对象集合的对象时，使用的是抽象层次的操作。高层程序框架并不区分相应操作中哪些操作对于派生类是不变的，哪些操作对于派生类是不同的，当实际执行到各操作时，运行时系统能够识别哪些操作需要用到动态绑定。从而找到对应派生类的修改或扩展的操作版本。即对继承体系的使用者而言，继承体系内部的多样性是透明的，不必关心其继承细节，处理的是一组对使用者而言整体行为一致的对象。即使继承体系内部增加、删除了某个派生类，或某个派生类的虚函数实现发生了改变，使用者的代码也不必做任何修改，使程序的模块化程度得到极大提高，其扩展性、维护性和代码可读性也会提高。对于对象继承体系使用者而言，只看到抽象类型，而不必关心具体是哪种具体类型。

2、虚函数的效率分析

虚函数的动态绑定特性虽然很好，但存在内存空间和时间开销，每个支持虚函数的类（基类或派生类）都会有一个包含其所有支持的虚函数的虚函数表的指针。每个类对象都会隐含一个虚函数表指针（virtual pointer），指向其所属类的虚函数表。当通过基类的指针或引用调用某个虚函数时，系统需要首先定位指针或引用真正对应的对象所隐含的虚函数指针，然后虚函数指针根据虚函数的名称对其所指向的虚函数表进行一个偏移定位，再调用偏移定位处的函数指针对应的虚函数，即动态绑定的解析过程。C++规范只需要编译器能够保证动态绑定的语义，但大多数编译器都采用上述方法实现虚函数。

（1）每个支持虚函数的类都有一个虚函数表，虚函数表的大小与类拥有的虚函数的多少成正比。一个程序中，每个类的虚函数表只有一个，与类对象的数量无关。支持虚函数的类的每个类对象都有一个指向类的虚函数表的虚函数指针，因此程序运行时虚函数指针引起的内存开销与生成的类对象数量成正比。

（2）支持虚函数的类生成每个对象时，在构造函数中会调用编译器在构造函数内部插入的初始化代码，来初始化其虚函数指针，使其指向正确的虚函数表。当通过指针或引用调用虚函数时，会根据虚函数指针找到相应类的虚函数表。

3、虚函数与内联

内联函数通常可以提高代码执行速度，很多普通函数会根据情况进行内联化，但虚函数无法利用内联化的优势。因为内联是在编译阶段编译器将调用内联函数的位置用内联函数体替代（内联展开），但虚函数本质上是运行期行为。在编译阶段，编译器无法知道某处的虚函数调用在真正执行的时后需要调用哪个具体的实现（即编译阶段无法确定其具体绑定），因此，编译阶段编译器不会对通过指针或引用调用的虚函数进行内联化。如果需要利用虚函数的动态绑定的设计优势，必须放弃内联带来的速度优势。

如果不使用虚函数，可以通过在抽象基类增加一个类型标识成员用于在运行时识别具体的派生类对象，在派生类对象构造时必须指定具体的类型。继承体系的使用者调用函数时不再需要一次间接地根据虚函数表查找虚函数指针的操作，但在调用前仍然需要使用switch语句对其类型进行识别。
因此虚函数的缺点可以认为只有两条，即虚函数表的空间开销以及无法利用内联函数的速度优势。由于每个含有虚函数的类在整个程序只有一个虚函数表，因此虚函数表引起的空间开销时非常小的。所以，可以认为虚函数引入的性能缺陷只是无法利用内联函数。

通常，非虚函数的常规设计假如需要增加一种新的派生类型，或者删除一种不再支持的派生类型，都必须修改继承体系所有使用者的所有与类型相关的函数调用代码。对于一个复杂的程序，某个继承体系的使用者会很多，每次对继承体系的派生类的修改都会波及使用者。因此，不使用虚函数的常规设计增加了代码的耦合度，模块化不强，导致项目的可扩展性、可维护性、代码可读性都会降低。面向对象编程的一个重要目的就是增加程序的可扩展性和可维护性，即当程序的业务逻辑发生改变时，对原有程序的修改非常方便，降低因为业务逻辑改变而对代码修改时出错的概率。

因此，在性能和其它特性的选择方面，需要开发人员根据实际情况进行进行权衡和取舍，如果性能检验确认性能瓶颈不是虚函数没有利用内联的优势引起，可以不必考虑虚函数对性能的影响。

四、临时对象

1、临时对象简介

对象的创建与销毁对程序的性能影响很大，尤其是对象的类处于一个复杂继承体系的末端，或者对象包含很多成员对象（包括其所有父类对象，即直接或者间接父类的所有成员变量对象）时，对程序性能影响尤其显著。因此，作为一个对性能敏感的程序员，应该尽量避免创建不必要的对象，以及随后的销毁。除了减少显式地创建对象，也要尽量避免编译器隐式地创建对象，即临时对象。

```
#include <iostream>
#include <cstring>

class Matrix
{
public:
    Matrix(double d = 1.0)
    {
        for(int i = 0; i < 10; i++)
        {
            for(int j = 0; j < 10; j++)
            {
                m[i][j] = d;
            }
        }
        cout << "Matrix(double d = 1.0)" << endl;
    }
    Matrix(const Matrix& another)
    {
        cout << "Matrix(const Matrix& another)" << endl;
        memcpy(this, &another, sizeof(another));
    }

    Matrix& operator=(const Matrix& another)
    {
        if(this != &another)
        {
            memcpy(this, &another, sizeof(another));
        }
        cout << "Matrix& operator=(const Matrix& another)" << endl;
```

```
        return *this;
    }
    friend const Matrix operator+(const Matrix& m1, const Matrix& m2);
private:
    double m[10][10];
};

const Matrix operator+(const Matrix& m1, const Matrix& m2)
{
    Matrix sum; // 1
    for(int i = 0; i < 10; i++)
    {
        for(int j = 0; j < 10; j++)
        {
            sum.m[i][j] = m1.m[i][j] + m2.m[i][j];
        }
    }
    return sum; // 2
}

int main()
{
    Matrix a(2.0), b(3.0), c; // 3
    c = a + b; // 4
    return 0;
}
```

由于`GCC`编译器默认进行了返回值优化（`Return Value Optimization`，简称`RVO`），因此需要指定`-fno-elide-constructors`选项进行编译：

> g++ -fno-elide-constructors main.cpp

输出结果如下：

> Matrix(double d = 1.0) // 1
> Matrix(double d = 1.0) // 2
> Matrix(double d = 1.0) // 3
> Matrix(double d = 1.0) // 4
> Matrix(const Matrix& another) // 5
> Matrix& operator=(const Matrix& another) // 6

分析代码，语句3生成3个`Matrix`对象，调用3次构造函数，语句4调用`operator+`执行到语句1时生成临时变量`sum`，调用1次构造函数，语句4调用赋值操作，不会生成新的`Matrix`对象。输出5则是因为`a+b`调用`operator+`函数时需要返回一个`Matrix`变量`sum`，然后进一步通过`operator=`函数将`sum`变量赋值给变量`c`，但`a+b`返回时，`sum`变量已经被销毁，即在`operator+`函数调用结束时被销毁，其返回的`Matrix`变量需要在调用`a+b`函数的栈中开辟空间来存放，临时的`Matrix`对象是在`a+b`返回时通过`Matrix`拷贝构造函数构造，即输出5打印。

如果使用默认`GCC`编译选项编译，`GCC`编译器默认会进行返回值优化。

> g++ main.cpp

程序输出如下：

> Matrix(double d = 1.0)
> Matrix(double d = 1.0)
> Matrix(double d = 1.0)
> Matrix(double d = 1.0)

> Matrix& operator=(const Matrix& another)

临时对象与临时变量并不相同。通常，临时变量是指为了暂时存放某个值的变量，显式出现在源码中；临时对象通常指编译器隐式生成的对象。

临时对象在C++语言中的特征是未出现在源代码中，而是从栈中产生未命名对象，开发人员并没有声明要使用临时对象，由编译器根据情况产生，通常开发人员不会注意到其产生。

返回值优化（`Return Value Optimization`，简称`RVO`）是一种优化机制，当函数需要返回一个对象的时候，如果自己创建一个临时对象用户返回，那么临时对象会消耗一个构造函数（`Constructor`）的调用、一个复制构造函数的调用（`Copy Constructor`）以及一个析构函数（`Destructor`）的调用的代价，而如果稍微做一点优化，就可以将成本降低到一个构造函数的代价。

2、临时对象生成

通常，产生临时对象的场合如下：

（1）当实际调用函数时传入的参数与函数定义中声明的变量类型不匹配。

（2）当函数返回一个对象时。

在函数传递参数为对象时，实际调用时因为函数体内的对象与实际传入的对象并不相同，而是传入对象的拷贝，因此有开发者认为函数体内的拷贝对象也是一个临时对象，但严格来说，函数体内的拷贝对象并不符合未出现在源码中。

对于类型不匹配生成临时对象的情况，示例如下：

```
#include <iostream>

using namespace std;
class Rational
{
public:
    Rational(int a = 0, int b = 1): real(a), imag(b)    // 1
    {
        cout << " Rational(int a = 0, int b = 0)" << endl;
    }
private:
    int real;
    int imag;
};

void func()
{
    Rational r;
    r = 100;  // 2
}

int main()
{
    func();
    return 0;
}
```

执行语句2时，由于`Rational`没有重载`operator=(int i)`，编译器会合成一个`operator=(const Rational& another)`函数，并执行逐位拷贝赋值操作，但由于100不是一个`Rational`对象，但编译器会尽可能查找合适的转换路径，以满足编译的需要。编译器发现存在一个`Rational(int a = 0, int b = 1)`构造函数，编译器会将语句2右侧的100通过`Rational100, 1)`生成一个临时对象，然后用编译器合成的`operator=(const Rational& another)`函数进行逐位赋值，语句2执行后，`r`对象内部的`real`为100，`img`为1。

C编译器为了成功编译某些语句会生成很多从源码中不易察觉的辅助函数，甚至对象。C编译器提供的自动类型转换确实提高了程序的可读性，简化了程序编写，提高了开发效率。但类型转换意味着临时对象的产生，对象的创建和销毁意味着性能的下降，类型转换还意味着编译器会生成其它的代码。因此，如果不需要编译器提供自动类型转换，可以使用`explicit`对类的构造函数进行声明。

```
#include <iostream>

using namespace std;
class Rational
{
public:
    explicit Rational(int a = 0, int b = 1): real(a), imag(b)    // 1
    {
        cout << " Rational(int a = 0, int b = 0)" << endl;
    }
private:
    int real;
    int imag;
};

void func()
{
    Rational r; // 2
    r = 100;    // 3
}

int main()
{
    func();
    return 0;
}
```

此时，进行代码编译会报错：

> error: no match for ‘operator=’ (operand types are ‘Rational’ and ‘int’)

错误信息提示没有匹配的`operator=`函数将`int`和`Rational`对象进行转换。C++编译器默认合成的`operator=`函数只接受`Rational`对象，不能接受`int`类型作为参数。要想代码编译能够通过，方法一是提供一个重载的`operator=`赋值函数，可以接受整型作为参数；方法二是能够将整型转换为`Rational`对象，然后进一步利用编译器合成的赋值运算符。将整型转换为`Rational`对象，可以提供能只传递一个整型作为参数的`Rational`构造函数，考虑到缺省参数，调用构造函数可能会是无参、一个参数、两个参数，此时编译器可以利用整型变量作为参数调用`Rational`构造函数生成一个临时对象。由于`explicit`关键字限定了构造函数只能被显示调用，不允许编译器运用其进行类型转换，此时编译器不能使用构造函数将整型100转换为`Rational`对象，所以导致编译报错。

通过重载以整型作为参数的`operator=`函数可以成功编译，代码如下：

```
#include <iostream>

using namespace std;
class Rational
{
public:
    explicit Rational(int a = 0, int b = 1): real(a), imag(b)    // 1
    {
        cout << " Rational(int a = 0, int b = 0)" << endl;
    }
    Rational& operator=(int r)
    {
        real = r;
        imag = 1;
        return *this;
    }
private:
    int real;
    int imag;
};

void func()
{
    Rational r; // 2
    r = 100;    // 3
}

int main()
{
    func();
    return 0;
}
```

重载`operator=`函数后，编译器可以成功将整型数转换为`Rational`对象，同时成功避免了临时对象产生。

当一个函数返回的是非内建类型的对象时，返回结果对象必须在某个地方存放，编译器会从调用相应函数的栈帧中开辟空间，并用返回值作为参数调用返回值对象所属类型的拷贝构造函数在所开辟的空间生成对象，在调用函数结束并返回后可以继续利用临时对象。

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

using namespace std;
class Rational
{
public:
    Rational(int a = 0, int b = 0): real(a), imag(b)
    {
        cout << " Rational(int a = 0, int b = 0)" << endl;
    }
    Rational(const Rational& another): real(another.real), imag(another.imag)
    {
        cout << " Rational(const Rational& another)" << endl;
    }
    Rational& operator = (const Rational& other)
```

```
    {
        if(this != &other)
        {
            real = other.real;
            imag = other.imag;
        }
        cout << " Rational& operator = (const Rational& other)" << endl;
        return *this;
    }
    friend const Rational operator+(const Rational& a, const Rational& b);
private:
    int real;
    int imag;
};

const Rational operator+(const Rational& a, const Rational& b)
{
    cout << " operator+ begin" << endl;
    Rational c;
    c.real = a.real + b.real;
    c.imag = a.imag + b.imag;
    cout << " operator+ end" << endl;
    return c; // 2
}


int main()
{
Rational r, a(10, 10), b(5, 8);
    r = a + b;// 1
    return 0;
}
```

执行语句1时，相当于在`main`函数中调用`operator+(const Rational& a, const Rational& b)`函数，在`main`函数的栈中会开辟一块`Rational`大小的空间，在`operator+(const Rational& a, const Rational& b)`函数内部的语句2处，函数返回使用被销毁的c对象作为参数调用拷贝构造函数在`main`函数栈中开辟空间生成一个`Rational`对象。然后使用`operator =`执行赋值操作。编译如下：

> g++ -fno-elide-constructors main.cpp

输出如下：

> Rational(int a = 0, int b = 0)
> Rational(int a = 0, int b = 0)
> Rational(int a = 0, int b = 0)
> operator+ begin
> Rational(int a = 0, int b = 0)
> operator+ end
> Rational(const Rational& another)
> Rational& operator = (const Rational& other)

由于`r`对象在默认构造后并没有使用，可以延迟生成，代码如下：

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

```
using namespace std;
class Rational
{
public:
    Rational(int a = 0, int b = 0): real(a), imag(b)
    {
        cout << " Rational(int a = 0, int b = 0)" << endl;
    }
    Rational(const Rational& another): real(another.real), imag(another.imag)
    {
        cout << " Rational(const Rational& another)" << endl;
    }
    Rational& operator = (const Rational& other)
    {
        if(this != &other)
        {
            real = other.real;
            imag = other.imag;
        }
        cout << " Rational& operator = (const Rational& other)" << endl;
        return *this;
    }
    friend const Rational operator+(const Rational& a, const Rational& b);
private:
    int real;
    int imag;
};

const Rational operator+(const Rational& a, const Rational& b)
{
    cout << " operator+ begin" << endl;
    Rational c;
    c.real = a.real + b.real;
    c.imag = a.imag + b.imag;
    cout << " operator+ end" << endl;
    return c; // 2
}


int main()
{
    Rational a(10, 10), b(5, 8);
    Rational r = a + b;  // 1
    return 0;
}
```

编译过程如下：

```
g++ -fno-elide-constructors main.cpp
```

输出如下：

```
Rational(int a = 0, int b = 0)
Rational(int a = 0, int b = 0)
operator+ begin
Rational(int a = 0, int b = 0)
operator+ end
Rational(const Rational& another)
Rational(const Rational& another)
```

分析代码，编译器执行语句1时语义发生了较大变化，编译器对`=`的解释不再是赋值操作符，而是对象`r`的初始化。在取得`a+b`的结果时，在`main`函数栈中开辟空间，使用`c`对象作为参数调用拷贝构造函数生成一个临时对象，然后使用临时对象作为参数调用拷贝构造函数生成`r`对象。

因此，对于非内建对象，尽量将对象延迟到确切直到其有效状态时，可以有效减少临时对象生成。如将`Rational r`; `r = a + b`;改写为`Rational r = a + b`;

进一步，可以将`operator+`函数改写为如下：

```
const Rational operator+(const Rational& a, const Rational& b)
{
    cout << " operator+ begin" << endl;
    return Rational(a.real + b.real, a.imag + b.imag); // 2
}
```

通常，`operator+`与`operator+=`需要以其实现，`Rational`的`operator+=`实现如下：

```
Rational operator+=(const Rational& a)
    {
        real += a.real;
        imag = a.imag;
        return *this;
    }
```

`operator+=`没有产生临时对象，尽量用`operator+=`代替`operator+`操作。考虑到代码复用性，`operator+`可以使用`operator+=`实现，代码如下：

```
const Rational operator+(const Rational& a, const Rational& b)
{
    cout << " operator+ begin" << endl;
    return Rational(a) += b; // 2
}
```

对于前自增操作符实现如下：

```
const Rational operator++()
{
        ++real;
        return *this;
}
```

对于后自增操作如下：

```
const Rational operator++(int)
{
        Rational temp(*this);
        ++(*this);
        return temp;
}
```

前自增只需要将自身返回，后自增需要返回一个对象，因此需要多生成两个对象：函数体内的局部变量和临时对象，因此对于非内建类型，在保证程序语义下尽量使用前自增。

3、临时对象的生命周期

C++规范中定义了临时对象的生命周期从创建时开始，到包含创建它的最长语句执行完毕。

```
string a, b;
const char* str;
if(strlen(str = (a + b).c_str()) > 5) // 1
{
    printf("%s\n", str);// 2
}
```

分析代码，语句1处首先创建一个临时对象存放a+b的值，然后将临时对象的内容通过`c_str`函数得到赋值给`str`，如果`str`长度大于5则执行语句2，但临时对象生命周期在包含其创建的最长语句已经结束，当进入`if`语句块时，临时对象已经被销毁，执行其内部字符串的`str`指向的是一段已经回收的内存，结果是无法预测的。但存在一个特例，当用一个临时对象来初始化一个常量引用时，临时对象的生命周期会持续到与绑定其上的常用引用销毁时。示例代码如下：

```
string a, b;
if(true)
{
    const string& c = a + b; // 1

}
```

语句1将`a+b`结果的临时对象绑定到常量引用`c`，临时对象生命周期会持续到`c`的作用域结束，不会在语句1结束时结束。

五、内联函数

1、C++内联函数简介

C语言的设计中，内联函数的引入完全是为了性能的考虑，因此在编写对性能要求较高的C程序时，极有必要考量内联函数的使用。

内联是将被调用函数的函数体代码直接地整个插入到函数被调用处，而不是通过`call`语句进行。

C++编译器在真正进行内联时，由于考虑到被内联函数的传入参数、自己的局部变量以及返回值的因素，不只进行简单的代码拷贝，还有许多细致工作。

2、C++函数内联的声明

开发人员可以有两种方法告诉C编译器需要内联哪些类成员函数，一种是在类的定义体外，一种是在类的定义体内。

（1）在类的定义体外时，需要在类成员函数的定义前加`inline`关键字，显式地告诉C编译器本函数在调用时需要内联处理。

```
class Student
{
public:
    void setName(const QString& name);
    QString getName()const;
    void setAge(const int age);
    getAge()const;
private:
    QString m_name;
    int m_age;
};

inline void Student::setName(const QString& name)
{
    m_name = name;
}
inline QString Student::getName()const
{
    return m_name;
}
inline void Student::setAge(const int age)
{
    m_age = age;
}
inline Student::getAge()const
{
    return m_age;
}
```

（2）在类的定义体内且声明成员函数时，同时提供类成员函数的实现体。此时，`inline`关键字不是必须的。

```
class Student
{
public:
    void setName(const QString& name)
    {
        m_name = name;
    }
    inline QString getName()const
    {
        return m_name;
    }
    inline void setAge(const int age)
    {
        m_age = age;
    }
    inline getAge()const
```

```
    {
        return m_age;
    }
private:
    QString m_name;
    int m_age;
};
```

（3）普通函数（非类成员函数）需要被内联时，需要在普通函数的定义前加`inline`关键字，显式地告诉C++编译器本函数在调用时需要内联处理。

```
inline int add(int a, int b)
{
    return a + b;
}
```

3、C++内联机制

C是以编译单元为单位编译的，通常一个编译单元基本等同于一个`CPP`文件。在编译的预处理阶段，预处理器会将`#include`的各个头文件（支持递归头文件展开）完整地复制到`CPP`文件的对应位置处，并进行宏展开等操作。预处理器处理后，编译才真正开始。一旦C编译器开始编译，C编译器将不会意识到其它`CPP`文件的存在，因此并不会参考其它`CPP`文件的内容信息。因此，在编译某个编译单元时，如果本编译单元会调用到某个内联函数，那么内联函数的函数定义（函数体）必须包含在编译单元内。因为C编译器在使用内联函数体代码替换内联函数调用时，必须知道内联函数的函数体代码，并且不能通过参考其它编译单元信息获得。

如果多个编译单元会用到同一个内联函数，C++规范要求在多个编译单元中同一个内联函数的定义必须是完全一致的，即`ODR（One Definition Rule）`原则。考虑到代码的可维护性，通常将内联函数的定义放在一个头文件中，用到内联函数的所有编译单元只需要`#include`相应的头文件即可。

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

using namespace std;
class Student
{
public:
    void setName(const string& name)
    {
        m_name = name;
    }
    inline string getName()const
    {
        return m_name;
    }
    inline void setAge(const int age)
    {
        m_age = age;
    }
    inline int getAge()const
    {
        return m_age;
    }
private:
    string m_name;
```

```
    int m_age;
};

void Print()
{
    Student s;
    s.setAge(20);
    cout << s.getAge() << endl;
}

int main()
{
    Print();
    return 0;
}
```

上述代码中，在不开启内联时调用函数`Print`的函数时相关的操作如下：

（1）进入`Print`函数时，从其栈帧中开辟了放置`s`对象的空间。

（2）进入函数体后，首先在开辟的s对象存储空间执行`Student`的默认构造函数构造`s`对象。

（3）将常数20压栈，调用`s`的`setAge`函数（开辟`setAge`函数的栈帧，返回时回退销毁此栈帧）。

（4）执行`s`的`getAge`函数，并将返回值压栈。

（5）调用`cout`操作符操作压栈的结果，即输出。

开启内联后，`Print`函数的等效代码如下：

```
void Print()
{
    Student s;
    {
        s.m_age = 20;
    }
    int tmp = s.m_age;
    cout << tmp << endl;
}
```

函数调用时的参数压栈，栈帧开辟与销毁等操作不再需要，结合内联后代码，编译器会进一步优化为如下结果：

```
int main()
{
    cout << 20 << endl;
    return 0;
}
```

如果不考虑`setAge/getAge`函数内联，对于非内联函数一般不会在头文件中定义，因此`setAge/getAge`函数可能在本编译单元之外的其它编译单元定义，`Print`函数所在的编译单元会看不到`setAge/getAge`，不知道函数体的具体代码信息，不能作出进一步的代码优化。

因此，函数内联的优点如下：

（1）减少因为函数调用引起的开销，主要是参数压栈、栈帧开辟与回收、寄存器保存与恢复。

（2）内联后编译器在处理调用内联函数的函数时，因为可供分析的代码更多，因此编译器能做的优化更深入彻底。

程序的唯一入口`main`函数肯定不会被内联化，编译器合成的默认构造函数、拷贝构造函数、析构函数以及赋值运算符一般都会被内联化。编译器并不保证使用`inline`修饰的函数在编译时真正被内联处理，`inline`只是给编译器的建议，编译其完全会根据实际情况对其忽视。

4、函数调用机制

```
int add(int a, int b)
{
    return a + b;
}

void func()
{
    ...
    int c = add(a, b);
    ...
}
```

函数调用时相关操作如下：
（1）参数压栈

参数是`a，b`；压栈时通常按照逆序压栈，因此是`b，a`；如果参数中有对象，需要先进行拷贝构造。
（2）保存返回地址

即函数调用结束后接着执行的语句的地址。

（3）保存维护`add`函数栈帧信息的寄存器内容，如`SP`（对栈指针），`FP`（栈栈指针）等。具体保存的寄存器与硬件平台有关。

（4）保存某些通用寄存器的内容。由于某些通用寄存器会被所有函数用到，所以在`func`函数调用`add`之前，这些通用寄存器可能已经存储了对`func`有用的信息。但这些通用寄存器在进入`add`函数体内执行时可能会被`add`函数用到，从而被覆写。因此，`func`函数会在调用`add`函数前保存一份这些通用寄存器的内容，在`add`函数返回后恢复。

（5）调用`add`函数。首先通过移动栈指针来分配所有在其内部声明的局部变量所需的空间，然后执行其函数体内的代码。

（6）`add`函数执行完毕，函数返回时，`func`函数需要进行善后处理，如恢复通用寄存器的值，恢复保存`func`函数栈帧信息的寄存器的值，通过移动栈指针销毁`add`函数的栈帧，将保存的返回地址出栈并赋值给`IP`寄存器，通过移动栈指针回收传给`add`函数的参数所占的空间。

如果函数的传入参数和返回值都为对象时，会涉及对象的构造与析构，函数调用的开销会更大。

5、内联的效率分析

因为函数调用的准备与善后工作最终都由机器指令完成，假设一个函数之前的准备工作与之后的善后工作的指令所需的空间为`SS`，执行指令所需的时间为`TS`，从时间和空间分析内联的效率如下：

（1）空间效率。通常认为，如果不采用内联，被调用函数代码只有一份，在调用位置使用`call`语句即可。而采用内联后，被调用函数的代码在所调用的位置都会有一份拷贝，因此会导致代码膨胀。

如果函数 `func` 的函数体代码为 `FuncS`，假设 `func` 函数在整个程序内被调用 `n` 次，不采用内联时，对 `func` 函数的调用只有准备工作与善后工作会增加最后的代码量开销，`func` 函数相关的代码大小为 `n*SS + FuncS`。采用内联后，在各个函数调用位置都需要将函数体代码展开，即 `func` 函数的相关代码大小为 `n*FuncS`。所以需要比较 `n*SS + FuncS` 与 `n*FuncS` 的大小，如果调用次数 `n` 较大，可以简化为比较 `SS` 与 `FuncS` 的大小。如果内联函数自己的函数体代码量比因为函数调用的准备与善后工作引入的代码量大，则内联后程序的代码量会变大；如果内联函数自己的函数体代码量比因为函数调用的准备与善后工作引入的代码量小，则内联后程序的代码量会变小；如果内联后编译器因为获得更多的代码信息，从而对调用函数的优化更深入彻底，则最终的代码量会更小。

（2）时间效率。通常，内联后函数调用都不再需要做函数调用的准备与善后工作，并且由于编译器可以获得更多的代码信息，可以进行深入彻底的代码优化。内联后，调用函体内需要执行的代码是相邻的，其执行的代码都在同一个页面或连续的页面中。如果没有内联，执行到被调用函数时，需要调转到包含被调用函数的内存页面中执行，而被调用函数的所属的页面极有可能当时不在物理内存中。因此，内联后可以降低缺页的概率，减少缺页次数的效果远比减少一些代码量执行的效果要好。即使被调用函数所在页面也在内存中，但与调用函数在空间上相隔甚远，可能会引起cache miss，从而降低执行速度。因此，内联后程序的执行时间会比没有内联要少，即程序执行速度会更快。但如果FunS远大于SS，且n较大，最终程序的大小会比没有内联大的多，用来存放代码的内存页也会更多，导致执行代码引起的缺页也会增多，此时，最终程序的执行时间可能会因为大量的缺页变得更多，即程序变慢。因此，很多编译器会对函数体代码很多的函数拒绝其内联请求，即忽略inine关键字，按照非内联函数进行编译。

因此，是否采用内联时需要根据内联函数的特征（如函数体代码量、程序被调用次数等）进行判断。判断内联效果的最终和最有效方法还是对程序执行速度和程序大小进行测量，然后根据测量结果决定是否采用内联和对哪些函数进行内联。

6、内联函数的二进制兼容问题

调用内联函数的编译单元必须具有内联函数的函数体代码信息，考虑到 `ODR` 规则和代码可维护性，通常将内联函数的定义放在头文件中，每个调用内联函数的编译单元通过 `#include` 相应头文件。

在大型软件中，某个内联函数因为比较通用，可能会被大多数编译单元用到，如果对内联函数进行修改会引起所有用到该内联函数的编译单元进行重新编译。对于大型程序，重新编译大部分编译单元会消耗大量的编译时间，因此，内联函数最好在开发的后期引入，以避免可能不必要的大量编译时间浪费。

如果某开发组使用了第三方提供的程序库，而第三方程序库中可能包含内联函数，因此在开发组代码中使用了第三方库的内联函数位置都会将内联函数体代码拷贝到函数调用位置。如果第三方库提供商在下一个版本中修改了某些内联函数的定义，即使没有修改任何函数的对外接口，开发组想要使用新版本的第三方库仍然需要重新编译。如果程序已经发布，则重新编译的成本会极高。如果没有内联，第三方库提供商只是修改了函数实现，开发组不必重新编译即可使用最新的第三方库版本。

7、递归函数的内联

内联的本质是使用函数体代码对函数调用进行替换，对于递归函数：

```
int sum(int n)
{
    if(n < 2)
    {
        return 1;
    }
    else
    {
        return sum(n - 1) + n;
    }
}
```

如果某个编译单元内调用了`sum`函数，如下：

```
void func()
{
    ...
    int ret = sum(n);
    ...
}
```

如果在编译本编译单元且调用`sum`函数时，提供的参数`n`不能够知道实际值，则编译器无法知道对`sum`函数进行了多少次替换，编译器会拒绝对递归函数`sum`进行内联；如果在编译本编译单元且调用`sum`函数时，提供的参数n可以知道实际值，则编译器可能会根据`n`的大小来判断时都对`sum`函数进行内联，如果`n`很大，内联展开可能会使最终程序的大小变得很大。

8、虚函数的内联

内联函数是编译阶段的行为，虚函数是执行阶段行为，因此编译器一般会拒绝对虚函数进行内联的请求。虚函数不能被内联是由于编译器在编译时无法知道调用的虚函数到底是哪一个版本，即无法确定虚函数的函数体，但在两种情况下，编译器能够知道虚函数调用的真实版本，因此可以内联。

一是通过对象而不是指向对象的指针或引用对虚函数进行调用，此时编译器在编译器已经知道对象的确切类型，因此会直接调用确切类型的虚函数的实现版本，而不会产生动态绑定行为的代码。

二是虽然通过对象指针或对象引用调用虚函数，但编译器在编译时能够知道指针或引用指向对象的确切类型，如在产生新对象时做的指针赋值或引用初始化与通过指针或引用调用虚函数处于同一编译单元，并且指针没有被改变赋值使其指向到其它不能知道确切类型的对象，此时编译器也不会产生动态绑定的代码，而是直接调用确切类型的虚函数实现版本。

```
inline virtual int x::y(char* a)
{
    ...
}

void func(char* b)
{
    x_base* px = new x();
    x ox;
    px->y(b);
    ox.y(b);
}
```

9、C++内联与C语言宏的区别

C语言宏与C内联的区别如下：

（1）C内联是编译阶段行为，宏是预处理行为，宏的替代展开由预处理器负责，宏对于编译器是不可见的。

（2）预处理器不能对宏的参数进行类型检查，编译器会对内联函数的参数进行类型检查。

（3）宏的参数在宏体内出现两次以上时通常会产生副作用，尤其是当在宏体内对参数进行自增、自减操作时，内联不会。

（4）宏肯定会被展开，`inline`修饰的函数不一定会被内联展开。

C性能优化（四）——C常用数据结构性能分析

本文将根据各种实用操作（遍历、插入、删除、排序、查找）并结合实例对常用数据结构进行性能分析。

一、常用数据结构简介

1、数组

数组是最常用的一种线性表，对于静态的或者预先能确定大小的数据集合，采用数组进行存储是最佳选择。
数组的优点一是查找方便，利用下标即可立即定位到所需的数据节点；二是添加或删除元素时不会产生内存碎片；三是不需要考虑数据节点指针的存储。然而，数组作为一种静态数据结构，存在内存使用率低、可扩展性差的缺点。无论数组中实际有多少元素，编译器总会按照预先设定好的内存容量进行分配。如果超出边界，则需要建立新的数组。

2、链表

链表是另一种常用的线性表，一个链表就是一个由指针连接的数据链。每个数据节点由指针域和数据域构成，指针一般指向链表中的下一个节点，如果节点是链表中的最后一个，则指针为NULL。在双向链表（Double Linked List）中，指针域还包括一个指向上一个数据节点的指针。在跳转链表（Skip Linked List）中，指针域包含指向任意某个关联向的指针。

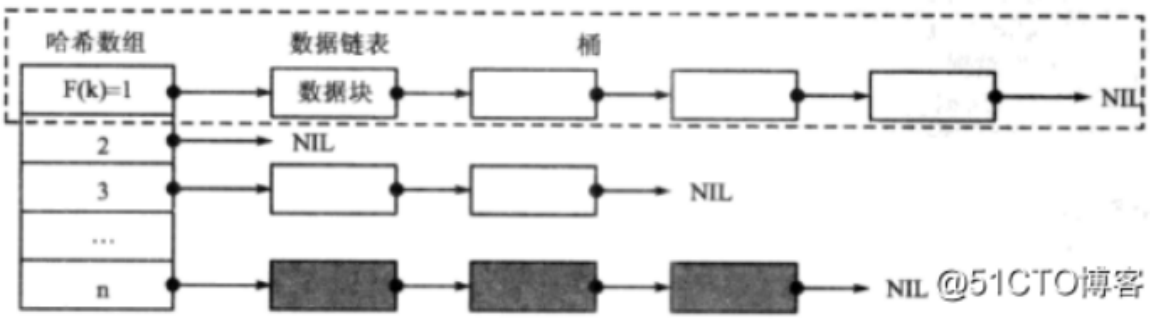
```
template <typename T>
class LinkedNode
{
public:
    LinkedNode(const T& e): pNext(NULL), pPrev(NULL)
    {
        data = e;
    }
    LinkedNode<T>* Next()const
    {
        return pNext;
    }
    LinkedNode<T>* Prev()const
    {
        return pPrev;
    }
private:
    T data;
    LinkedNode<T>* pNext;// 指向下一个数据节点的指针
    LinkedNode<T>* pPrev;// 指向上一个数据节点的指针
    LinkedNode<T>* pConnection;// 指向关联节点的指针
};
```

与预先静态分配好存储空间的数组不同，链表的长度是可变的。只要内存空间足够，程序就能持续为链表插入新的数据项。数组中所有的数据项都被存放在一段连续的存储空间中，链表中的数据项会被随机分配到内存的某个位置。

3、哈希表

数组和链表有各自的优缺点，数组能够方便定位到任何数据项，但扩展性较差；链表则无法提供快捷的数据项定位，但插入和删除任意一个数据项都很简单。当需要处理大规模的数据集合时，通常需要将数组和链表的优点结合。通过结合数组和链表的优点，哈希表能够达到较好的扩展性和较高的访问效率。

虽然每个开发者都可以构建自己的哈希表，但哈希表都有共同的基本结构，如下：



哈希数组中每个项都有指针指向一个小的链表，与某项相关的所有数据节点都会被存储在链表中。当程序需要访问某个数据节点时，不需要遍历整个哈希表，而是先找到数组中的项，然后查询子链表找到目标节点。每个子链表称为一个桶（Bucket），如何定位一个存储目标节点的桶，由数据节点的关键字域Key和哈希函数共同确定，虽然存在多种映射方法，但实现哈希函数最常用的方法还是除法映射。除法函数的形式如下：

`F（k） = k % D`

k是数据节点的关键字，D是预先设计的常量，F（k）是桶的序号（等同于哈希数组中每个项的下标），哈希表实现如下：

```
// 数据节点定义
template <class E, class Key>
class LinkNode
{
public:
    LinkNode(const E& e, const Key& k): pNext(NULL), pPrev(NULL)
    {
        data = e;
        key = k;
    }
    void setNextNode(LinkNode<E, Key>* next)
    {
        pNext = next;
    }
    LinkNode<E, Key>* Next()const
    {
        return pNext;
    }
    void setPrevNode(LinkNode<E, Key>* prev)
    {
        pPrev = prev;
    }
    LinkNode<E, Key>* Prev()const
    {
        return pPrev;
    }

    E& getData()const
```

```
    {
        return data;
    }
    Key& getKey()const
    {
        return key;
    }
private:
    // 指针域
    LinkNode<E, Key>* pNext;
    LinkNode<E, Key>* pPrev;
    // 数据域
    E data;// 数据
    Key key;//关键字
};


// 哈希表定义
template <class E, class Key>
class HashTable
{
private:
    typedef LinkNode<E, Key>* LinkNodePtr;
    LinkNodePtr* hashArray;// 哈希数组
    int size;// 哈希数组大小
public:
    HashTable(int n = 100);
    ~HashTable();
    bool Insert(const E& data);
    bool Delete(const Key& k);
    bool Search(const Key& k, E& ret)const;
private:
    LinkNodePtr searchNode()const;
    // 哈希函数
    int HashFunc(const Key& k)
    {
        return k % size;
    }

};
// 哈希表的构造函数
template <class E, class Key>
HashTable<E, Key>::HashTable(int n)
{
    size = n;
    hashArray = new LinkNodePtr[size];
    memset(hashArray, 0, size * sizeof(LinkNodePtr));
}
// 哈希表的析构函数S
template <class E, class Key>
HashTable<E, Key>::~HashTable()
{
    for(int i = 0; i < size; i++)
    {
        if(hashArray[i] != NULL)
        {
            // 释放每个桶的内存
            LinkNodePtr p = hashArray[i];
```

```
            while(p)
            {
                LinkNodePtr toDel = p;
                p = p->Next();
                delete toDel;
            }
        }
    }
    delete [] hashArray;
}
```

分析代码，哈希函数决定了一个哈希表的效率和性能。

当F（k）=k时，哈希表中的每个桶仅有一个节点，哈希表是一个一维数组，虽然每个数据节点的指针会造成一定的内存空间浪费，但查找效率最高（时间复杂度O（1））。

当F（k）=c时，哈希表所有的节点存放在一个桶中，哈希表退化为链表，同时还加上一个多余的、基本为空的数组，查找一个节点的时间效率为O（n），效率最低。

因此，构建一个理想的哈希表需要尽可能的使用让数据节点分配更均匀的哈希函数，同时哈希表的数据结构也是影响其性能的一个重要因素。例如，桶的数量太少会造成巨大的链表，导致查找效率低下，太多的桶则会导致内存浪费。因此，在设计和实现哈希表前，需要分析数据节点的关键值，根据其分布来决定需要造多大的哈希数组和使用什么样的哈希函数。

哈希表的实现中，数据节点的组织方式多种多样，并不局限于链表，桶可以是一棵树，也可以是一个哈希表。

4、二叉树

二叉树是一种常用数据结构，开发人员通常熟知二叉查找树。在一棵二叉查找树中，所有节点的左子节点的关键值都小于等于本身，而右子节点的关键值大于等于本身。由于平衡二叉查找树与有序数组的折半查找算法原理相同，所以查询效率要远高于链表（O（Log2n）），而链表为O（n）。

由于树中每个节点都要保存两个指向子节点的指针，空间代价要远高于单向链表和数组，并且树中每个节点的内存分配是不连续的，导致内存碎片化。但二叉树在插入、删除以及查找等操作上的良好表现使其成为最常用的数据结构之一。二叉树的链表实现如下：

```
template <class T>
class TreeNode
{
public:
    TreeNode(const TreeNode& e): left(NULL), right(NULL)
    {
        data = e;
    }
    TreeNode<T>* Left()const
    {
        return left;
    }
    TreeNode<T>* Right()const
    {
        return right;
    }
private:
    T data;
    TreeNode<T>* left;
    TreeNode<T>* right;
```

```
};
```

二、数据结构的遍历操作

1、数组的遍历

遍历数组的操作很简单，无论是顺序还是逆序都可以遍历数组，也可以任意位置开始遍历数组。

2、链表的遍历

跟踪指针便能完成链表的遍历：

```
    LinkNode<E>* pNode = pFirst;
    while(pNode)
    {
        pNode = pNode->Next();
        // do something
    }
```

双向链表可以支持顺序和逆序遍历，跳转链表通过过滤某些无用节点可以实现快速遍历。

3、哈希表的遍历

如果预先知道所有节点的Key值，可以通过Key值和哈希函数找到每一个非空的桶，然后遍历桶的链表。否则只能通过遍历哈希数组的方式遍历每个桶。

```
 for(int i = 0; i < size; i++)
    {
        LinkNodePtr pNode = hashArray[i];
        while(pNode) != NULL)
        {
            // do something
            pNode = pNode->Next();
        }
    }
```

4、二叉树的遍历

遍历二叉树由三种方式：前序，中序，后序，三种遍历方式的递归实现如下：

```
// 前序遍历
template <class E>
void PreTraverse(TreeNode<E>* pNode)
{
    if(pNode != NULL)
    {
        // do something
        doSothing(pNode);
        PreTraverse(pNode->Left());
        PreTraverse(pNode->Right());
    }
}

// 中序遍历
template <class E>
```

```
void InTraverse(TreeNode<E>* pNode)
{
    if(pNode != NULL)
    {
        InTraverse(pNode->Left());
        // do something
        doSothing(pNode);
        InTraverse(pNode->Right());
    }
}

// 后序遍历
template <class E>
void PostTraverse(TreeNode<E>* pNode)
{
    if(pNode != NULL)
    {
        PostTraverse(pNode->Left());
        PostTraverse(pNode->Right());
        // do something
        doSothing(pNode);
    }
}
```

使用递归方式对二叉树进行遍历的缺点主要是随着树的深度增加，程序对函数栈空间的使用越来越多，由于栈空间的大小有限，递归方式遍历可能会导致内存耗尽。解决办法主要有两个：

一是使用非递归算法实现前序、中序、后序遍历，即仿照递归算法执行时函数工作栈的变化状况，建立一个栈对当前遍历路径上的节点进行记录，根据栈顶元素是否存在左右节点的不同情况，决定下一步操作（将子节点入栈或当前节点退栈），从而完成二叉树的遍历；

二是使用线索二叉树，即根据遍历规则，在每个叶子节点增加指向后续节点的指针。

三、数据结构的插入操作

1、数组的插入

由于数组中的所有数据节点都保存在连续的内存中，所以插入新的节点需要移动插入位置之后的所有节点以腾出空间，才能正确地将新节点复制到插入位置。如果恰好数组已满，还需要重新建立一个新的容量更大的数组，将原数组的所有节点拷贝到新数组，因此数组的插入操作与其它数据结构相比，时间复杂度更高。

如果向一个未满的数组插入节点，最好的情况是插入到数组的末尾，时间复杂度是O（1），最坏情况是插入到数组头部，需要移动数组的所有节点，时间复杂度是O（n）。

如果向一个满数组插入节点，通常做法是先创建一个更大的数组，然后将原数组的所有节点拷贝到新数组，同时插入新节点，最后删除元数组，时间复杂度为O（n）。在删除元数组之前，两个数组必须并存一段时间，空间开销较大。

2、链表的插入

在链表中插入一个新节点很简单，对于单链表只需要修改插入位置之前节点的pNext指针使其指向本节点，然后将本节点的pNext指针指向下一个节点即可（对于链表头不存在上一个节点，对于链表尾不存在下一个节点）。对于双向链表和跳转链表，需要修改相关节点的指针。链表的插入操作与长度无关，时间复杂度为O（1），当然链表的插入操作通常会伴随链表插入节点位置的定位，需要一定时间。

3、哈希表的插入

在哈希表中插入一个节点需要完成两部操作，定位桶并向链表插入节点。

```
template <class E, class Key>
bool HashTable<E, Key>::Insert(const E& data)
{
    Key k = data;// 提取关键字
    // 创建一个新节点
    LinkNodePtr pNew = new LinkNodePtr(data, k);
    int index = HashFunc(k);//定位桶
    LinkNodePtr p = hashArray[index];
    // 如果是空桶，直接插入
    if(NULL == p)
    {
        hashArray[index] = pNew;
        return true;
    }
    // 在表头插入节点
    hashArray[index] = pNew;
    pNew->SetNextNode(p);
    p->SetPrevNode(pNew);
    return true;
}
```

哈希表插入操作的时间复杂度为O（1），如果桶的链表是有序的，需要花时间定位链表中插入的位置，如果链表长度为M，则时间复杂度为O（M）。

4、二叉树的插入

二叉树的结构直接影响插入操作的效率，对于平衡二叉查找树，插入节点的时间复杂度为O（Log2N）。对于非平衡二叉树，插入节点的时间复杂度比较高，在最坏情况下，非平衡二叉树所有的left节点都为NULL，二叉树退化为链表，插入节点新节点的时间复杂度为O（n）。
当节点数量很多时，对平衡二叉树中进行插入操作的效率要远高于非平衡二叉树。工程开发中，通常避免非平衡二叉树的出现，或是将非平衡二叉树转换为平衡二叉树。简单做法如下：
（1）中序遍历非平衡二叉树，在一个数组中保存所有的节点的指针。
（2）由于数组中所有元素都是有序排列的，可以使用折半查找遍历数组，自上而下逐层构建平衡二叉树。

四、数据结构的删除操作

1、数组的删除

从数组中删除节点，如果需要数组没有空洞，需要在删除节点后将其后所有节点向前移动。最坏情况下（删除首节点），时间复杂度为O（n），最好情况下（删除尾节点），时间复杂度为O（1）。
在某些场合（如动态数组），当删除完成后如果数组中存在大量空闲位置，则需要缩小数组，即创建一个较小的新数组，将原数组中所有节点拷贝到新数组，再将原数组删除。因此，会导致较大的空间与时间开销，应谨慎设置数组的大小，即要尽量避免内存空间的浪费也要减少数组的放大或缩小操作。通常，每当需要删除数组中的某个节点时，并不将其真正删除，而是在节点的位置设计一个标记位bDelete，将其设置为true，同时禁止其它程序使用本节点，待数组中需要删除的节点达到一定阈值时，再统一删除，避免多次移动节点操作，降低时间复杂度。

2、链表的删除

链表中删除节点的操作，直接将被删除节点的上一节点的指针指向被删除节点的下一节点即可，删除操作的时间复杂度是O（1）。

3、哈希表的删除

从哈希表中删除一个节点的操作如下：首先通过哈希函数和链表遍历（桶由链表实现）找到待删除节点，然后删除节点并重新设置前向和后向指针。如果被删除节点是桶的首节点，则将桶的头指针指向后续节点。

```
template <class E, class Key>
bool HashTable<E, Key>::Delete(const Key& k)
{
    // 找到关键值匹配的节点
    LinkNodePtr p = SearchNode(k);

    if(NULL == p)
    {
        return false;
    }
    // 修改前向节点和后向节点的指针
    LinkNodePtr pPrev = p->Prev();
    if(pPrev)
    {
        LinkNodePtr pNext = p->Next();
        if(pNext)
        {
            pNext->SetPrevNode(pPrev);
            pPrev->SetNextNode(pNext);
        }
        else
        {
            // 如果前向节点为NULL，则当前节点p为首节点
            // 修改哈希数组中的节点的指针，使其指向后向节点。
            int index = HashFunc(k);
            hashArray[index] = p->Next();
            if(p->Next() != NULL)
            {
                p->Next()->SetPrevNode(NULL);
            }
        }
    }
    delete p;
    return true;
}
```

4、二叉树的删除

从二叉树删除一个节点需要根据情况讨论：

（1）如果节点是叶子节点，直接删除。

（2）如果删除节点仅有一个子节点，则将子节点替换被删除节点。

（3）如果删除节点的左右子节点都存在，由于每个子节点都可能有自己的子树，需要找到子树中合适的节点，并将其立为新的根节点，并整合两棵子树，重新加入到原二叉树。

五、数据结构的排序操作

1、数组的排序

数组的排序包括冒泡、选择、插入等排序方法。

```
template <typename T>
void Swap(T& a, T& b)
{
  T temp;
  temp = a;
  a = b;
  b = temp;
}
```

冒泡排序实现：

```
/*********************************************
* 排序方式：冒泡排序
* array:序列
* len：序列中元素个数
* min2max：按从小到大进行排序
* ******************************************/
template <typename T>
static void Bubble(T array[], int len, bool min2max = true)
{
    bool exchange = true;
    //遍历所有元素
    for(int i = 0; (i < len) && exchange; i++)
    {
            exchange = false;
            //将尾部元素与前面的每个元素作比较交换
            for(int j = len - 1; j > i; j--)
            {
                    if(min2max?(array[j] < array[j-1]):(array[j] > array[j-1]))
                    {
                            //交换元素位置
                            Swap(array[j], array[j-1]);
                            exchange = true;
                    }
            }
    }
}
```

冒泡排序的时间复杂度为O（n^2），冒泡排序是稳定的排序方法。
选择排序实现：

```
/*****************************************
* 排序方式：选择排序
* array:序列
* len：序列中元素个数
* min2max：按从小到大进行排序
* ***************************************/
template <typename T>
void Select(T array[], int len, bool min2max = true)
```

```
{
 for(int i = 0; i < len; i++)
 {
     int min = i;//从第i个元素开始
     //对待排序的元素进行比较
     for(int j = i + 1; j < len; j++)
     {
         //按排序的方式选择比较方式
         if(min2max?(array[min] > array[j]):(array[min] < array[j]))
         {
             min = j;
         }
     }
     if(min != i)
     {
        //元素交换
        Swap(array[i], array[min]);
     }
 }
}
```

选择排序的时间复杂度为O（n^2），选择排序是不稳定的排序方法。
插入排序实现：

```
/*****************************************
 * 排序方式：选择排序
 * array:序列
 * len：序列中元素个数
 * min2max：按从小到大进行排序
 * ***************************************/
template <typename T>
void Select(T array[], int len, bool min2max = true)
{
  for(int i = 0; i < len; i++)
  {
      int min = i;//从第i个元素开始
      //对待排序的元素进行比较
      for(int j = i + 1; j < len; j++)
      {
          //按排序的方式选择比较方式
          if(min2max?(array[min] > array[j]):(array[min] < array[j]))
          {
              min = j;
          }
      }
      if(min != i)
      {
         //元素交换
         Swap(array[i], array[min]);
      }
  }
}
```

插入排序的时间复杂度为O（n^2），插入排序是稳定的排序方法。

2、链表的排序

虽然链表在插入和删除操作上性能优越，但排序复杂度却很高，尤其是单向链表。由于链表中访问某个节点需要依赖其它节点，不能根据下标直接定位到任意一项，因此节点定位的时间复杂度为O（N），排序效率低下。

工程开发中，可以使用数组链表，当需要排序时构造一个数组，存放链表中每个节点的指针。在排序过程中通过数组定位每个节点，并实现节点的交换。

链表数组为直接访问链表的节点提供了便利，但是使用空间换时间的方法，如果希望得到一个有序链表，最好是在构建链表时将每个节点插入到合适的位置。

3、哈希表的排序

由于采用哈希函数访问每个桶，因此哈希表中对哈希数组排序毫无意义，但具体节点的定位需要通过查询每个桶链表完成（桶由链表实现），将桶的链表排序可以提高节点的查询效率。

4、二叉树的排序

对于二叉查找树，其本身是有序的，中序遍历可以得到二叉查找树有序的节点输出。对于未排序的二叉树，所有节点被随机组织，定位节点的时间复杂度为O（N）。

六、数据结构的查找操作

1、数组的查找

数组的最大优点是可以通过下标任意的访问节点，而不需要借助指针、索引或遍历，时间复杂度为O（1）。对于下标未知的情况查找节点，则只能遍历数组，时间复杂度为O（N）。对于有序数组，最好的查找算法是二分查找法。

```
template <class E>
int BinSearch(E array[], const E& value, int start, int end)
{
    if(end - start < 0)
    {
        return INVALID_INPUT;
    }
    if(value == array[start])
    {
        return start;
    }
    if(value == array[end])
    {
        return end;
    }
    while(end > start + 1)
    {
        int temp  = (end + start) / 2;
        if(value == array[temp])
        {
            return temp;
        }
        if(array[temp] < value)
        {
            start = temp;
        }
```

```
        else
        {
            end = temp;
        }
    }
    return -1;
}
```

折半查找的时间复杂度是O（Log2N），与二叉树查询效率相同。

对于乱序数组，只能通过遍历方法查找节点，工程开发中通常设置一个标识变量保存更新节点的下标，执行查询时从标识变量标记的下标开始遍历数组，执行效率比从头开始要高。

2、链表的查找

对于单向链表，最差情况下需要遍历整个链表才能找到需要的节点，时间复杂度为O（N）。

对于有序链表，可以预先获取某些节点的数据，可以选择与目标数据最接近的一个节点查找，效率取决于已知节点在链表中的分布，对于双向有序链表效率会更高，如果正中节点已知，则查询的时间复杂度为O（N/2）。

对于跳转链表，如果预先能够根据链表中节点之间的关系建立指针关联，查询效率将大大提高。

3、哈希表的查找

哈希表中查询的效率与桶的数据结构有关。桶由链表实现，则查询效率和链表长度有关，时间复杂度为O（M）。查找算法实现如下：

```
template <class E, class Key>
bool HashTable<E, Key>::SearchNode(const Key& k)const
{
    int index = HashFunc(k);
    // 空桶，直接返回
    if(NULL == hashArray[index])
        return NULL;
    // 遍历桶的链表，如果由匹配节点，直接返回。
    LinkNodePtr p = hashArray[index];
    while(p)
    {
        if(k == p->GetKey())
            return p;
        p = p->Next();
    }
}
```

4、二叉树的查找

在二叉树中查找节点与树的形状有关。对于平衡二叉树，查找效率为O（Log2N）；对于完全不平衡的二叉树，查找效率为O（N）；

工程开发中，通常需要构建尽量平衡的二叉树以提高查询效率，但平衡二叉树受插入、删除操作影响很大，插入或删除节点后需要调整二叉树的结构，通常，当二叉树的插入、删除操作很多时，不需要在每次插入、删除操作后都调整平衡度，而是在密集的查询操作前统一调整一次。

七、动态数组的实现及分析

1、动态数组简介

工程开发中，数组是常用数据结构，如果在编译时就知道数组所有的维数，则可以静态定义数组。静态定义数组后，数组在内存中占据的空间大小和位置是固定的，如果定义的是全局数组，编译器将在静态数据区为数组分配空间，如果是局部数组，编译器将在栈上为数组分配空间。但如果预先无法知道数组的维数，程序只有在运行时才知道需要分配多大的数组，此时C编译器可以在堆上为数组动态分配空间。

动态数组的优点如下：

（1）可分配空间较大。栈的大小都有限制，Linux系统可以使用ulimit -s查看，通常为8K。开发者虽然可以设置，但由于需要保证程序运行效率，通常不宜太大。堆空间的通常可供分配内存比较大，达到GB级别。

（2）使用灵活。开发人员可以根据实际需要决定数组的大小和维数。

动态数组的缺点如下：

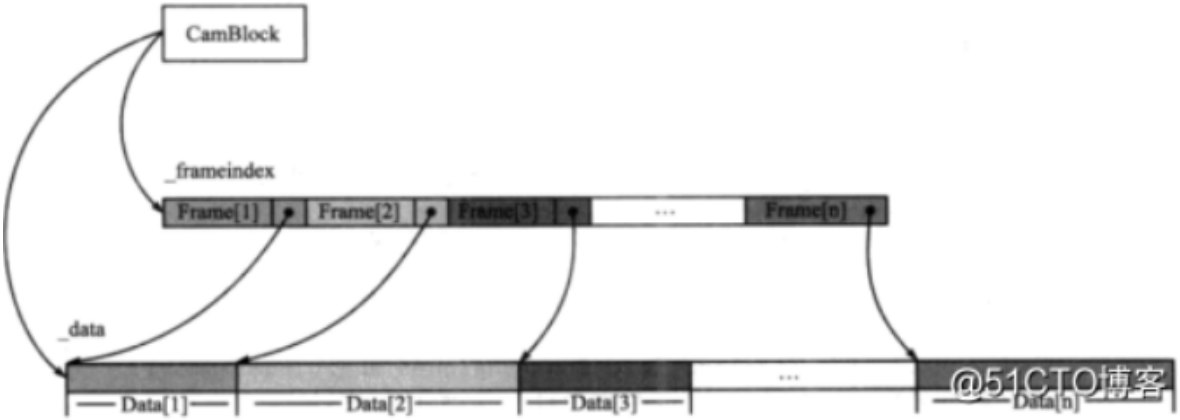
（1）空间分配效率比静态数组低。静态数组一般由栈分配空间，动态数组一般由堆分配空间。栈是机器系统提供的数据结构，计算机会在底层为栈提供支持，即分配专门的寄存器存放栈的地址，压栈和出栈都有专门的机器指令执行，因而栈的效率比较高。堆由C函数库提供，其内存分配机制比栈要复杂得多，为了分配一块内存，库函数会按照一定的算法在堆内存内搜索可用的足够大小的空间，如果发现空间不够，将调用内核方法去增加程序数据段的存储空间，从而程序就有机会分配足够大的内存。因此堆的效率要比栈低。

（2）容易造成内存泄露。动态内存需要开发人员手工分配和释放内存，容易由于开发人员的疏忽造成内存泄露。

2、动态数组实现

在实时视频系统中，视频服务器承担视频数据的缓存和转发工作。一般，服务器为每台摄像机开辟一定大小且独立的缓存区。视频帧被写入此缓存区后，服务器在某一时刻再将其读出，并向客户端转发。视频帧的转发是临时的，由于缓存区大小有限而视频数据源源不断，所以一帧数据在被写入过后，过一段时间并会被新来的视频帧所覆盖。视频帧的缓存时间由缓存区和视频帧的长度决定。
由于视频帧数据量巨大，而一台服务器通常需要支持几十台甚至数百台摄像机，缓存结构的设计是系统的重要部分。一方面，如果预先分配固定数量的内存，运行时不再增加、删除，则服务器只能支持一定数量的摄像机，灵活性小；另一方面，由于视频服务器程序在启动时将占据一大块内存，将导致系统整体性能下降，因此考虑使用动态数组实现视频缓存。

首先，服务器中为每台摄像机分配一个一定大小的缓存块，由类CamBlock实现。每个CamBlock对象中有两个动态数组，分别存放视频数据的*data*和存放视频帧索引信息的frameIndex。每当程序在内存中缓存（读取）一个视频帧时，对应的CamBlock对象将根据视频帧索引表*frameIndex*找到视频帧在data中的存放位置，然后将数据写入或读出。*data*是一个循环队列，一般根据*FIFO*进行读取，即如果有新帧进入队列，程序会在data中最近写入帧的末尾开始复制，如果超出数组长度，则从头覆盖。



```
// 视频帧的数据结构
typedef struct
{
    unsigned short idCamera;// 摄像机ID
    unsigned long length;// 数据长度
    unsigned short width;// 图像宽度
    unsigned short height;// 图像高度
    unsigned char* data; // 图像数据地址
} Frame;

// 单台摄像机的缓存块数据结构
class CamBlock
{
public:
    CamBlock(int id, unsigned long len, unsigned short numFrames):
        _data(NULL), _length(0), _idCamera(-1), _numFrames(0)
    {
        // 确保缓存区大小不超过阈值
        if(len > MAX_LENGTH || numFrames > MAX_FRAMES)
        {
            throw;
        }
        try
        {
            // 为帧索引表分配空间
            _frameIndex = new Frame[numFrames];
            // 为摄像机分配指定大小的内存
            _data = new unsigned char[len];
        }
        catch(...)
        {
            throw;
        }
        memset(this, 0, len);
        _length = len;
        _idCamera = id;
        _numFrames = numFrames;

    }
    ~CamBlcok()
    {
        delete [] _frameIndex;
        delete [] _data;
    }
    // 根据索引表将视频帧存入缓存
    bool SaveFrame(const Frame* frame);
    // 根据索引表定位到某一帧，读取
    bool ReadFrame(Frame* frame);
private:
    Frame* _frameIndex;// 帧索引表
    unsigned char* _data;//存放图像数据的缓存区
    unsigned long _length;// 缓存区大小
    unsigned short _idCamera;// 摄像机ID
    unsigned short _numFrames;//可存放帧的数量
    unsigned long _lastFrameIndex;//最后一帧的位置
};
```

为了管理每台摄像机独立的内存块，快速定位到任意一台摄像机的缓存，甚至任意一帧，需要建立索引表CameraArray来管理所有的CamBlock对象。

```
class CameraArray
{
    typedef CamBlock BlockPtr;
    BlockPtr* cameraBufs;// 摄像机视频缓存
    unsigned short cameraNum;// 当前已经连接的摄像机台数
    unsigned short maxNum;//cameraBufs容量
    unsigned short increaseNum;//cameraBufs的增量
public:
    CameraArray(unsigned short max, unsigned short inc);
    ~CameraArray();
    // 插入一台摄像机
    CamBlock* InsertBlock(unsigned short idCam, unsigned long size, unsigned
short numFrames);
    // 删除一台摄像机
    bool RemoveBlock(unsigned short idCam);
private:
    // 根据摄像机ID返回其在数组的索引
    unsigned short GetPosition(unsigned short idCam);
};

CameraArray::CameraArray(unsigned short max, unsigned short inc):
    cameraBufs(NULL), cameraNum(0), maxNum(0), increaseNum(0)
{
    // 如果参数越界，抛出异常
    if(max > MAX_CAMERAS || inc > MAX_INCREMENTS)
        throw;
    try
    {
        cameraBufs = new BlockPtr[max];
    }
    catch(...)
    {
        throw;
    }
    maxNum = max;
    increaseNum = inc;
}

CameraArray::~CameraArray()
{
    for(int i = 0; i < cameraNum; i++)
    {
        delete cameraBufs[i];
    }
    delete [] cameraBufs;
}
```

通常，会为每个摄像机安排一个整型的ID，在CameraArray中，程序按照ID递增的顺序排列每个摄像机的CamBlock对象以方便查询。当一个新的摄像机接入系统时，程序会根据它的ID在CameraArray中找到一个合适的位置，然后利用相应位置的指针创建一个新的CamBlock对象；当某个摄像机断开连接，程序也会根据它的ID，找到对应的CamBlock缓存块，并将其删除。

```
CamBlock* CameraArray::InsertBlock(unsigned short idCam, unsigned long size,
```

```
                                    unsigned short numFrames)
{
    // 在数组中找到合适的插入位置
    int pos = GetPosition(idCam);
    // 如果已经达到数组边界，需要扩大数组
    if(cameraNum == maxNum)
    {
        // 定义新的数组指针，指定其维数
        BlockPtr* newBufs = NULL;
        try
        {
            BlockPtr* newBufs = new BlockPtr[maxNum + increaseNum];
        }
        catch(...)
        {
            throw;
        }
        // 将原数组内容拷贝到新数组
        memcpy(newBufs, cameraBufs, maxNum * sizeof(BlockPtr));
        // 释放原数组的内存
        delete [] cameraBufs;
        maxNum += increaseNum;
        // 更新数组指针
        cameraBufs = newBufs;
    }
    if(pos != cameraNum)
    {
        // 在数组中插入一个块，需要将其后所有指针位置后移
        memmov(cameraBufs + pos + 1, cameraBufs + pos, (cameraNum - pos) *
sizeof(BlockPtr));
    }
    ++cameraNum;
    CamBlock* newBlock = new CamBlock(idCam, size, numFrames);
    cameraBufs[pos] = newBlock;
    return cameraBufs[pos];
}
```

如果接入系统的摄像机数量超出了最初创建CameraArray的设计容量，则考虑到系统的可扩展性，只要硬件条件允许，需要增加cameraBufs的长度。

```
bool CameraArray::RemoveBlock(unsigned short idCam)
{
    if(cameraNum < 1)
        return false;
    // 在数组中找到要删除的摄像机的缓存区的位置
    int pos = GetPosition(idCam);
    cameraNum--;
    BlockPtr deleteBlock = cameraBufs[pos];
    delete deleteBlock;
    if(pos != cameraNum)
    {
        // 将pos后所有指针位置前移
        memmov(cameraBufs + pos, cameraBufs + pos + 1, (cameraNum - pos) *
sizeof(BlockPtr));
    }
    // 如果数组中有过多空闲的位置，进行释放
    if(maxNum - cameraNum > increaseNum)
```

```
    {
        // 重新计算数组的长度
        unsigned short len = (cameraNum / increaseNum + 1) * increaseNum;
        // 定义新的数组指针
        BlockPtr* newBufs = NULL;
        try
        {
            newBufs = new BlockPtr[len];
        }
        catch(...)
        {
            throw;
        }
        // 将原数组的数据拷贝到新的数组
        memcpy(newBufs, cameraBufs, cameraNum * sizeof(BlockPtr));
        delete cameraBufs;
        cameraBufs = newBufs;
        maxNum = len;
    }
    return true;
}
```

如果删除一台摄像机时，发现数组空间有过多空闲空间，则需要释放相应空闲空间。

C++性能优化（五）——操作系统的内存管理

一、操作系统内存管理简介

长期以来，在计算机系统中，内存都是一种紧缺和宝贵的资源，应用程序必须在载入内存后才能执行。早期，在内存空间不够大时，同时运行的应用程序的数量会受到很大的限制，甚至当某个应用程序在某个运行时所需内存超过物理内存时，应用程序就会无法运行。现代操作系统（Windows、Linux）通过引入虚拟内存进行内存管理，解决了应用程序在内存不足时不能运行的问题。

本质上，虚拟内存就是要让一个程序的代码和数据在没有全部载入内存时即可运行。运行过程中，当执行到尚未载入内存的代码，或者要访问还没有载入到内存的数据时，虚拟内存管理器动态地将相应的代码或数据从硬盘载入到内存中。而且在通常情况下，虚拟内存管理器也会相应地先将内存中某些代码或数据置换到硬盘中，为即将载入的代码或数据腾出空间。

因为内存和硬盘间的数据传输相对于代码执行非常慢，因此虚拟内存管理器在保证工作正确的前提下还必须考虑效率因素，如需要优化置换算法，尽量避免将要被执行的代码或访问的数据刚被置换出内存，而很久没有访问的代码或数据却一直驻留在内存中。虚拟内存管理器还需要将驻留在内存中的各个进程的代码数据维持在一个合理的数量上，并且根据进程性能的表现动态调整，使得程序运行时将涉及的磁盘IO次数降到尽可能低，以提高程序的运行性能。

二、Windows内存管理

1、Windows虚拟内存管理系统简介

Win32虚拟内存管理器为每一个Win32进程提供了进程私有并且基于页的4GB（32bit）大小的线性虚拟地址空间。

进程私有即每个进程只能访问属于自己的内存空间，而无法访问属于其它进程的地址空间，也不用担心自己的地址空间被其它进程看到（父子进程例外，比如调试器利用父子进程关系来访问被被调试进程的地址空间）。进程运行时用到的dll并没有属于自己的地址空间，而是其所属进程的虚拟地址空间，dll的全局数据，以及通过dll函数申请的内存都是从调用其进程的虚拟地址空间开辟的。

基于页是指虚拟地址空间被划分为多个称为页的单元，页的大小由底层处理器决定，x86架构处理器中页的大小为4KB。页是Win32虚拟内存管理器处理的最小单元，相应的物理内存也被划分为多个页。虚拟内存地址空间的申请和释放，以及内存和磁盘的数据传输或置换都是以页为最小单位进行的。

4GB大小意味着进程中的地址取值范围可以从0x00000000到0xFFFFFFFF，Win32将低区的2GB留给进程使用，高区的2GB留给系统使用。

Win32中用来辅助实现虚拟内存的硬盘文件称为调页文件，可以有16个，调页文件用来存放被虚拟内存管理器置换出内存的数据。当调页文件的数据再次被进程访问时，虚拟内存管理器会将其从调页文件中置换进内存，进程可以正确对其访问。用户可以自己配置调页文件，出于空间利用效率和性能的考虑，程序代码不会被修改（包括exe和dll），所以当其所在页被置换出内存时，并不会被写进调页文件中，而是直接抛弃。当再次被需要时，虚拟内存管理器直接从存放程序代码的exe或dll文件中找到并调入内存。另外，对exe和dll文件中包含的只读数据的处理与程序代码处理相同，不会在调页文件中开辟空间存储。

当进程执行某段代码或访问某些数据，而代码或数据还不在内存中时，称为缺页错误。缺页错误的原因很多，最常见的是代码和数据被虚拟内存管理器置换出内存，虚拟内存管理器会在代码被执行或数据被访问前将其调入内存。内存置换对开发人员来说是透明的，大大简化了开发人员的工作。但调页错误涉及磁盘IO，大量的调页错误会大大降低程序的总体性能，因此需要了解缺页错误的主要原因和规避方法。

2、使用虚拟内存

Win32中分配内存分为两个步骤，预留和提交。因此在进程虚拟地址空间中的页有三种状态：自由free、预留reserved和提交committed。

自由表示此页尚未被分配，可以用来满足新的内存分配请求。

预留是指从虚拟地址空间划出一块区域（region，页的整数倍），划出后的内存空间不能用来满足新的内存分配请求，而是用来供要求预留此段内存的代码以后使用。预留时并没有分配物理内存，只是增加了一个描述进程虚拟地址空间使用状态的数据结构（VAD，虚拟地址描述符），用来记录此段内存空间已经被预留。预留操作相对较快，因为没有真正分配物理内存，因此预留的空间不能够直接访问，对预留页的访问会引起内存访问违例。

提交，如果想要得到真正的物理内存，必须对预留的内存进行提交。提交会从调页文件中开辟空间，并修改VAD中的相应项。提交时也并没有立刻从物理内存中分配空间，而是从磁盘的调页文件中开辟空间，作为置换的备份空间。当代码第一次访问提交内存中的数据时，系统发现并没由真正的物理内存，抛出缺页操作。虚拟内存管理器会处理缺页错误，直到此时才会真正分配物理内存，提交也可以在预留的同时进行。提交操作会从磁盘的调页文件中开辟空间，所以比预留操作耗时。

Win32虚拟内存管理中demand-paging策略要求不到真正访问时不会为某虚拟地址分配真正的物理内存。demand-paging策略一是处于性能考虑，将工作分段完成，提高总体性能；二是出于空间效率考虑，不到真正访问时，Win32总是假定认为进程不会访问大多数数据，因而不必要为其开辟存储空间或将其置换进物理内存，以提高存储空间的利用率。

如果某些程序对内存有很大的需求，但并不是立刻需要所有内存，则一次性从物理存储中开辟空间满足潜在的需求，从执行性能和存储空间效率上是一种浪费。由于需求只是潜在的，极有可能分配的内存中很大一部分最后都没有被真正利用。如果在申请时一次性为其分配所有物理存储，会极大降低空间的利用率。

但如果完全不用预留和提交机制，只是随需分配内存来满足每次的请求，则对一个会在不同时间点频繁请求内存的代码来说，因为在其请求内存的不同时间点的间隙极有可能会由其它代码请求内存，会导致在不同时间点频繁请求内存的代码得到的内存因为虚拟地址不连续，无法很好利用空间的locality特性，对其整体进行访问（如遍历）时就会增加缺页错误的数量，从而降低程序性能。

预留和提交在Win32程序中都使用VirtualAlloc函数完成，预留传入MEM_RESERVE参数，提交传入MEM_COMMIT参数。释放虚拟内存时使用VirtualFree函数，根据不同的传入参数，与VirtualAlloc函数对应，可以释放与虚拟地址区域相对应的物理内存，但虚拟地址区域还可以处于预留状态，也可以连同虚拟地址区域一同释放，则虚拟地址区域恢复为自由状态。

线程栈和进程堆的实现利用了预留和提交两步机制，Win32系统中，线程栈使用预留和提交两步机制如下：

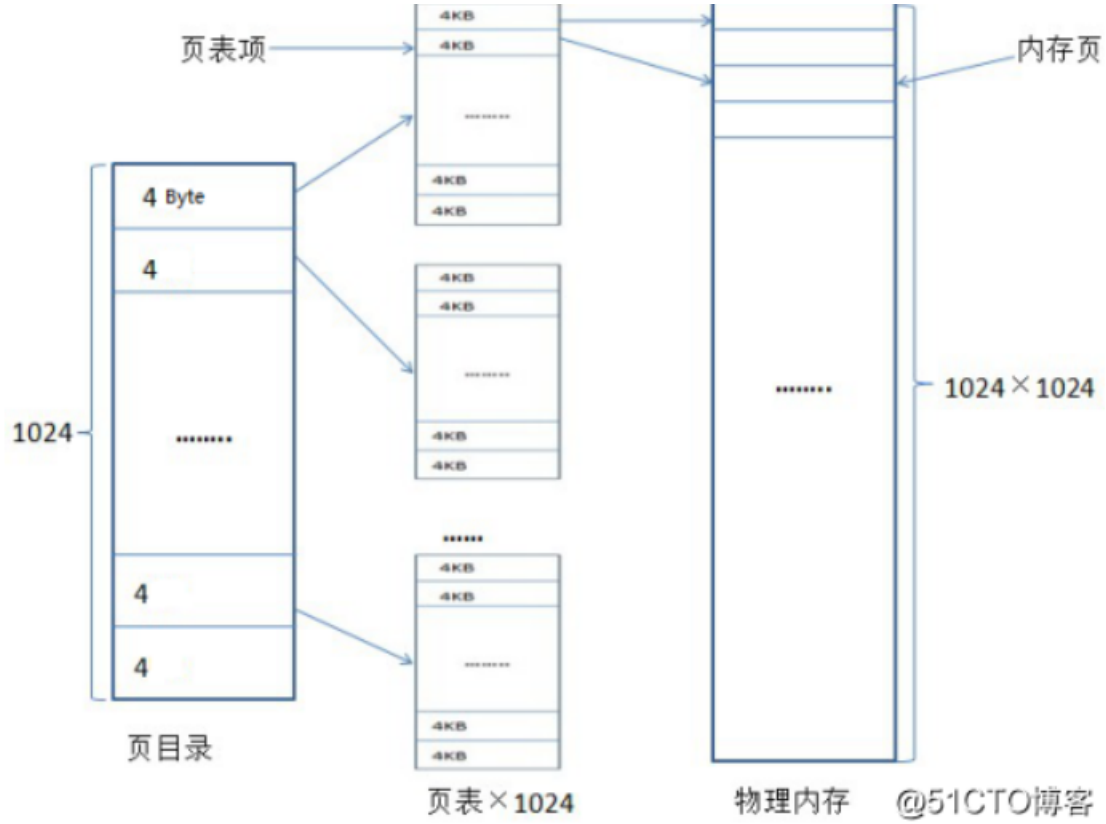
创建线程栈时，只是预留一个虚拟的地址区域，默认为1M（可以在CreateThread或链接时通过链接选项修改），初始时只有前两页是提交的。当线程栈因为函数的嵌套调用需要更多的提交页时，虚拟内存管理器会动态地提交线程的虚拟地址区域中的后续页以满足其需求，直到到达1M的上限。当到达预留区域大小的上限（默认1M）时，虚拟内存管理器不会增加预留区域的大小，而是在提交最后一页时抛出一个栈溢出异常，抛出栈溢出异常时线程栈还有一页空间可以利用，程序仍可正常运行。当程序继续使用栈空间，用完最后一页时，还继续需要存储空间，此时超过上限，会直接导致进程退出。

为了防止线程栈溢出导致整个程序退出，应该尽量控制栈的使用大小。比如减少函数的嵌套层数，减少递归函数的使用，尽量不要在函数中使用较大的局部变量（大的对象可以从堆中开辟空间存放，因为堆会动态扩大，而线程栈的可用内存区域在线程创建时已经固定，在线程的整个生命期都无法扩展）。

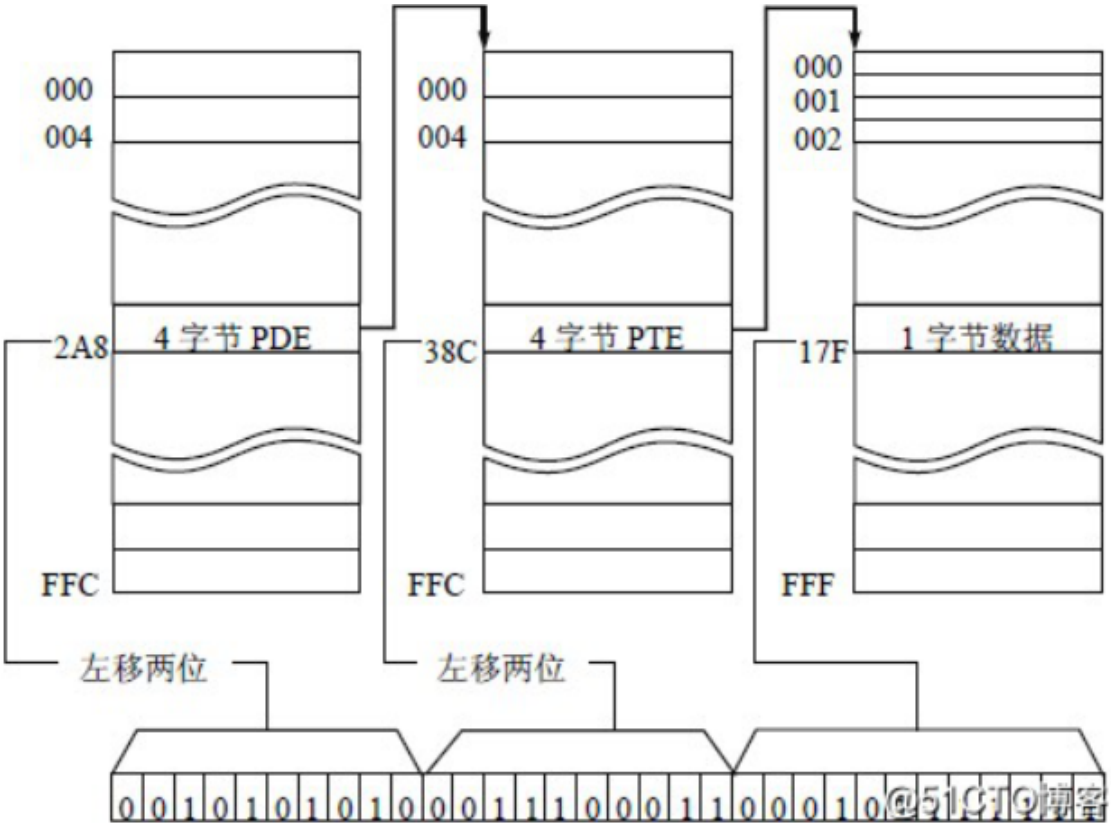
为了防止一个线程栈的溢出导致整个程序退出，可以对可能产生线程栈溢出的线程体函数加异常处理，捕获在提交最后一页时抛出的溢出异常，并做相应处理。

3、访问虚拟内存时的处理流程

对某虚拟内存区域进行了预留并提交后，就可以对虚拟内存区域中数据进行访问。当程序对某段内存访问时处理流程如下：



Win32提供了4GB（32bit）大小的虚拟地址空间，因此每个虚拟地址都是一个32位的整数值，由三部分组成，前10bit为页目录下标，用于定位在页目录的1024项的某一项，根据定位到的某一项的值可以找到第二层页表中的某一个页表；后续10bit为页表下标，用于定位页表的1024项中的某一项，其值可以找到物理内存中的某一个页，此页包含此虚拟地址所代表的数据；后12bit为字节下标，用于定位物理页中特定的字节位置，12位刚好可以定位一个页中的任意位置的字节。



0X2A8E317F的二进制为0010 1010 1000 1110 0011 0001 0111 1111，将其分为三部分，前10bit为00 1010 1010，用于定位页目录中的页目录项，因为页目录项为4个字节，定位前将00 1010 1010左移2bit，得到10 1010 1000（0X2A8），使用0X2A8作为下标找到对应的页目录项，此页目录项指向一个页表。使用后续10bit即00 1110 0011定位此页表中的页表项，00 1110 0011左移2bit后为11 1000 1100（0X38B），使用0X38B作为下标找到此页表中对应的页表项。找到的页表项指向真正的内存。最后使用最后12bit即0001 0111 1111（0X17F），定位页内的数据，即为此指针指向的数据。

Win32总是假定数据已经在物理内存中，并进行地址映射。页表项中有一位用于标记包含此数据的页是否在物理内存页中，当取得页表项时，检测此位，如果在，进行地址映射；如果不在，抛出缺页错误，此时此页表项中包含了此数据是否在调页文件中，如果不在，则访问违例；如果在，此页表项可以查出此数据页是否在调页文件中，以及此数据页在调页文件中的起始位置，然后将此数据页从磁盘中调入物理内存中，再继续进行地址映射过程。为了实现虚拟地址空间各进程私有，每个进程都有自己的页目录项和页表结构，对不同进程而言，页目录中的页目录项，以及页表中的页表项都是不同的，因此相同的指针（虚拟地址)被不同的进程映射到的物理地址也是不同的，即不同进程间传递指针是没有意义的。

5、虚拟内存空间使用状态记录

Win32虚拟内存管理器使用另一个数据结构来记录和维护每个进程的4GB虚拟地址空间的使用及状态信息，即虚拟地址描述符树（VAD，Virtua Address Discriptor）。每一个进程都有自己的VAD集合，VAD集合被组织成一个自平衡二叉树，以提高查找的效率。另外由于只有预留或提交的内存块才会有VAD，自由的内存块没有VAD（即不在VAD树结构中的虚拟地址块就是自由的）。

（1）当程序申请一块新内存时，虚拟内存管理器执行访问VAD，找到两个相邻VAD，只要小的VAD的上限与大的VAD的下限之间的差值满足所申请的内存块的大小需求，即可使用二者之间的虚拟内存。

（2）当第一访问提交的内存时，虚拟内存管理器总是假定要访问的数据所在数据页已经在物理内存中，并进行虚拟地址到物理地址映射。当找到相应的页目录项后发现页目录项并没有指向一个合法的页表，虚拟内存管理器就会查找进程的VAD树，找到包含该地址的VAD，并根据VAD中的信息，比如内存块大小、范围，以及在调页文件中的起始位置，随需生成相应的页表项。然后从刚才发生缺页错误的位置继续进行地址映射。因此，一个虚拟内存页被提交时，除了在调页文件中开辟一个备份页外，不会生成指向它的页表项的页表，也不会填充指向它的页表项，更不会开辟真正的物理内存页，而是直到第一次访问提交页时才会随需地从VAD中取得包含该页的整个区域的信息，生成相应页表，并填充相应页的页表项。

（3）当能够访问预留的内存时，虚拟地址管理器进行虚拟地址到物理地址的映射，找到相应的页目录项后发现页目录项并没有指向一个合法的页表，虚拟地址管理器就会查找进程的VAD树，找到包含该地址的VAD，此时发现此段内存块只是预留的，而没有提交，即没有对应物理内存，直接抛出访问违例，进程退出。

（4）当访问自由的内存时，虚拟地址内存管理器进行虚拟地址到物理地址的映射，找到相应的页目录项后发现页目录项并没有指向一个合法的页表，虚拟地址管理器就会查找进程的VAD树，发现并没有VAD包含此虚拟地址，发现此虚拟地址所在的虚拟内存页是自由状态，直接抛出访问违例，进程退出。

6、进程工作集

因为频繁的调页操作引起的磁盘IO会大大降低程序的运行效率，因此对每一个进程，虚拟内存管理器都会将一定量的内存页驻留在物理内存中，并跟踪其执行的性能指标，并动态调整驻留的内存页数量。Win32中驻留在物理内存中的内存页称为进程的工作集（working set），进程的工作集可以通过任务管理器查看，内存使用列即为工作集大小。

工作集是会动态变化的，进程初始时只有很少的代码页和数据页被调入物理内存。当执行到未被调入内存的代码或访问到尚未调入内存的数据时，相应代码页或数据页会被调入物理内存，工作集也会随之增加。但工作集不能无限增加，系统为每个进程设定了一个最小工作集和最大工作集，当工作集达到最大工作集大小，进程需要再次调入新页到物理内存时，虚拟内存管理器会架构原来工作集中某些内存页先置换出物理内存，然后再将需要调入的新页调入内存。

因为工作集的页驻留在物理内存中，对工作集页的访问不会涉及磁盘IO，因此速度非常快。如果访问的代码或数据不在工作集中，会引发额外的磁盘IO，从而降低程序的执行效率。极端情况下会出现所谓的颠簸或抖动（thrashing），即程序的大部分执行时间都花在调页操作上，而不是执行代码上。

虚拟内存管理器在调页时，不仅仅只是调入需要的页，同时还将其附近的页一起调入内存中，对于开发人员，如果要提高程序的运行效率需要考虑如下：

（1）对代码李硕，尽量编写紧凑代码，最理想情形是工作集不会达到最大阈值，在每次调入新页时，就不需要置换已经载入的内存页，因为根据locality特性，以前执行的代码和访问的数据在后面有很大可能会被再次执行好访问，因此程序执行时，缺页错误会大大降低，即减少磁盘IO。从进程任务管理器也可以查看一个进程从开始时到当前时刻共发生的缺页错误次数。即使不能达到理想情形，紧凑的代码往往意味着接下来执行的代码更大可能就在当前页或相邻页。根据时间locality特性，程序80%的时间花费在20%代码上，如果能将耗时的20%代码尽量紧凑且排在一起，会大大提高程序的整体性能。

（2）对数据来说，尽量将那些会一起访问的数据（如链表）放在一起，当访问数据时，数据在同一页或相邻页，只需要一次调页操作就可以完成。如果数据分散在分散在多个页（多个页不相邻），每次对数据的整体访问都会引发大量的缺页错误，从而降低性能。利用Win32提供的预留和提交两步机制，可以为一同访问的数据预留一大块空间，此时并没有分配实际存储空间，而是在后续执行过程中生成数据时格局需要提交内存，既不浪费存储空间（物理内存和磁盘的调页文件存储空间），又能利用locality特性。

三、Linux内存管理

1、Linux内存管理机制简介

Linux的内存管理主要分为两部分，一部分负责物理内存的申请与释放，物理内存的申请与释放的最小单位为页，在IA32中，页的大小为4KB；另一部分负责处理虚拟内存，虚拟内存的主要操作包括虚拟地址空间与物理地址空间的映射，物理内存页与磁盘页之间的置换等。

2、Linux进程的内存布局

一个32位Linux进程的地址空间为4GB，其中高位1GB，即0XC0000000–0XFFFFFFFF，为内核空间，低位3GB，即0X00000000–0XBFFFFFFF为用户地址空间。用户地址空间进一步被分为程序代码区、数据区（包括初始化数据区DATA和未初始化数据区BSS）、堆和栈。程序代码区占据最低端，往上是初始化数据区DATA和未初始化数据区BSS。代码区存放应用程序的机器代码，运行过程中代码不能修改，因此代码区内存为只读，且大小固定。数据区中存放应用程序的全局数据，静态数据和常量字符串，数据区大小也是固定的。

堆从未初始化数据区开始，向上端动态增长，增长过程中虚拟地址值变大；栈从高位地址开始，向下动态增长，虚拟地址值变小。

堆是应用程序在运行过程中动态申请的内存空间，如通过malloc/new动态生成对象或开辟内存空间时，最终会调用系统调用brk来动态调整数据区的大小。当申请的动态内存区域使用完毕，需要开发者明确使用相应的free/delete对申请的动态内存空间进行释放，free/delete最终也会使用brk系统调用调整数据区的大小。

栈是用来存放函数的传入参数、临时变量以及返回地址等数据，不需要通过malloc/new开辟空间，栈的增长与缩减是因为函数的调用与返回，不需要开发人员操作，没有内存泄漏的危险。

初始化数据区存放的是编译期就能够知道由程序设定初始值的全局变量及静态变量等，其初始值必须保存在最终生成的二进制文件中，并且在程序运行时会原封不动地将此区域映射到进程的初始化数据区。如果一个全局变量或静态变量在源代码中没有被赋初始值，在程序启动后，在第一次被赋值前，其初始值为0，本质上是有初始值的，其初始值为0。但当最终生成二进制文件时，未初始化数据区不会占据对应变量总大小的区域，而是只用一个值进行标识其未初始化数据区的总大小。如一个程序的代码指令有100KB，所有初始化数据总大小为100KB，所有未初始化数据总大小为150KB，则在最终生成的二进制文件中代码区有100KB，接着是100KB的初始化数据区，然后是4字节的大小空间，用于标记未初始化数据区大小，其值为150X1024，用于节省磁盘空间。但在进程虚拟地址空间中，对应未初始化数据区的大小必须是150KB，因为在程序运行时，程序必须真正能够访问到变量中的每一个，即当程序启动时，当检测到二进制文件中未初始化数据区的值为150X1024，则系统会开辟出150KB大小的区域作为进程的未初始化数据区并同时使用0对其进行初始化。

3、Linux物理内存管理

物理内存是用来存放代码指令与供代码指令操作的数据的最终场所，因此物理内存的管理是内存管理系统极其重要的任务。Linux使用页分配器（page allocator）来管理物理内存，页分配器负责分配和回收所有的物理内存页（物理内存的分配与回收的最小单位为4KB大小的页）。

页分配器的核心算法称为兄弟堆算法（buddy-heap algorithm），算法思想是每个物理内存区域都会有一个与之相邻的所谓兄弟区域，当两个区域被回收后，会被合并成为一个区域。如果被合并区域的相邻区域也被回收后，会被进一步合并为更大的区域。当有物理内存请求到来时，页分配器会首先检测是否有大小与之一致的区域。如果有，直接使用找到的匹配区域满足请求；如果没有，则找到更大的一个区域，并继续划分，直到分出的区域能够满足请求。为了配合兄弟堆算法，必须有链表来记录自由的物理内存区域，对于每个相同大小的自由区域，会有一个链表将其连接，每种大小的区域都会有一个链表对其进行管理。自由区域的大小都是2的幂。

当有一个8KB大小的内存请求到来，当前最小可供分配的区域为64KB，此时64KB会被划分为两个32KB，继而将低位的32KB继续划分为两个16KB大小的区域，再将最低位的16KB大小区域划分为两个8KB大小的区域，然后分配高位的8KB区域满足请求。

4、Linux虚拟内存管理

虚拟内存管理器的主要任务是维护应用程序的虚拟地址空间使用信息，如哪些区域已经被使用（映射），是否有磁盘文件作为备份存储。如果有，每个区域对应在磁盘的哪个区域，另外一个重要功能就是调页，如程序访问某些尚未调至物理内存的数据时，虚拟内存管理器负责定位数据，并将其置换进物理内存。如果物理内存此时没有自由页，还需要将物理内存中的某些页先置换出去。

用来维护应用程序的虚拟地址空间使用信息的数据结构是vm_area_struct。每个vm_area_struct结构体都描述了一个进程虚拟地址空间中被分配的区域，当vm_area_struct个数不超过32个时，被连接成为一个链表；当超过32个时，所有的vm_area_struct会被组织为一棵自平衡二叉树，利于提高查询速度。当程序通过某个指针访问某个数据时，系统会查询vm_area_struct树，如果发现指针没有落在任何一个vm_area_struct所表示的区域内，则判定指针所代表的地址没有被分配，即非法的指针访问。

5、虚拟地址映射为物理地址

当通过程序的指针访问某个数据时，因为指针本质是一个虚拟地址值，因此虚拟地址值必须被转化为物理地址值，才能真正访问其所指代的数据。

Linux使用三层映射策略将一个虚拟地址映射为一个物理地址。与Windows相比，多了Middle层，当对于IA32体系，Middle层没有用，因此Linux与Windows相同。

C++性能优化（六）——动态内存管理

一、`new/delete`操作符

C实现通过全局`new/new []`和`delete/delete []`来提高动态内存的访问和管理。`operator new`可以是类的成员函数或全局函数，通常C运行库提供了默认的全局`new/new []`和`delete/delete []`实现，应用程序也可以用自定义实现代替C运行库提供的默认实现，但一个程序最多只能有一个自定义实现。

C标准中`new/new []`和`delete/delete []`声明定义如下：

```
namespace std
{
class bad_alloc;
struct nothrow_t {};
extern const nothrow_t nothrow;
typedef void (*new_handler)();
new_handler set_new_handler (new_handler new_p) throw();
new_handler get_new_handler() noexcept;
}

void* operator new (std::size_t size) throw (std::bad_alloc);
void* operator new (std::size_t size, const std::nothrow_t& nothrow_value)
throw();
void* operator new (std::size_t size, void* ptr) throw();

void* operator new[] (std::size_t size) throw (std::bad_alloc);
void* operator new[] (std::size_t size, const std::nothrow_t& nothrow_value)
throw();
void* operator new[] (std::size_t size, void* ptr) throw();

void operator delete (void* ptr) throw();
void operator delete (void* ptr, const std::nothrow_t& nothrow_constant)
throw();
```

```
void operator delete (void* ptr, void* voidptr2) throw();

void operator delete[] (void* ptr) throw();
void operator delete[] (void* ptr, const std::nothrow_t& nothrow_constant)
throw();
void operator delete[] (void* ptr, void* voidptr2) throw();
```

C标准没有规定是否要对获得的内存进行初始化，因此内存中初始值取决于C编译器实现和开发人员对内存的显式赋值。`operator new`定义有三种方式：

```
void* operator new (std::size_t size) throw (std::bad_alloc);
void* operator new (std::size_t size, const std::nothrow_t& nothrow_value)
throw();
void* operator new (std::size_t size, void* ptr) throw();
```

第一种方式在分配内存失败时会抛出`bad_alloc`异常，是C标准规定。

第二种方式在分配内存失败时不抛出异常，返回`NULL`，用于兼容C早期代码。

第三种方式返回指定位置的内存作为分配的内存。

C标准对内存分配失败有明确规定，系统会调用当前安装的`new_handler`函数，`new_handler`错误处理函数通过`set_new_handler`函数安装到系统。C标准规定`new_handler`要执行下列三种操作的一种：

（1）使`new`有更多内存可用，然后返回。

（2）抛出`bad_alloc`异常或`bad_alloc`派生异常。

（3）调用`abort`或`exit`退出。

二、自定义全局`new/delete`操作符

当应用程序需要使用统一机制来控制数据的内存分配情况，并且不想使用系统提供的内存管理机制时，可以通过重写自己的全局`operator new/delete`来实现。通常，对于内存要求较高的应用程序或调试内存分配情况（生成`log`用于调试和排错）可能会需要重写全局`operator new/delete`。

```
#include <stdio.h>
#include <new>
#include <stdlib.h>

using namespace std;

char* gPool = NULL;

const unsigned int N = 1024* 1024*1024;

void def_new_handler()
{
    if(gPool != NULL)
    {
        printf("try to malloc more memory...\n");
        delete [] gPool;
        gPool = NULL;
        return ;
    }
    else
```

```
    {
        printf("new_handler no memory\n");
        throw std::bad_alloc();
    }

}

void* operator new(std::size_t n)
{
    void* pReturn = NULL;
    printf("global operator new, size = %u\n", n);
    if(n == 0)
    {
        n = 1;
    }
    while(true){
        pReturn = malloc(n);
        if(pReturn != NULL){
            printf("malloc %u Bytes memory at %x\n", n, pReturn);
            return pReturn;
        }
        new_handler global_handler = set_new_handler(0);
        set_new_handler(global_handler);
        if(global_handler != NULL){
            printf("call def_new_handler\n");
            (*global_handler)();
        }
        else
        {
            printf("no memory, no new_handler\n");
            throw std::bad_alloc();
        }

    }

}

void operator delete(void* p)
{
    printf("operator delete at %x\n", p);
    return free(p);
}

int main()
{
    set_new_handler(def_new_handler);
    gPool = new char[3 * N];
    if(gPool != NULL)
    {
        printf("preserve memory at %x\n", gPool);
    }
    char* p[10] = {0};
    for (size_t i = 0; i < 10; i++)
    {
        p[i] = new char[N];
        printf("%d GB p = %x\n", i + 1, p[i]);
    }
```

```
    return 0;
}
```

三、自定义类new/delete操作符

自定义全局operator new/delete时，程序中所有的内存分配和释放都使用统一方式，但某些使用场景中则要求不同的类对象使用不同的内存分配方式。可以通过重载类成员函数operator new/delete实现自定义类对象内存分配，其它程序则使用系统默认operator new/delete分配和释放内存。

自定义类的operator new/delete需要声明为static函数。

```
#include <stdio.h>
#include <new>

using namespace std;

class Parent
{
public:
    static void* operator new(size_t n)
    {
        printf("Parent::operator new, size = %u\n", n);
        return ::operator new(n);
    }
    static void operator delete(void* p)
    {
        printf("Parent::operator delete, p = %x\n", p);
        return ::operator delete(p);
    }
public:
    int m_value;
};

class Child : public Parent
{
public:
    int m_childValue;
};

int main()
{
    Parent* p = new Parent();
    Child* c = new Child();

    delete p;
    delete c;
    return 0;
}
```

派生类默认继承使用基类的自定义operator new/delete方法，如果基类自定义operator new/delete方法只针对基类内存分配效率优化，不希望派生类使用基类自定义的operator new/delete方法，则需要在基类的operator new/delete方法中进行处理。

也可以在派生类中重写operator new/delete方法，直接调用全局operator new/delete方法。

```
#include <stdio.h>
```

```
#include <new>

using namespace std;

class Parent
{
public:
    static void* operator new(size_t n)
    {
        printf("Parent::operator new, size = %u\n", n);
        return ::operator new(n);
    }
    static void operator delete(void* p)
    {
        printf("Parent::operator delete, p = %x\n", p);
        return ::operator delete(p);
    }
public:
    int m_value;
};

class Child : public Parent
{
public:
        static void* operator new(size_t n)
    {
        return ::operator new(n);
    }
    static void operator delete(void* p)
    {
        return ::operator delete(p);
    }
    int m_childValue;
};

int main()
{
    Parent* p = new Parent();
    Child* c = new Child();

    delete p;
    delete c;
    return 0;
}
```

四、避免内存泄露

1、内存泄露简介

C语言没有自动垃圾回收机制，但提供了强大灵活的机制避免内存泄露问题。C语言中通过`new`分配的内存或创建的对象必须使用`delete`释放，因此可以将分配分配和释放封装到类中，在构造函数中分配内存，在析构函数释放内存。

```
#include <stdio.h>
#include <string.h>
```

```
using namespace std;

class SimpleClass
{
public:
    SimpleClass(size_t n = 1)
    {
        m_buf = new char[n];
        m_nSize = n;
    }
    char* buffer()
    {
        return m_buf;
    }
    ~SimpleClass()
    {
        printf("delete at %x, size = %u\n", m_buf, m_nSize);
        delete [] m_buf;
    }
private:
    char* m_buf;
    size_t m_nSize;
};

void func()
{
    SimpleClass obj(10);
    char* buf = obj.buffer();
    strcpy(buf, "hello");
    printf("buffer: %s\n", buf);
}


int main()
{
    func();
    return 0;
}
```

`SimpleClass`类声明对象时分配所需内存，分配的内存空间在`SimpleClass`类对象作用域退出时将自动释放，不需要显示`delete`释放。

2、拷贝构造

`SimpleClass`类没有实现拷贝构造函数，C++编译器会构造一个默认拷贝构造函数，执行位拷贝操作（浅拷贝），将`SimpleClass`对象内容逐个字节拷贝。因此，拷贝后不同对象的`m_buf`指向同一块内存空间，对象析构时会造成`m_buf`多次释放。

`SimpleClass`类可以在将拷贝构造函数声明为私有函数，禁止拷贝构造，但当`SimpleClass`类使用者试图进行赋值或作为参数传递给函数时会出现编译错误。

```
#include <stdio.h>
#include <string.h>

using namespace std;
```

```
class SimpleClass
{
public:
    SimpleClass(size_t n = 1)
    {
        m_buf = new char[n];
        m_nSize = n;
    }

    char* buffer()
    {
        return m_buf;
    }
    ~SimpleClass()
    {
        printf("delete at %x, size = %u\n", m_buf, m_nSize);
        delete [] m_buf;
    }
private:
    SimpleClass(const SimpleClass& other);
private:
    char* m_buf;
    size_t m_nSize;
};

void func()
{
    SimpleClass obj(10);
    SimpleClass objb = obj;
    char* buf = obj.buffer();
    strcpy(buf, "hello");
    printf("buffer: %s\n", buf);
}


int main()
{
    func();
    return 0;
}
```

3、引用计数

引用计数是对要使用的内存维护一个计数器，记录当前有多少指针指向分配的内存空间。当有指针指向内存空间时，计数器加1；当指向内存空间的指针被销毁时，计数器减1；当内存空间的计数器为0时，没有指针指向内存空间，内存空间可以被释放。

引用计数不仅要考虑类对象初始化时多个对象指向内存空间的引用计数，还需要考虑不同对象之间赋值时指针指向内存空间的转移。

```
#include <stdio.h>
#include <string.h>

using namespace std;

class SimpleClass
{
```

```
public:
    SimpleClass(size_t n = 1)
    {
        m_buf = new char[n];
        m_nSize = n;
        m_count = new int;
        (*m_count) = 1;
        printf("count is %u\n", *m_count);
    }
    SimpleClass(const SimpleClass& other)
    {
        m_buf = other.m_buf;
        m_nSize = other.m_nSize;
        m_count = other.m_count;
        (*m_count)++;
        printf("count is %u\n", *m_count);
    }

    SimpleClass& operator=(const SimpleClass& other)
    {
        if(m_buf == other.m_buf)
        {
            return *this;
        }
        (*m_count)--;
        if((*m_count) == 0)
        {
            printf("delete at %x, size = %u\n", m_buf, m_nSize);
            delete [] m_buf;
            delete m_count;
        }
        m_buf = other.m_buf;
        m_nSize = other.m_nSize;
        (*m_count)++;
    }

    char* buffer()
    {
        return m_buf;
    }
    ~SimpleClass()
    {
        (*m_count)--;
        printf("count is %u\n", *m_count);
        if(*m_count == 0)
        {
            printf("delete at %x, size = %u\n", m_buf, m_nSize);
            delete [] m_buf;
            delete m_count;
        }
    }
private:
    char* m_buf;
    size_t m_nSize;
    int* m_count;
};

void func()
```

```
{
    SimpleClass obj(10);
    char* buf = obj.buffer();
    strcpy(buf, "hello");
    printf("buffer: %s\n", buf);
    SimpleClass obja = obj;
    SimpleClass objb(8);
    objb = obj;
}


int main()
{
    func();
    return 0;
}
```

4、深拷贝

等位拷贝只会逐字节拷贝，造成多个对象的指针指向同一块内存空间。因此，在拷贝构造函数和赋值操作符函数内需要对内存空间资源进行深度拷贝，即深拷贝。

```
#include <stdio.h>
#include <string.h>

using namespace std;

class SimpleClass
{
public:
    SimpleClass(size_t n = 1)
    {
        m_buf = new char[n];
        m_nSize = n;
        printf("SimpleClass Constructor, size = %u\n", m_nSize);
    }
    SimpleClass(const SimpleClass& other)
    {
        m_nSize = other.m_nSize;
        m_buf = new char[m_nSize];
        strncpy(m_buf, other.m_buf,  m_nSize - 1);
        printf("SimpleClass Copy Constructor, size = %u\n", m_nSize);
    }

    SimpleClass& operator=(const SimpleClass& other)
    {
        if(m_buf == other.m_buf)
        {
            return *this;
        }
        delete [] m_buf;
        m_nSize = other.m_nSize;
        m_buf = new char[m_nSize];
        strncpy(m_buf, other.m_buf,  m_nSize - 1);
        printf("SimpleClass assign operator=, size = %u\n", m_nSize);
    }
```

```
    char* buffer()
    {
        return m_buf;
    }
    ~SimpleClass()
    {
        printf("delete at %x, size = %u\n", m_buf, m_nSize);
        delete [] m_buf;
    }
private:
    char* m_buf;
    size_t m_nSize;
};

void func()
{
    SimpleClass obj(10);
    char* buf = obj.buffer();
    strcpy(buf, "hello");
    printf("buffer: %s\n", buf);
    SimpleClass obja = obj;
    SimpleClass objb(8);
    objb = obj;
}


int main()
{
    func();
    return 0;
}
```

五、智能指针

1、智能指针简介

智能指针是存储指向动态内存、对象指针的类，工作机制与C语言指针相同，但会在适当时机自动删除指向的内存或对象。

2、`auto_ptr`

`auto_ptr`是C++标准库提供的类模板，`auto_ptr`对象通过初始化指向由`new`创建的动态内存。当`auto_ptr`对象生命周期结束时，其析构函数会将`auto_ptr`对象拥有的动态内存自动释放。即使发生异常，通过异常的栈展开过程也能将动态内存释放。

`auto_ptr`使用注意事项如下：

（1）`auto_ptr`不能共享所有权。

（2）`auto_ptr`不能指向数组。

（3）`auto_ptr`不能作为容器的成员。

（4）不能通过赋值操作来初始化`auto_ptr`。

```
#include <memory>
#include <string>
#include <stdio.h>

using namespace std;

class Test
{
public:
    Test(const char* c)
    {
        m_buffer = c;
        printf("Constructor, %s\n", m_buffer.c_str());
    }
    ~Test()
    {
        printf("DeConstructor, %s\n", m_buffer.c_str());
    }
    void print()
    {
        printf("%s\n", m_buffer.c_str());
    }
private:
    string m_buffer;
};

int main()
{
    auto_ptr<string> p(new string("HelloWorld."));
    printf("%s\n", p->c_str());
    auto_ptr<Test> pTest(new Test("HelloTest."));
    // auto_ptr<Test> pTest = new Test("HelloTest.");// error
    pTest->print();
}
```

3、boost::scoped_ptr

boost::scoped_ptr是一个简单的智能指针，能够保证在离开作用域后对象被自动释放。

boost::scoped_ptr的实现利用了一个栈上的对象去管理一个堆上的对象，从而使得堆上的对象随着栈上的对象销毁时自动删除，scoped_ptr不能拷贝，因此不能转换其所有权的。

scoped_ptr使用注意事项如下：

（1）不能转换所有权

boost::scoped_ptr所管理对象生命周期仅局限于一个作用域，因此不能转换所有权给其它对象。

（2）不能共享所有权

scoped_ptr将拷贝构造函数和赋值运算符定义为私有的，因此scoped_ptr不能用于STL容器中，因为容器中的push_back操作需要调用scoped_ptr的赋值运算符重载函数，会导致编译失败。

（3）不能用于管理数组对象

scoped_ptr是通过delete来删除所管理对象的，而数组对象必须通过deletep[]来删除，因此boost::scoped_ptr不能管理数组对象，如果要管理数组对象需要使用boost::scoped_array类。

（4）多个`boost::scoped_ptr`对象不能管理同一个对象。

`scoped_ptr`对象在离开自己作用域时会调用了自身析构函数，在析构函数内部会调用释放对象，当多个`scoped_ptr`管理同一个对象时，在离开作用域后，会多次调用`delete`以释放所管理的对象，从而造成程序运行出错。

（5）`boost::scoped_ptr`对象可以提前释放所管理对象。

`boost::scoped_ptr`通过新生成的无名临时变量，将新地址与旧地址交换，在最后脱离作用域范围，对象消亡，调用析构函数，释放原先空间，达到不内存泄漏，并且对新空间进行管理。

```
#include <stdio.h>
#include <boost/scoped_ptr.hpp>
#include <vector>

using namespace std;
using namespace boost;

class Test
{
public:
    Test(const char* c)
    {
        m_buffer = c;
        printf("Creating Test ..., %s\n", m_buffer.c_str());
    }
    void print()
    {
        printf("print Test ..., %s\n", m_buffer.c_str());
    }
    ~Test()
    {
        printf("Destroying Test ..., %s\n", m_buffer.c_str());
    }
private:
    string m_buffer;
};

int main()
{
    printf("=====Main Begin=====\n");
    {
        scoped_ptr<Test> pTest(new Test("HelloTest"));
        // scoped_ptr<Test> pTest1(pTest);// 拷贝构造
        // scoped_ptr<Test> pTest2 = pTest;// 赋值操作符
        pTest->print();
        Test* test = new Test("HelloTest1");
        pTest.reset(test);
        pTest->print();
        pTest.reset();

        vector< scoped_ptr<Test> > vec;
        // vec.push_back(pTest); // error
    }
    printf("===== Main End =====\n");

    return 0;
```

```
}
```

4、boost::shared_ptr

boost::shared_ptr是可以共享所有权的智能指针。如果有多个shared_ptr共同管理同一个对象时，只有全部shared_ptr与对象脱离关系后，被管理的对象才会被释放。

boost::shared_ptr对所管理的对象进行了引用计数，当新增一个boost::shared_ptr对对象进行管理时，就将对象的引用计数加1；减少一个boost::shared_ptr对对象进行管理时，就将对象的引用计数减1，如果对象的引用计数为0时，delete释放其所占的内存。
boost::shared_ptr使用注意事项：

（1）避免对shared_ptr所管理的对象的直接内存管理操作，以免造成对象的多次释放

（2）shared_ptr不能对循环引用的对象内存自动管理。

（3）不要构造一个临时的shared_ptr作为函数的参数

```
#include <stdio.h>
#include <boost/shared_ptr.hpp>
#include <vector>

using namespace std;
using namespace boost;

class Test
{
public:
    Test(const char* c)
    {
        m_buffer = c;
        printf("Creating Test ..., %s\n", m_buffer.c_str());
    }
    void print()
    {
        printf("print Test ..., %s\n", m_buffer.c_str());
    }
    ~Test()
    {
        printf("Destroying Test ..., %s\n", m_buffer.c_str());
    }
private:
    string m_buffer;
};

int main()
{
    shared_ptr<Test> pTest(new Test("HelloTest"));
    pTest->print();
    shared_ptr<Test> pTest2 = pTest;
    pTest2->print();

    Test* test = new Test("HelloTest1");
    pTest.reset(test);
    pTest->print();
    pTest.reset();
```

```
    return 0;
}
```

5、boost::weak_ptr

weak_ptr是为了配合shared_ptr而引入的智能指针，用于弥补shared_ptr不足，解决循环引用的问题
boost::weak_ptr是boost::shared_ptr的观察者对象，当被观察的boost::shared_ptr失效后，相应的boost::weak_ptr也随之失效。boost::weak_ptr只对boost::shared_ptr进行引用，而不改变其引用计数，weak_ptr的构造不会引起指针引用计数的增加，weak_ptr析构也不会引起指针引用计数的减少。

```
#include <stdio.h>
#include <boost/shared_ptr.hpp>
#include <boost/weak_ptr.hpp>

int main(int argc, char *argv[])
{
    boost::shared_ptr<int> sp(new int(10));
    boost::weak_ptr<int> wp(sp);

    // 测试计数器是否是0
    if (!wp.expired())
    {
        // 提升wp到shared_ptr---sp2
        boost::shared_ptr<int> sp2 = wp.lock();
        *sp2 = 100;
    }

    // shared_ptr置空，weak_ptr失效
    sp.reset();
    printf("expired= %d\n", wp.expired());
    wp.lock();
}
```

C++性能优化（七）——内存池技术

一、内存池简介

1、C++内存池简介

内存池(Memory Pool)是一种内存分配方式，是在真正使用内存前，先申请分配一定数量的、大小相等(一般情况下)的内存块留作备用。当有新的内存需求时，就从内存池中分出一部分内存块，若内存块不够再继续申请新的内存。

通用内存分配和释放的缺点如下：

（1）使用malloc/new申请分配堆内存时系统需要根据最先匹配、最优匹配或其它算法在内存空闲块表中查找一块空闲内存；使用free/delete释放堆内存时，系统可能需要合并空闲内存块，因此会产生额外开销。

（2）频繁使用时会产生大量内存碎片，从而降低程序运行效率。

（3）造成内存泄漏。

内存池（Memory Pool)是代替直接调用`malloc/free、new/delete`进行内存管理的常用方法，当申请内存空间时，会从内存池中查找合适的内存块，而不是直接向操作系统申请。

内存池技术的优点如下：

（1）堆内存碎片很少。

（2）内存申请/释放比`malloc/new`方式快。

（3）检查任何一个指针是否在内存池中。

（4）写一个堆转储(`Heap-Dump`)到硬盘。

（5）内存泄漏检测(`memory-leak detection`)，当没有释放分配的内存时，内存池(`Memory Pool`)会抛出一个断言`(assertion)`。

内存池可以分为不定长内存池和定长内存池两类。不定长内存池的典型实现包括`Apache Portable Runtime`中的`apr_pool`和`GNU lib C`中的`obstack`，而定长内存池的实现则有`boost_pool`等。对于不定长内存池，不需要为不同的数据类型创建不同的内存池，其缺点是无法将分配出的内存回收到池内；对于定长内存池，在使用完毕后，可以将内存归还到内存池中，但需要为不同类型的数据结构创建不同的内存池，需要内存的时候要从相应的内存池中申请内存。

2、常见C++内存池实现方案

（1）固定大小缓冲池

固定大小缓冲池适用于频繁分配和释放固定大小对象的情况。

（2）dlmalloc

`dlmalloc`是一个内存分配器，由Doug Lea从1987年开始编写，目前最新版本为2.8.3，由于其高效率等特点被广泛使用和研究。

ftp://g.oswego.edu/pub/misc/malloc.c

（3） SGI STL内存分配器

`SGI STL allocator`是目前设计最优秀的 C++ 内存分配器之一，其内部`free_list[16]`数组负责管理从 8 bytes到128 bytes不同大小的内存块（ chunk ），每一个内存块都由连续的固定大小（ fixed size block ）的很多 chunk 组成，并用指针链表连接。

（4）Loki小对象分配器

Loki 分配器使用`vector`管理数组，可以指定 fixed size block 的大小。free blocks分布在一个连续的大内存块中，free chunks 可以根据使用情况自动增长和减少合适的数目，避免内存分配得过多或者过少。

（5）Boost object_pool

`Boost object_pool`可以根据用户具体应用类的大小来分配内存块，通过维护一个free nodes的链表来管理。可以自动增加nodes块，初始32个nodes，每次增加都以两倍数向system heap要内存块。object_pool 管理的内存块需要在其对象销毁的时候才返还给 system heap 。

（6）ACE_Cached_Allocator 和 ACE_Free_List

`ACE`框架中包含一个可以维护固定大小的内存块的分配器，通过在`ACE_Cached_Allocator`中定义`Free_list`链表来管理一个连续的大内存块，内存块中包含多个固定大小的未使用内存区块（ free chunk），同时使用`ACE_unbounded_Set`维护已使用的`chuncks`。

（7）TCMalloc

Google开源项目`gperftools`提供了内存池实现方案。 https://code.google.com/p/gperftools/
`TCMalloc`替换了系统的`malloc`，更加底层优化，性能更好。

3、STL内存分配器

分配器(`allocator`)是C ++标准库的一个组件, 主要用来处理所有给定容器(`vector，list，map`等)内存的分配和释放。C ++标准库提供了默认使用的通用分配器`std::allocator`，但开发者可以自定义分配器。

`GNU STL`除了提供默认分配器，还提供了`__pool_alloc`、`__mt_alloc`、`array_allocator`、`malloc_allocator`内存分配器。
`__pool_alloc`：SGI内存池分配器
`__mt_alloc`： 多线程内存池分配器
`array_allocator`： 全局内存分配，只分配不释放，交给系统来释放
`malloc_allocator`：堆`std::malloc`和`std::free`进行的封装

二、STL allocator

1、STL allocator简介

`new`会分配内存并执行对象构造函数，`delete`会执行对象析构函数并释放内存。如果将内存分配和对象构造分离，可以先分配大块内存，只在需要时才真正执行对象构造函数。

`STL`在头文件`memory`中提供了一个`allocator`类，允许将分配和对象构造分离，提供更好的性能和更灵活的内存管理能力。为了定义一个`allocator`对象，必须指明`allocator`可以分配的对象类型。当`allocator`分配内存时，会根据给定的对象类型来确定恰当的内存大小和对齐位置。

2、STL allocator接口

`STL allocator`的标准接口如下：

```
typedef size_t     size_type;
typedef ptrdiff_t  difference_type;
typedef _Tp*       pointer;
typedef const _Tp* const_pointer;
typedef _Tp&       reference;
typedef const _Tp& const_reference;
typedef _Tp        value_type;

void construct(pointer __p, const _Tp& __val) { ::new((void *)__p)
value_type(__val); }

void destroy(pointer __p) { __p->~_Tp(); }

size_type max_size() const _GLIBCXX_USE_NOEXCEPT { return size_t(-1) /
sizeof(_Tp); }

address(const_reference __x) const _GLIBCXX_NOEXCEPT

deallocate(pointer, size_type);

allocate(size_type __n, const void* = 0);

template<typename _Tp1>
struct rebind { typedef allocator<_Tp1> other; };
```

根据C++标准规范，`STL`中分配器的对外接口、成员变量都一样，只是接口内部实现有区别。`allocator`实现在模板类`new_allocator`中：

```
namespace __gnu_cxx _GLIBCXX_VISIBILITY(default)
{
_GLIBCXX_BEGIN_NAMESPACE_VERSION
  template<typename _Tp>
    class new_allocator
    {
    public:
      typedef _Tp*       pointer;
      typedef const _Tp* const_pointer;

      // NB: __n is permitted to be 0.  The C++ standard says nothing
      // about what the return value is when __n == 0.
      pointer
      allocate(size_type __n, const void* = 0)
      {
    if (__n > this->max_size())
      std::__throw_bad_alloc();

    return static_cast<_Tp*>(::operator new(__n * sizeof(_Tp)));
      }

      // __p is not permitted to be a null pointer.
      void
      deallocate(pointer __p, size_type)
      { ::operator delete(__p); }
    }
_GLIBCXX_END_NAMESPACE_VERSION
}
```

`STL`中容器默认分配器为`std::allocator<_Tp>`，内存分配和释放的接口`allocate`和`deallocate`内部实现只是将`::operator new`和`::operator delete`进行封装，没用做特殊处理。

3、`STL allocator`实例

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

class Test
{
public:
    Test()
    {
        cout << "Test Constructor" << endl;
    }
    ~Test()
    {
        cout << "Test DeConstructor" << endl;
```

```
    }

    Test(const Test &t)
    {
        cout << "Copy Constructor" << endl;
    }
};

int main(int argc, const char *argv[])
{
    allocator<Test> alloc;
    //申请三个单位的Test内存，未经初始化
    Test *pt = alloc.allocate(3);
    {
        // 构造对象，使用默认值
        alloc.construct(pt, Test());
        // 调用拷贝构造函数
        alloc.construct(pt + 1, Test());
        alloc.construct(pt + 2, Test());
    }
    alloc.destroy(pt);
    alloc.destroy(pt + 1);
    alloc.destroy(pt + 2);

    alloc.deallocate(pt, 3);
    return 0;
}

// output:
//Test Constructor
//Copy Constructor
//Test DeConstructor
//Test Constructor
//Copy Constructor
//Test DeConstructor
//Test Constructor
//Copy Constructor
//Test DeConstructor
//Test DeConstructor
//Test DeConstructor
//Test DeConstructor
```

4、自定义`allocator`

实现`Allocator`只需要实现`allocate`和`deallocate`，来实现自己的内存分配策略。

```
#ifndef ALLOCATOR_HPP
#define ALLOCATOR_HPP

#include <stddef.h>
#include <limits>

template <typename T>
class Allocator
{
public:
    typedef size_t size_type;
```

```
typedef ptrdiff_t difference_type;
typedef T*  pointer;
typedef const T* const_pointer;
typedef T& reference;
typedef const T& const_reference;
typedef T value_type;

//Allocator::rebind<T2>::other
template <typename V>
struct rebind
{
    typedef Allocator<V> other;
};

pointer address(reference value) const
{
    return &value;
}
const_pointer address(const_reference value) const
{
    return &value;
}

Allocator() throw() {   }
Allocator(const Allocator &) throw() {  }
//不同类型的allcator可以相互复制
template <typename V> Allocator(const Allocator<V> &other) { }
~Allocator() throw() {  }

//最多可以分配的数目
size_type max_size() const throw()
{
    return std::numeric_limits<size_type>::max() / sizeof(T);
}

//分配内存，返回该类型的指针
pointer allocate(size_type num)
{
    return (pointer)(::operator new(num * sizeof(T)));
}

//执行构造函数，构建一个对象
void construct(pointer p, const T &value)
{
    new ((void*)p) T(value);
}

//销毁对象
void destroy(pointer p)
{
    p->~T();
}

//释放内存
void deallocate(pointer p, size_type num)
{
    ::operator delete((void *)p);
}
```

```
};

template <typename T, typename V>
bool operator==(const Allocator<T> &, const Allocator<V> &) throw()
{
    return true;
}

template <typename T, typename V>
bool operator!=(const Allocator<T> &, const Allocator<V> &) throw()
{
    return false;
}

#endif
```

测试代码：

```
#include "Allocator.hpp"
#include <string>
#include <vector>
#include <iostream>
using namespace std;

class Test
{
public:
    Test()
    {
       cout << "Test Constructor" << endl;
    }
    Test(const Test& other)
    {
       cout << "Test Copy Constructor" << endl;
    }
    ~Test()
    {
       cout << "Test DeConstructor" << endl;
    }
};

class Test2
{
public:
    Test2()
    {
       cout << "Test2 Constructor" << endl;
    }
    Test2(const Test2& other)
    {
       cout << "Test2 Copy Constructor" << endl;
    }
    ~Test2()
    {
       cout << "Test2 DeConstructor" << endl;
    }
```

```
};


int main(int argc, char const *argv[])
{
    // 定义容器时指定分配器
    vector<string, Allocator<string> > vec(10, "haha");

    vec.push_back("foo");
    vec.push_back("bar");
    // 使用Test分配器分配Test2类型
    Allocator<Test>::rebind<Test2>::other alloc;
    Test2 *pTest = alloc.allocate(5);
    alloc.construct(pTest, Test2());
    alloc.construct(pTest + 1, Test2());
    alloc.construct(pTest + 2, Test2());

    alloc.destroy(pTest);
    alloc.destroy(pTest + 1);
    alloc.destroy(pTest + 2);

    alloc.deallocate(pTest, 3);

    return 0;
}
```

5、`mt allocator`

`mt allocator（__gnu_cxx::__mt_alloc）`是`STL`扩展库的支持多线程应用的内存分配器，是为多线程应用程序设计的固定大小（2的幂）内存的分配器，目前在单线程应用程序表现一样出色。

`mt allocator`由3个部分组成：描述内存池特征的参数；关联内存池到通用或专用方案的`policy`类；从`policy`类继承的实际内存分配器类。

（1）线程支持参数模板类

`template<bool _Thread> class __pool;`

表示是否支持线程，然后对多线程（`booltrue`）和单线程（`boolfalse`）情况进行显式特化，开发者可以使用定制参数替代。

（2）内存池`Policy`类

通用内存池策略:
`__common_pool_policy`实现了一个通用内存池，即使分配的对象类型不同（比如`char`和`long`）也使用同一个的内存池，是默认策略。

```
template<bool _Thread>
struct __common_pool_policy;
```

专用策略类:

`__per_type_pool_policy`会对每个对象类型都实现一个单独的内存池，因此不同对象类型会使用不同的内存池，可以对某些类型进行单独调整。

```
template<typename _Tp, bool _Thread>
struct __per_type_pool_policy;
```

(3) 内存分配器

```
template<typename _Tp, typename _Poolp = __default_policy>
class __mt_alloc : public __mt_alloc_base<_Tp>,  _Poolp
template<typename _Tp,
         typename _Poolp = __common_pool_policy<__pool, __thread_default> >
class __mt_alloc : public __mt_alloc_base<_Tp>
{
public:
    typedef size_t                          size_type;
    typedef ptrdiff_t                       difference_type;
    typedef _Tp*                            pointer;
    typedef const _Tp*                      const_pointer;
    typedef _Tp&                            reference;
    typedef const _Tp&                      const_reference;
    typedef _Tp                             value_type;
    typedef _Poolp                          __policy_type;
    typedef typename _Poolp::pool_type      __pool_type;

    template<typename _Tp1, typename _Poolp1 = _Poolp>
    struct rebind
    {
        typedef typename _Poolp1::template _M_rebind<_Tp1>::other pol_type;
        typedef __mt_alloc<_Tp1, pol_type> other;
    };
    __mt_alloc() _GLIBCXX_USE_NOEXCEPT { }

    __mt_alloc(const __mt_alloc&) _GLIBCXX_USE_NOEXCEPT { }

    template<typename _Tp1, typename _Poolp1>
    __mt_alloc(const __mt_alloc<_Tp1, _Poolp1>&) _GLIBCXX_USE_NOEXCEPT { }

    ~__mt_alloc() _GLIBCXX_USE_NOEXCEPT { }

    pointer
    allocate(size_type __n, const void* = 0);

    void
    deallocate(pointer __p, size_type __n);

    const __pool_base::_Tune
    _M_get_options()
    {
        // Return a copy, not a reference, for external consumption.
        return __policy_type::_S_get_pool()._M_get_options();
    }

    void
    _M_set_options(__pool_base::_Tune __t)
    {
        __policy_type::_S_get_pool()._M_set_options(__t);
    }
};
```

（4）内存池特征参数调整

`mt allocator`提供了用于调整内存池参数的嵌套类`_Tune`。

```
struct _Tune
{
    enum { _S_align = 8 };
    enum { _S_max_bytes = 128 };
    enum { _S_min_bin = 8 };
    enum { _S_chunk_size = 4096 - 4 * sizeof(void*) };
    enum { _S_max_threads = 4096 };
    enum { _S_freelist_headroom = 10 };

    size_t    _M_align;// 字节对齐
    size_t    _M_max_bytes;// 128字节以上的内存直接用new分配
    size_t    _M_min_bin;//可分配的最小的内存块大小8字节
    size_t    _M_chunk_size;//每次从os申请的内存块的大小为4080字节
    size_t    _M_max_threads;//可支持的最多的线程数是4096
    size_t    _M_freelist_headroom;//单线程能保存的空闲块的百分比为10%
    bool      _M_force_new;//是否直接使用new和delete 根据 是否设置
getenv("GLIBCXX_FORCE_NEW")
};
```

`_Tune`参数的设置和获取接口如下：

```
const _Tune& _M_get_options() const
{
    return _M_options;
}

void _M_set_options(_Tune __t)
{
    if (!_M_init)
        _M_options = __t;
}
```

内存池参数调整必须在任何内存分配动作前。

`mt allocator`实例如下：

```
#include <ext/mt_allocator.h>
#include <string.h>

class Test
{
public:
    Test(int id = 0)
    {
        memset(this, 0, sizeof(Test));
        this->id = id;
    }
private:
    int id;
    char name[32];
```

```
};

typedef __gnu_cxx::__mt_alloc<Test> TestAllocator;
typedef __gnu_cxx::__pool_base::_Tune TestAllocatorTune;

int main()
{
    TestAllocator pool;

    TestAllocatorTune t_defaut;
    TestAllocatorTune t_opt(16, 5120, 32, 5120, 20, 10, false);
    TestAllocatorTune t_single(16, 5120, 32, 5120, 1, 10, false);

    TestAllocatorTune t;
    t = pool._M_get_options();
    pool._M_set_options(t_opt);
    t = pool._M_get_options();
    // allocate
    TestAllocator::pointer p1 = pool.allocate(sizeof(Test));
    TestAllocator::pointer p2 = pool.allocate(5120);
    // free
    pool.deallocate(p1, sizeof(Test));
    pool.deallocate(p2, 5120);

    return 0;
}
```

如果分配内存大于内存池特征参数设置的值，将会抛出异常或导致段错误。

三、`Boost`内存池

1、`pool`

`pool`是一个`Object Usage`的内存池，每个内存池都是一个可以创建和销毁的对象，一旦内存池被销毁则其所分配的所有内存都会被释放，溢出时返回`NULL`。

`pool`内存池是最基本的的定长内存池，只可用于内嵌数据类型（如`int，char`等）的分配，而不适合用于复杂的类和对象。`pool`内存池只简单地分配内存而不调用类构造函数，对于复杂的类和对象，则应该使用`object_pool`内存池。

`boost::pool`接口函数如下：

`bool release_memory();`

释放所有空闲block给操作系统。pool对空闲block的判断标准是：block中所有的chunk都在空闲链表中并且空闲链表有序

`bool purge_memory();`

释放所有的block给操作系统，不管block中的chunk块有没有被分配；pool析构函数会调用完成内存释放

`size_type get_next_size() const;`

获取下一次要分配的block的大小

`size_type set_next_size(const size_type nnext_size);`

设置下一次要分配的block的大小

`size_type get_requested_size();`

获取每个内存块的大小；以Byte为单位

`void * malloc();`

分配一个chunk块；如果pool中没有剩余空间供分配，则会向操作系统申请一块新的block

`void * ordered_malloc();`

分配一个chunk块；如果没有足够的空间进行分配，那么pool向操作系统申请新的block并将block分块后，会进行block链表的排序以保证内存块有序

`void * ordered_malloc(size_type n);`

从内存池中请求物理地址连续的n块内存，可以用来进行内存数组的分配；函数要求某个block中的chunk链表是有序的，否则，即使有连续内存存在，仍然有可能分配失败从而再申请的新的block内存块。在内存池的`request_size`（内存池初始化时指定的内存块大小）小于`size_type(void *)`时，内存池并不会分配n块chunk，而是分配`ceil(n*request_size/size_type(void *))`块内存

`void free(void * const chunk);`

将`malloc`申请到的内存返回给内存池

`void ordered_free(void * const chunk);`

将`malloc`申请到的内存返回给内存池并保证空闲chunk链表有序

`void free(void * const chunks, const size_type n);`

返回chunk开头的连续n块内存给内存池

`void ordered_free(void * const chunks, const size_type n);`

返回chunk开头的连续n块内存给内存池并保持空闲chunk链表有序

`bool is_from(void * const chunk) const;`

判断chunk是否由本内存池所释放的

```
#include <boost/pool/pool.hpp>
#include <iostream>

using namespace std;

int main()
{
    boost::pool<> testpool(sizeof(int));
    for(int i = 0; i < 1000; i++)
    {
        int* pn = (int*)testpool.malloc();
        *pn = i + 1;
        cout << *pn << endl;
    }
    return 0;
}
g++ test.cpp -o test -lboost_system
```

2、object_pool

Pool是一个`Object Usage`的内存池，每个内存池都是一个可以创建和销毁的对象，一旦内存池被销毁则其所分配的所有内存都会被释放，溢出时返回`NULL`。

`object_pool`对象内存池适用于对复杂对象的分配和释放，除了分配释放内存外，`object_pool`会调用对象的构造函数和析构函数。

`object_pool`内存池的内存分配算法与pool不同，`object_pool`内存池的分配和释放要比pool慢的多。

`object_pool`接口如下：

`element_type * malloc();`

分配一块内存，调用pool的`ordered_malloc`函数；

`void free();`

释放一块内存，调用pool的`ordered_free`函数

`element_type * construct();`

分配一块内存并调用构造函数

`void destroy(element_type * const chunk);`

析构一个对象并将其内存返回给内存池

```
#include <boost/pool/object_pool.hpp>
#include <iostream>

class Test
{
public:
    Test(int id = 0)
    {
        this->id = id;
    }
    int id;
};

using namespace std;

int main()
{
    boost::object_pool<Test> testpool;
    Test* pTest1 = testpool.malloc();
    cout << pTest1->id << endl;
    Test* pTest2 = testpool.construct();
    cout << pTest2->id << endl;
    testpool.free(pTest1);
    return 0;
}
g++ test.cpp -o test -lboost_system
```

3、singleton_pool

singleton_pool是一个`Singleton Usage`的内存池，每个内存池都是一个被静态分配的对象，直至程序结束才会被销毁，溢出时返回`NULL`。

singleton_pool内存池是线程安全的，只有使用`release_memory`或者`purge_memory`方法才能释放内存。

singleton_pool则是pool的一个单例模式的实现，其接口和pool相同，并且通过互斥量的方法来保证线程安全。

4、pool_alloc

`pool_alloc`是一个Singleton Usage的内存池，溢出时抛出异常。

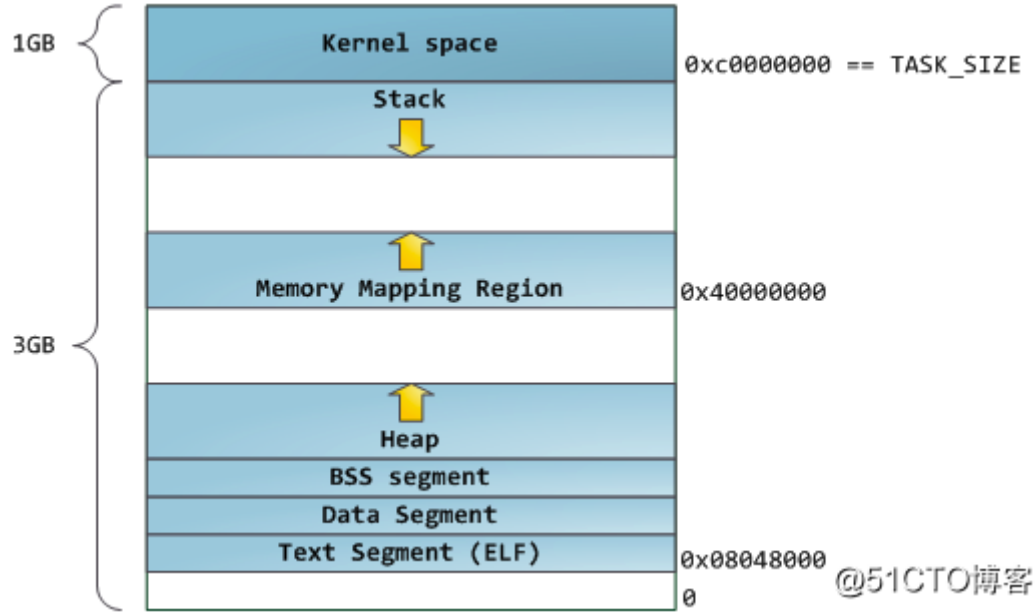
`pool_alloc`提供了两个用于标准容器类型的分配器：`pool_allocator`和`fast_pool_allocator`。但`STL`标准容器提供了自己的内存分配器，通常应当使用`STL`标准的内存分配器进行内存分配，而不要使用`pool_alloc`提供的内存分配器。

C++性能优化（八）——内存分配机制

一、操作系统内存布局

1、32位系统经典内存布局

Linux Kernel 2.6.7前版本采用的默认内存布局形式如下：



（1）32操作系统中，loader将可执行文件的各个段次依次载入到从0x80048000（128M）位置开始的空间中。应用程序能够访问的最后地址是0xbfffffff(3G)的位置，3G以上的位置是给内核使用的，应用程序不能直接访问。

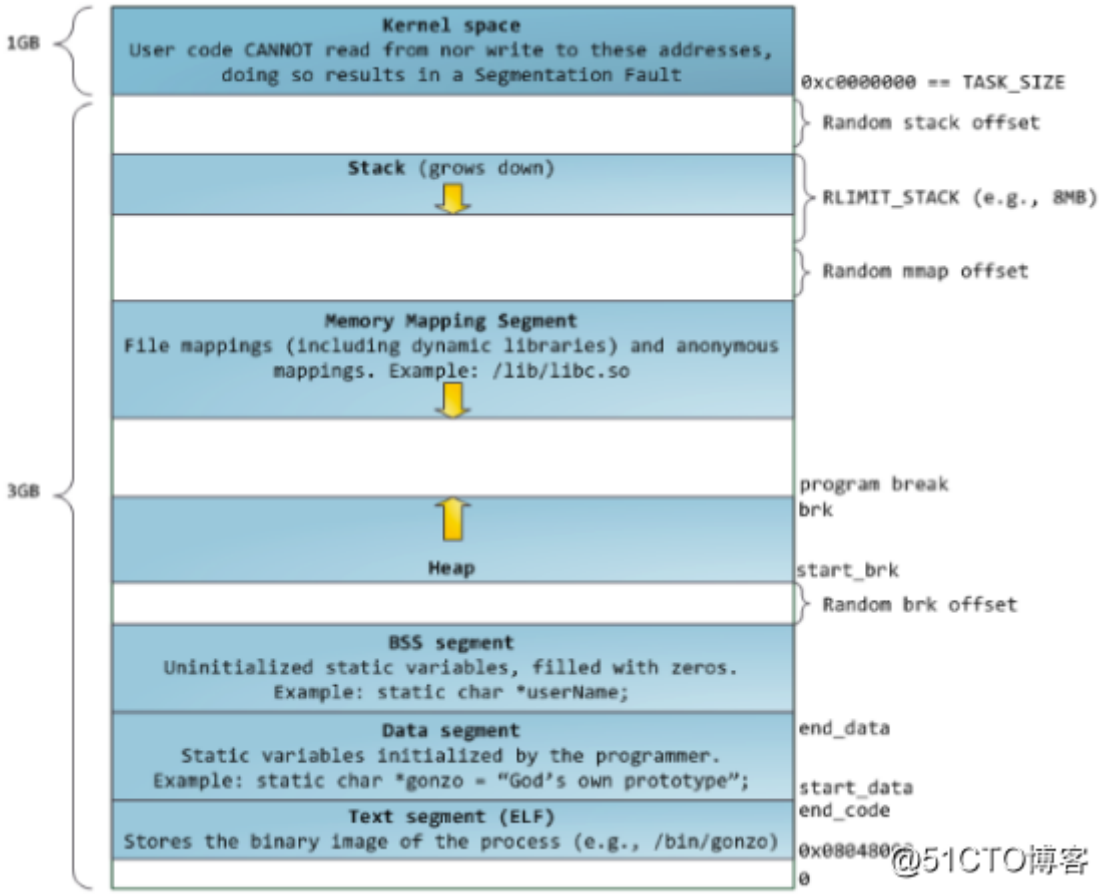
（2）内存布局从低地址到高地址依次为：txet段、data段、bss段、heap、mmap映射区、stack栈区。

（3）heap和mmap是相对增长的，heap只有1G的虚拟地址空间可供使用。

（4） stack空间不需要映射，用户可以直接访问栈空间，因此是利用堆栈溢出进行**的基础。**
起始*1GB*地址为内核空间，随后是向下增加的栈空间和由*0x40000000*向上增加的*MMAP*地址；堆空间从底部开始，去除*ELF*、数据段、代码段、常量段后的地址，并向上增长。缺点是容易遭受溢出，堆地址空间只有不到1GB。

2、32位系统默认内存布局

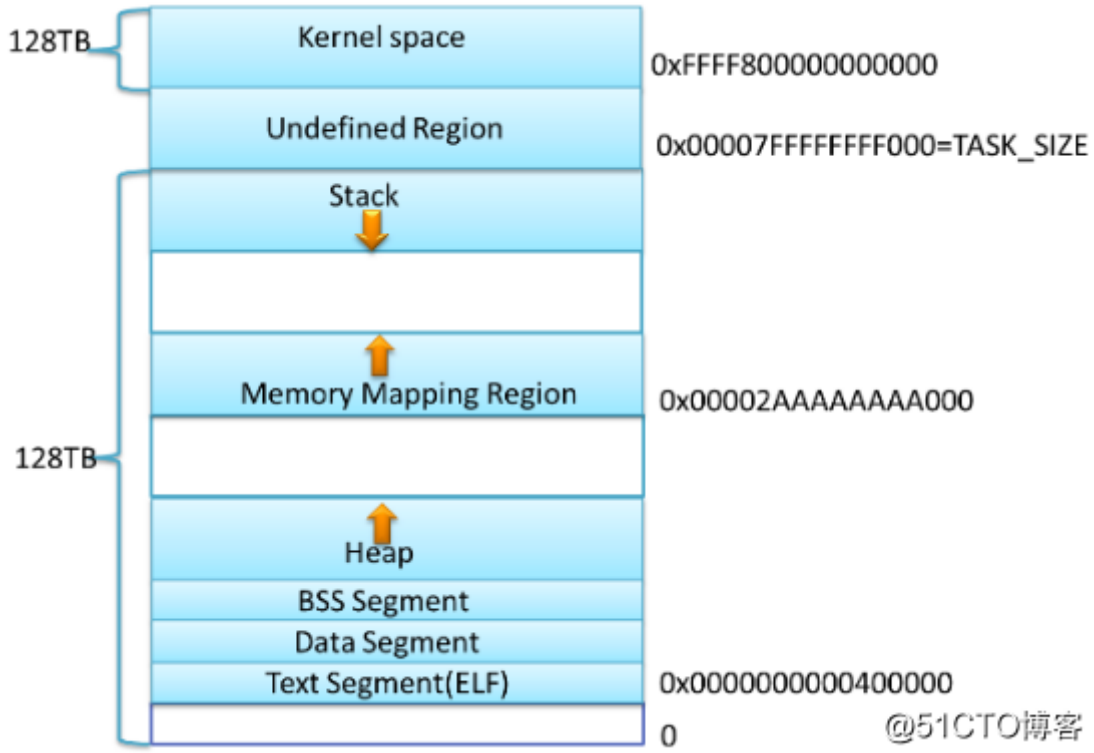
Linux Kernel 2.6.7版本后32位操作系统的默认内存布局方式如下：



在经典内存布局基础上增加了Random offset随机偏移，不容易遭受溢出***；堆地址向上增长，但MMAP向下增长，栈空间不是动态增长的，会受到限制；内存地址利用率较高。

栈自顶向下扩展，但栈有边界，因此栈大小有限制（ulimit -s查看)。堆自底向上扩展，mmap映射区自顶向下扩展，mmap和heap是相对扩展，直至消耗尽虚拟地址空间中的剩余区域。

3、64位系统内存布局



64位操作系统的寻址空间比较大，沿用32位操作系统的经典内存布局，增加随机MMAP地址，防止溢出。

二、操作系统内存管理机制

1、操作系统内存管理简介

内存管理自底向上分为三个层次：

（1）操作系统内核的内存管理。

（2）glibc层使用系统调用维护的内存管理算法。

（3）应用程序从glibc动态分配内存后，根据应用程序本身的程序特性进行优化，比如使用引用计数std::shared_ptr，内存池方式等等。

应用程序可以直接使用系统调用从内核分配内存，根据程序特性自己维护内存，但会大大增加开发成本。

2、操作系统内存管理机制

Linux Kernel内存管理的基本思想是内存延迟分配，即只有在真正访问一个地址的时候才建立地址的物理映射。Linux Kernel在用户申请内存的时候，只分配一个虚拟地址，并没有分配实际物理地址，只有当用户使用内存时，Linux Kernel才会分配具体的物理地址给用户使用。

对于大内存，通常不同的内存分配方式都是直接MMAP；对于小数据，则通过向操作系统申请扩大堆顶，操作系统会把内存分页映射到进程堆空间，再由malloc管理内存堆块，减少系统调用；free内存时，不同内存分配方式有不同策略，不一定会将内存还给操作系统，因此如果访问释放的内存并不会立即Run Time Error，只有访问的地址没有对应的内存分页才会。

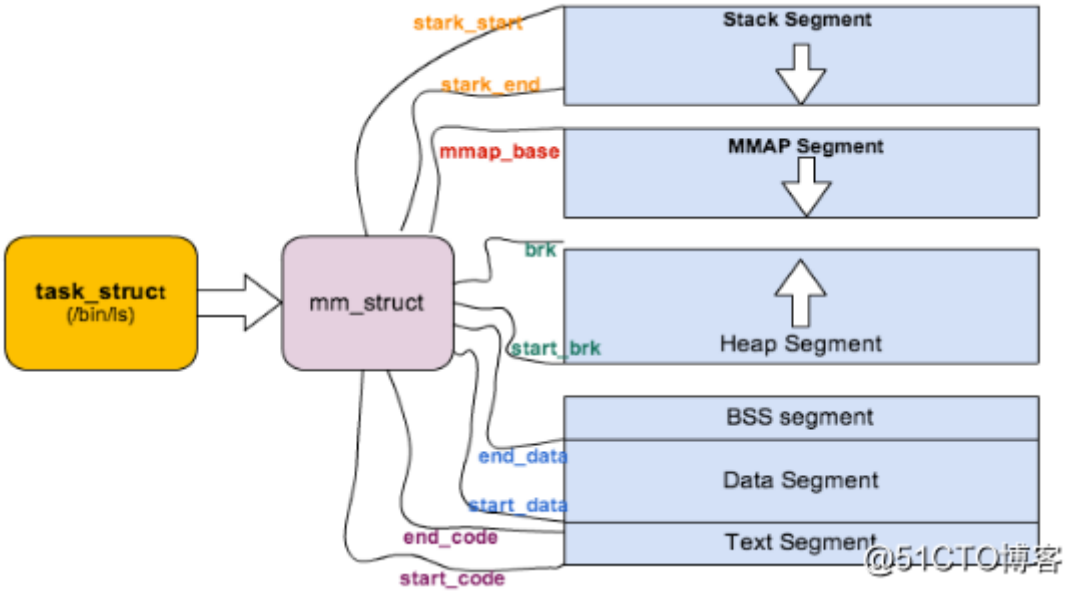
对于heap操作，操作系统提供brk系统调用，c运行库提供sbrk库函数；对于mmap映射区操作，操作系统提供了mmap和munmap系统调用。

3、heap操作系统调用接口

```
#include <unistd.h>
int brk(void *addr);
void *sbrk(intptr_t increment);
```

当参数increment为0时，sbrk返回进程当前的brk值，increment为正数时扩展brk值，increment为负数时收缩brk值。

brk是系统调用，sbrk是库函数。C语言的动态内存分配基本函数是malloc，在glibc中malloc函数调用库函数sbrk，sbrk调用brk函数。brk系统调用只是简单的改变mm_struct结构体的成员变量brkd的值。



start_code和end_code是进程代码段的起始和结束地址、start_data和end_data是进程数据段的起始和终止地址。

start_stack是进程堆栈段的起始地址。

start_brk是进程动态内存分配的起始地址（堆的起始地址）。

brk是进程动态内存分配当前的终止地址（堆的当前最后地址）。

4、mmap操作系统调用接口

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags,int fd, off_t offset);
int munmap(void *addr, size_t length);
```

addr：映射区的开始地址。

length：映射区的长度。

prot：期望的内存保护标志。

flags：指定映射对象的类型，映射选项和映射页是否可以共享。

fd：有效的文件描述符。

offset：被映射对象内容的起点。

mmap函数将一个文件或者其它对象映射进内存，文件被映射到多个页上，如果文件大小不是所有页大小之和，最后一个页不被使用的空间将会清零；munmap函数删除特定地址区域的对象映射。

三、ptmalloc2简介

1、ptmalloc2简介

ptmalloc/ptmalloc2/ptmalloc3版本基于dlmalloc进行开发，
Wolfram Gloger在Doug Lea的基础上对dlmalloc进行了改进，使得ptmalloc可以支持多线程。Linux操作系统默认使用glibc malloc版本，即ptmalloc2，glibc-2.3.x中已经集成了ptmalloc2，目前ptmalloc最新版本为ptmalloc3。

ptmalloc内存分配器处于内核和用户程序之间，用于响应用户的内存分配请求，向操作系统申请内存，然后将内存返回给用户程序。为了保证高效，内存分配器一般都会预先分配一块大于用户请求的内并进行管理；用户释放掉的内存不会立即返回给操作系统，内存分配器会管理被释放掉的空闲空间以应对用户以后的内存分配请求。内存分配器不仅需要管理分配的内存块，还需要管理空间的内存块。当响应用户的请求时，内存分配器会首先在空闲空间中寻找一块合适的内存返回给用户，在空闲空间中找不到时才会重新分配一块新内存。

2、ptmalloc2多线程支持

dlmalloc内存分配器中只有一个主分配区，每次分配内存都必须对主分配区加锁，分配完成后再释放锁。在多线程的环境下，对主分配区的锁的竞争很激烈，严重影响内存分配效率。

ptmalloc内存分配器可以支持多线程，增加了非主分配区支持，主分配区和非主分配区形成一个环形链表进行管理，每一个分配区使用互斥锁保证多线程访问安全。每个进程只有一个主分配区，可以有多个非主分配区，ptmalloc根据系统对分配区的争用情况动态的增加非主分配区的个数，分配区的数量一旦增加就不会再减少。

主分配区可以访问进程的heap和mmap映射区域，即主分配区可以使用brk、sbrk、mmap向操作系统申请虚拟内存；非主分配区只能访问进程的mmap映射区域，非主分配区每次使用mmap系统调用向操作系统申请HEAP_MAX_SIZE（32位系统1M，64位系统64M）大小的虚拟内存，当用户向非主分配区请求分配内存时再分割成小块内存进行分配。

由于系统调用的效率比较低，直接从用户空间分配内存效率会比较高，因此ptmalloc只在必要情况下才会调用mmap系统调用向操作系统申请内存。

主分配区可以访问heap区域，如果用户不调用sbrk或者brk函数，分配程序就可以保证分配到连续的虚拟内存，因为一个进程只有一个主分配区使用sbrk分配heap区域的虚拟内存。如果主分配区的内存是通过mmap向系统申请的，当free内存时，主分配区会直接调用munmap将内存归还给操作系统。

当线程需要使用malloc函数分配内存空间时，线程会先查看线程的私有变量中是否已经存在有一个分配区。如果存在，尝试对分配区加锁，如果加锁成功就使用相应分配区分配内存；如果失败，线程就搜索环形链表试图获得一个没有加锁的分配区来分配内存；如果所有分配区都加锁，那么malloc会开辟一个新分配区，把新分配区添加到循环链表中并加锁，使用新分配区进行分配内存操作。

在释放操作中，线程试图获得待释放内存块所在的分配区的锁，如果分配区正在被其它线程使用，则需要等待其它线程释放分配区的互斥锁后才可以进行释放操作。

为了支持多线程，多个线程可以从同一个分配区中分配内存，ptmalloc假设线程A释放掉一块内存后，线程B会申请类似大小的内存，但A释放的内存跟B需要的内存不一定完全相等，可能有一个小的误差，就需要不停地对内存块作切割和合并，因此分配过程中可能产生内存碎片。

3、ptmalloc2缺点

（1）ptmalloc收缩内存从top chunk开始，如果与top chunk相邻的chunk不能释放，top chunk以下都不能释放。

（2）多线程锁开销大，需要避免多线程频繁分配释放。

（3）多线程使用内存不均衡时，容易导致内存浪费。

（4）每个chunk至少8字节的开销很大。

（5）不定期分配长生命周期的内存容易造成内存碎片，不利于回收。

四、ptmalloc2 BINS架构

1、chunk简介

若每次申请内存都调用sbrk、brk、mmap，那么每次都会产生系统调用，影响性能，并且容易产生内存碎片，因为堆是从低地址到高地址，如果高地址的内存没有被释放，低地址的内存就不能被回收。

ptmalloc使用内存池管理方式，采用边界标记法将内存划分成很多块，从而对内存的分配与回收进行管理。为了高效分配内存，ptmalloc会预先向操作系统申请一块内存供用户使用，当申请和释放内存时，ptmalloc会对相应内存块进行管理，并通过内存分配策略判断是否将其回收给操作系统，使用户申请和释放内存更加高效，避免产生过多的内存碎片。

用户请求分配的内存在ptmalloc中使用chunk表示， 每个chunk至少需要8个字节额外的开销。 用户释放掉的内存不会马上归还操作系统，ptmalloc会统一管理heap和mmap区域的空闲chunk，避免了频繁的系统调用。

```
struct malloc_chunk {
  INTERNAL_SIZE_T      prev_size;    /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T      size;         /* Size in bytes, including overhead. */
  struct malloc_chunk* fd;           /* double links -- used only if free. */
  struct malloc_chunk* bk;
  /* Only used for large blocks: pointer to next larger size.  */
  struct malloc_chunk* fd_nextsize;  /* double links -- used only if free. */
  struct malloc_chunk* bk_nextsize;
};
```

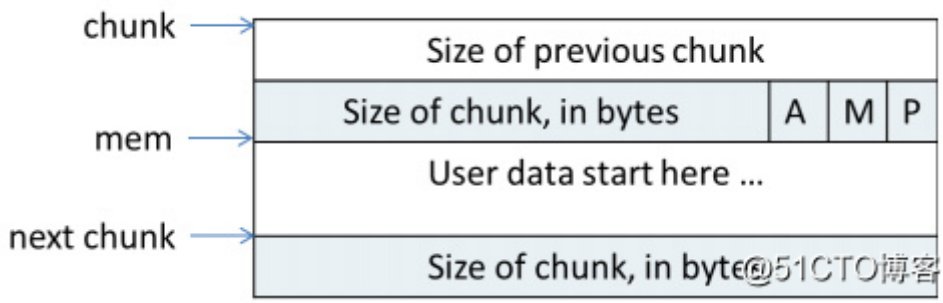
prev_size: 如果前一个chunk空闲，表示前一个chunk的大小；如果前一个chunk不空闲，字段无意义。

size：当前chunk的大小，并且记录了当前chunk和前一个chunk的一些属性，包括前一个chunk是否在使用中，当前chunk是否是通过mmap获得的内存，当前chunk是否属于非主分配区。

fd和bk：指针fd和bk只有当chunk空闲时才存在，用于将对应的空闲chunk块加入到空闲chunk块链表中统一管理，如果chunk 块被分配给应用程序使用，chunk块被从空闲chunk链表中拆出，那么这两个指针也就没有用了，所以也当作应用程序的使用空间，而不至于浪费。

fd_nextsize和bk_nextsize：当前chunk存在于large bins 中时， large bins中空闲chunk按照大小排序，但同一个大小的 chunk可能有多个，增加fd_nextsize和bk_nextsize字段可以加快遍历空闲chunk，并查找满足需要的空闲chunk， fd_nextsize 指向下一个比当前chunk大小大的第一个空闲chunk，bk_nextsize指向前一个比当前chunk大小小的第一个空闲 chunk 。

使用中的chunk结构如下：



chunk指针指向chunk开始的地址；

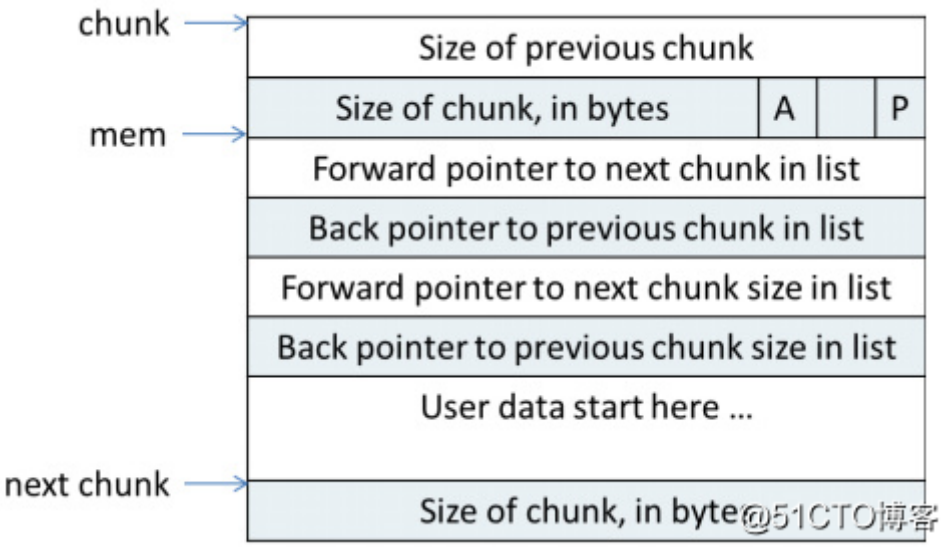
mem指针指向用户内存块开始的地址。

P=0时，表示前一个chunk为空闲，prev_size才有效；P=1时，表示前一个chunk正在使用，prev_size无效；P主要用于内存块的合并操作。ptmalloc分配的第一个块总是将P设为1，以防止程序引用到不存在的区域。

M=1为mmap映射区域分配；M=0为heap区域分配。

A=1为非主分区分配；A=0为主分区分配。

空闲chunk结构如下：



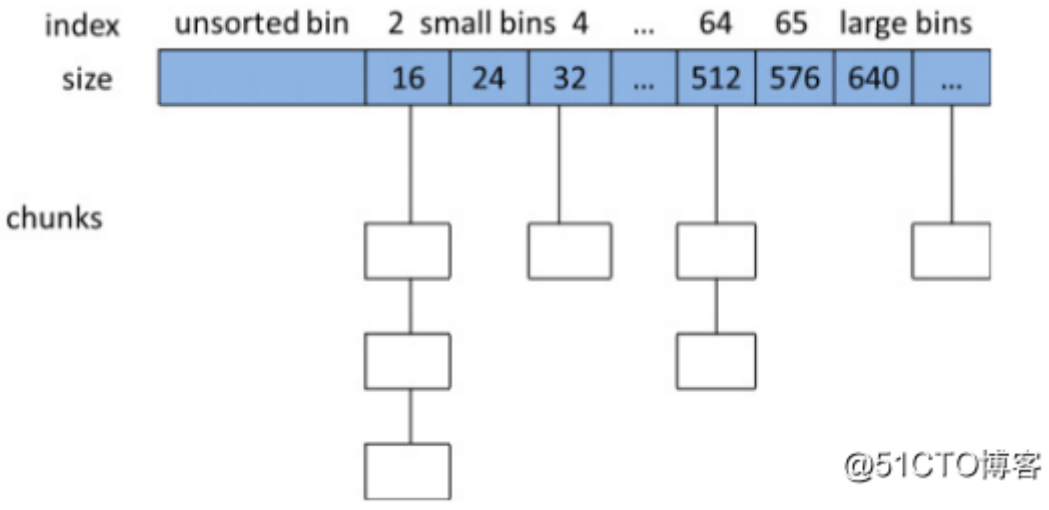
空闲的chunk会被放置到空闲的链表bins上。当用户申请分配内存时，会先去查找空闲链表bins上是否有合适的内存。

fp和bp分别指向前一个和后一个空闲的chunk。

fp_nextsize和bp_nextsize分别指向前一个空闲chunk和后一个空闲chunk的大小，用于在空闲链表上快速查找合适大小的chunk。

2、BINS架构简介

ptmalloc将相同大小的chunk用双向链表链接起来，称为bin。ptmalloc共维护128个bin，并使用一个数组来存储bin。



数组中第1个bin为unsorted bin，如果被用户释放的chunk大于max_fast或者fast bins中的空闲chunk合并后，chunk首先会被放到unsorted bin队列中。如果unsorted bin不能满足分配要求，ptmalloc会将unsorted bin中的chunk加入bins中，然后再从bins中继续进行查找和分配过程。unsorted bin是bins的一个缓冲区，可以加快内存分配速度。

数组中第2个至第64个的bin称为small bin，同一个small bin 中的chunk具有相同的大小，相邻的small bin中的chunk大小相差为8字节。

数组中第65个开始bin称为large bin。large bins中的每一个 bin分别包含了一个给定范围内的chunk，其中chunk按大小序排列，相同大小的chunk按照最近使用顺序排列。

3、Fast Bins

程序在运行时会经常需要申请和释放小块内存空间。当内存分配器合并相邻的若干chunk后，可能立即会有另一个小块内存的请求，内存分配器需要从大块空闲内存中分割出一块内存，效率低下，因此ptmalloc在分配过程中引入了fast bins。

fast bins是bins的高速缓冲区，每个fast bin是空闲chunk的单链表，采用单链表是由于fast bin中增删chunk都发生在链表的前端。fast bins中包含7个fast bin，fast bin中的chunk大小以8字节递增，分别为16、24、32、40、48、56、64。

当用户申请分配不大于max_fast（默认值64字节）内存块时，ptmalloc首先会先到fast bins中相应chunk尺寸的fast bin寻找是否有合适chunk，在fast bin中被检索出的第一个chunk将被摘除并返回给用户；当用户释放一块不大于max_fast（默认值64字节）的chunk时，默认会被放到fast bins中chunk尺寸大小的fast bin的前端，除非特定情况，两个相邻空闲chunk并不会被合并成一个空闲chunk，不合并可能会导致碎片化问题，但可以大大加速释放过程。

4、Unsorted Bin

Unsorted Bin存储在bins数组的第1个，是bins的缓冲区，用于加快内存分配速度。当用户释放的chunk大于max_fast或者fast bins中合并后的chunk都会首先进入unsorted bin上。Unsorted Bin用双向链表管理chunk，chunk无大小限制，任何大小chunk都可以添加进Unsorted Bin。

Unsorted Bin提供了ptmalloc重复使用最近释放的chunk，因此内存的分配和释放会更快。

用户申请分配内存时，如果在fast bins中没有找到合适的chunk，则ptmalloc会先在Unsorted Bin中查找合适的空闲chunk；如果没有合适的chunk，ptmalloc会将unsorted bin上的chunk放入bins上，然后继续到bins上查找合适的空闲chunk。

5、Small Bins

小于512字节的chunk即small chunk，保存small chunk的bin即small bin。bins数组从第2至第64的63个bin为Small Bin，Small Bins中相邻bin之间相差8个字节，同一个Small Bin中的chunk具有相同大小，第0个Small Bin中的chunk大小为16字节，第62个Small Bin中的chunk大小为512字节。

每个Small Bin是一个空闲chunk的双向循环链表，释放掉的chunk添加在链表的前端，而分配出的chunk则从链表后端摘除并返回给用户。

分配chunk时，从相应chunk大小的small bin中摘除最后一个chunk并返回给用户；释放chunk时，ptmalloc会检查相邻chunk是否空闲，若是则将相邻chunk从所属的small bin中摘除并合并成一个新chunk，新chunk会添加在unsorted bin的前端；合并chunk消除了碎片化的影响但减慢了释放速度。

6、Large Bins

大小大于等于512字节的chunk即Large Chunk，保存Large Chunk的bin即Large Bin，位于bins数组中small bins后。Large Bins中的每一个bin分别包含了一个给定范围内的chunk，其中的chunk按大小递减排序，大小相同则按照最近使用时间排列。

Large Bins包含63个bin，每个bin中的chunk大小不是一个固定的等差数列，而是分成6组；每组bin中chunk数量是一个固定的等差数列，依次为32、16、8、4、2、1，chunk大小公差依次为64B、512B、4096B、32768B、262144B等。

7、Top Chunk

top chunk是分配区的顶部空闲内存，当bins上都不能满足内存分配要求时，就会在top chunk上进行分配。

当top chunk大小比用户所请求大小还大的时候，top chunk会分割为两个部分：User chunk（用户请求大小）和Remainder chunk（剩余大小），Remainder chunk会成为新的top chunk；当top chunk大小小于用户所请求chunk大小时，top chunk会通过sbrk（主分配区）或mmap（非主分配区）系统调用来扩容。

主分配区是唯一能够映射进程heap区域的分配区，可以通过sbrk来增大和收缩进程heap的大小。ptmalloc在开始的时候会预先分配一块较大的空闲内存。主分配区的top chunk在第一次调用malloc时会分配一块空间作为初始化的heap。

非主分配区会预先从mmap区域分配一块较大的空闲内存模拟 sub-heap，通过管理sub-heap来响应用户的需求，因为内存是按地址从低向高进行分配的，在空闲内存的最高处存在着一块空闲 chunk，即top chunk。当fast bin和bins都满足不了用户的内存分配需求时，ptmalloc会从top chunk分出一块内存给用户，如果top chunk空间不足，会重新分配一个sub-heap，将top chunk迁移到新的sub-heap上。在分配过程中，top chunk的大小随着切割动态变化。

主分配区是唯一能够映射进程heap区域的分配区，可以通过sbrk来增大或收缩进程heap大小。top chunk在heap的最上面，如果申请内存时，top chunk空间不足，ptmalloc会调用sbrk将进程heap的边界brk上移，然后修改top chunk的大小。

8、mmaped chunk

当需要分配的chunk足够大，fast bins和bins都不能满足要求，甚至top chunk都不能满足分配需求时，ptmalloc会使用mmap来直接使用页映射机制来将页映射到进程空间，放到mmaped chunk上，当释放mmaped chunk上内存的时候会直接交还给操作系统。

mmap分配阈值默认为128KB，分配阈值可以动态调整。如果开启mmap分配阈值的动态调整机制，并且当前回收的chunk大小大于mmap分配阈值，将mmap分配阈值设置为chunk的大小，将mmap收缩阈值设定为mmap分配阈值的2倍。

9、Last remainder chunk

Last remainder chunk是一种特殊chunk，不会在任何bins中找到。当需要分配一个small chunk，但在small bins中找不到合适的chunk，如果last remainder chunk的大小大于所需要的small chunk大小，last remainder chunk被分割成两个chunk，其中一个chunk返回给用户，另一个chunk变成新的last remainder chunk。

C++性能优化（九） —— TCMalloc

一、TCMalloc简介

1、TCMalloc简介

TCMalloc(Thread-Caching Malloc，线程缓存的malloc）是Google开发的内存分配算法库，最初作为Google性能工具库 perftools 的一部分，提供高效的多线程内存管理实现，用于替代操作系统的内存分配相关的函数（malloc、free，new，new[]等），具有减少内存碎片、适用于多核、更好的并行性支持等特性。

TCMalloc属于gperftools，gperftools项目包括heap-checker、heap-profiler、cpu-profiler、TCMalloc等组件。

gperftools源码地址：

https://github.com/gperftools/gperftools
TCMalloc源码地址：

https://github.com/google/tcmalloc

2、TCMalloc安装

（1）TCMalloc源码安装

bazel源增加：

```
/etc/yum.repos.d/bazel.repo
```

```
[copr:copr.fedorainfracloud.org:vbatts:bazel]
name=Copr repo for bazel owned by vbatts
baseurl=https://download.copr.fedorainfracloud.org/results/vbatts/bazel/epel-7-$basearch/
type=rpm-md
skip_if_unavailable=True
gpgcheck=1
gpgkey=https://download.copr.fedorainfracloud.org/results/vbatts/bazel/pubkey.gpg
repo_gpgcheck=0
enabled=1
enabled_metadata=1
```

在线安装`bazel`：

```
yum install bazel3
```

TCMalloc源码下载：

```
git clone https://github.com/google/tcmalloc.git
cd tcmalloc && bazel test //tcmalloc/...
```

由于TCMalloc依赖gcc 9.2+,clang 9.0+: -std=c++17，因此推荐使用其它方式安装。

（2）gperftools源码安装

gperftools源码下载：

```
git clone https://github.com/gperftools/gperftools.git
```

生成构建工具：

```
autogen.sh
```

配置编译选项：

```
configure --disable-debugalloc --enable-minimal
```

编译：make -j4

安装：make install

TCMalloc库安装在/usr/local/lib目录下。

（3）在线安装

epel源安装：

```
yum install -y epel-release
```

gperftools安装：

```
yum install -y gperftools.x86_64
```

3、Linux64位系统支持

在Linux64位系统环境下，gperftools使用glibc内置的stack-unwinder可能会引发死锁，因此官方推荐在配置和安装gperftools前，先安装libunwind-0.99-beta。

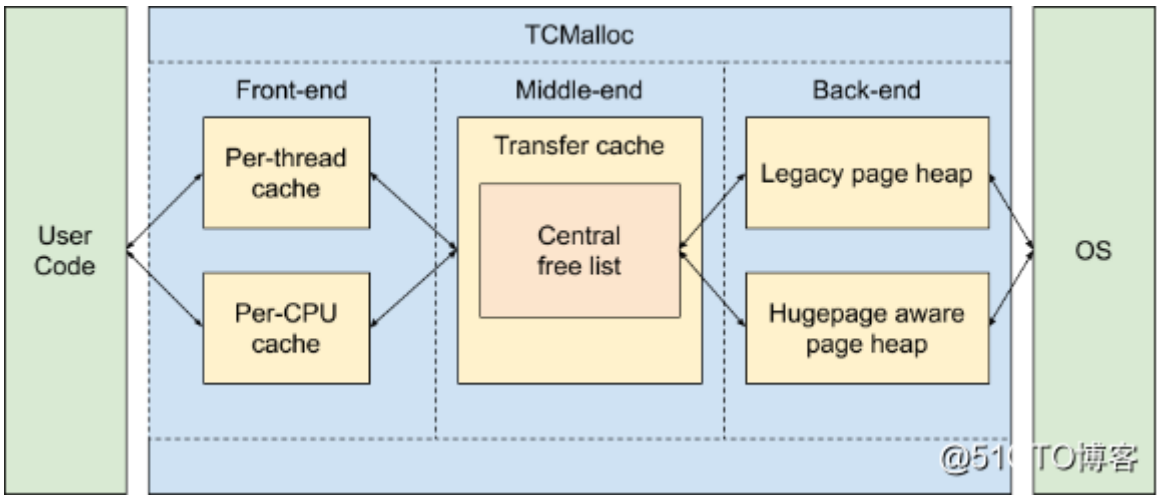
在Linux64位系统上使用libunwind只能使用TCMalloc，但heap-checker、heap-profiler和cpu-profiler不能正常使用。

如果不希望安装libunwind，也可以用gperftools内置的stack unwinder，但需要应用程序、TCMalloc库、系统库（比如libc）在编译时开启帧指针（frame pointer）选项。

在x86-64下，编译时开启帧指针选项并不是默认行为。因此需要指定-fno-omit-frame-pointer编译所有应用程序，然后在configure时通过–enable-frame-pointers选项使用内置的gperftools stack unwinder。

二、TCMalloc架构

1、TCMalloc架构



Front-end（前端）：负责提供快速分配和重分配内存给应用，由Per-thread cache和Per-CPU cache两部分组成。

Middle-end（中台）：负责给Front-end提供缓存。当Front-end缓存内存不够用时，从Middle-end申请内存。

Back-end（后端）：负责从操作系统获取内存，并给Middle-end提供缓存使用。

TCMalloc中每个线程都有独立的线程缓存ThreadCache，线程的内存分配请求会向ThreadCache申请，ThreadCache内存不够用会向CentralCache申请，CentralCache内存不够用时会向PageHeap申请，PageHeap不够用就会向OS操作系统申请。

TCMalloc将整个虚拟内存空间划分为n个同等大小的Page，将n个连续的page连接在一起组成一个Span；PageHeap向OS申请内存，申请的span可能只有一个page，也可能有n个page。

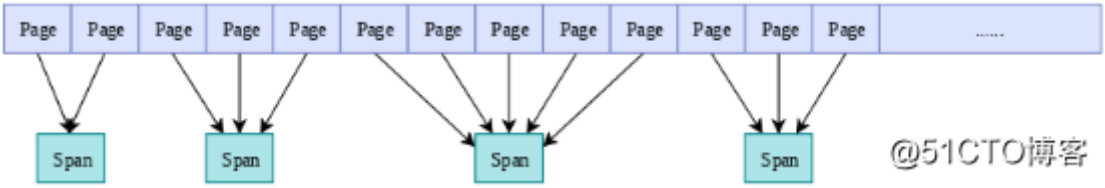
（1）Page

Page是操作系统对内存管理的单位，TCMalloc中以Page为单位管理内存，Page默认大小为8KB，通常为Linux系统中Page大小的倍数关系，如8、32、64，可以在编译选项配置时通过–with-tcmalloc-pagesize参数指定。

Page越大，TCMalloc的速度相对越快，但其占用的内存也会越高。默认Page大小通过减少内存碎片来最小化内存使用，使用更大的Page则会带来更多的内存碎片，但速度上会有所提升。

（2）Span

Span是PageHeap中管理内存Page的单位，由一个或多个连续的Page组成，比如2个Page组成的span，多个span使用链表来管理，TCMalloc以Span为单位向操作系统申请内存。



第1个span包含2个page，第2个和第4个span包含3个page，第3个span包含5个page。
Span会记录起始page的PageID（start）以及所包含page的数量（length）。

Span要么被拆分成多个相同size class的小对象用于小对象分配，要么作为一个整体用于中对象或大对象分配。当作用作小对象分配时，span的sizeclass成员变量记录了其对应的size class。

span中包含两个Span类型的指针（prev，next），用于将多个span以链表的形式存储。

Span有三种状态：IN_USE、ON_NORMAL_FREELIST、ON_RETURNED_FREELIST。

IN_USE是正在使用中，要么被拆分成小对象分配给CentralCache或者ThreadCache，要么已经分配给应用程序。

ON_NORMAL_FREELIST是空闲状态。

ON_RETURNED_FREELIST指span对应的内存已经被PageHeap释放给系统。

（3）ThreadCache

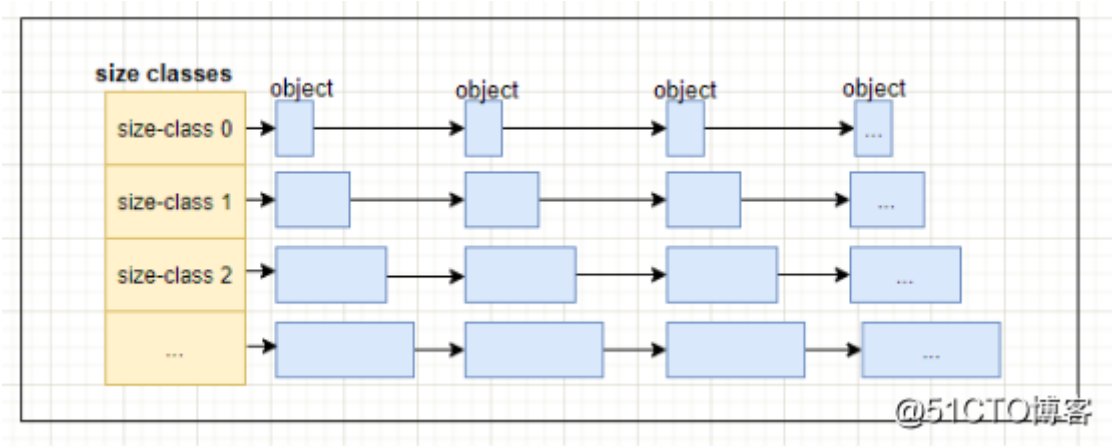
ThreadCache是每个线程独立拥有的Cache，包含多个空闲内存链表（size classes），每一个链表（size-class）都有大小相同的object。

线程可以从各自Thread Cache的FreeList获取对象，不需要加锁，所以速度很快。如果ThreadCache的FreeList为空，需要从CentralCache中的CentralFreeList中获取若干个object到ThreadCache对应的size class列表中，然后再取出其中一个object返回。

（4）Size Class

TCMalloc定义了很多个size class，每个size class都维护了一个可分配的FreeList，FreeList中的每一项称为一个object，同一个size-class的FreeList中每个object大小相同。

在申请小内存时(小于256K)，TCMalloc会根据申请内存大小映射到某个size-class中。比如，申请0到8个字节的大小时，会被映射到size-class1中，分配8个字节大小；申请9到16字节大小时，会被映射到size-class2中，分配16个字节大小，以此类推。



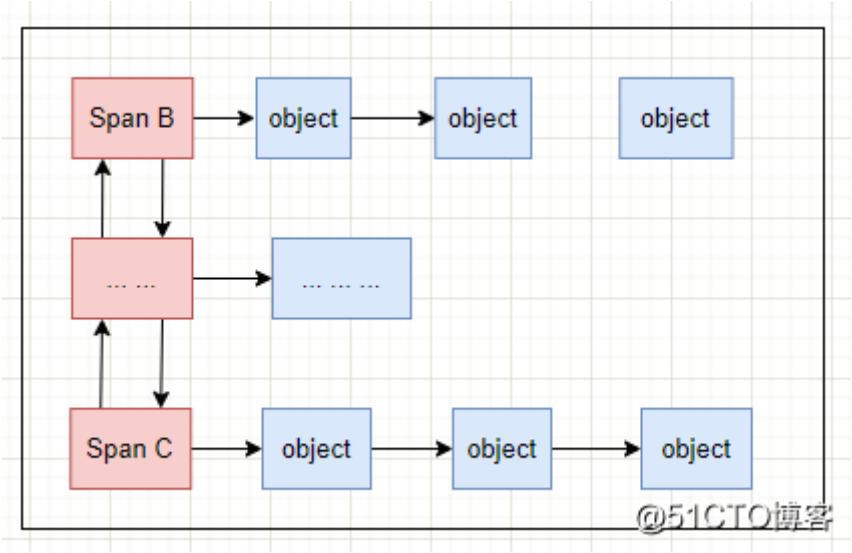
（5）CentralCache

CentralCache是ThreadCache的缓存，ThreadCache内存不足时会向CentralCache申请。

CentralCache本质是一组CentralFreeList，链表数量和ThreadCache数量相同。ThreadCache中内存过多时，可以放回CentralCache中。

如果CentralFreeList中的object不够，CentralFreeList会向PageHeap申请一连串由Span组成的Page，并将申请的Page切割成一系列的object后，再将部分object转移给ThreadCache。

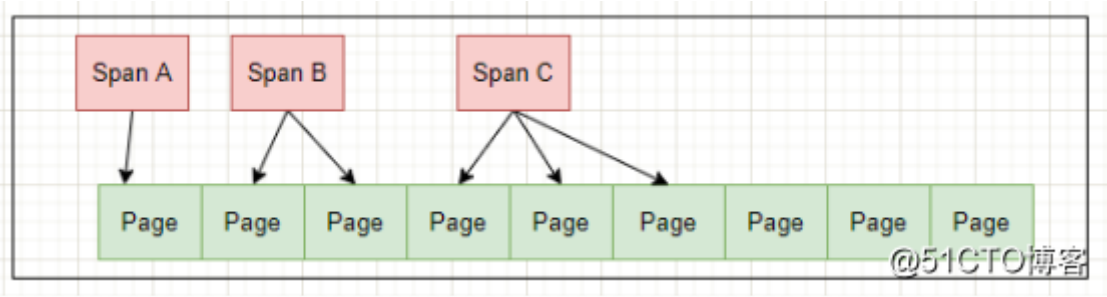
当申请的内存大于256K时，不在通过ThreadCache分配，而是通过PageHeap直接分配大内存。

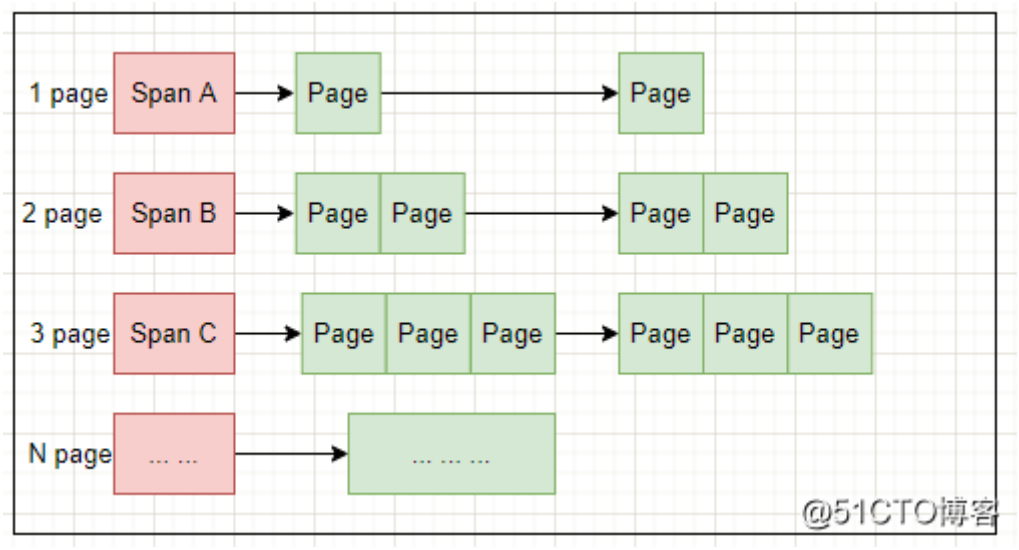


（6）PageHeap

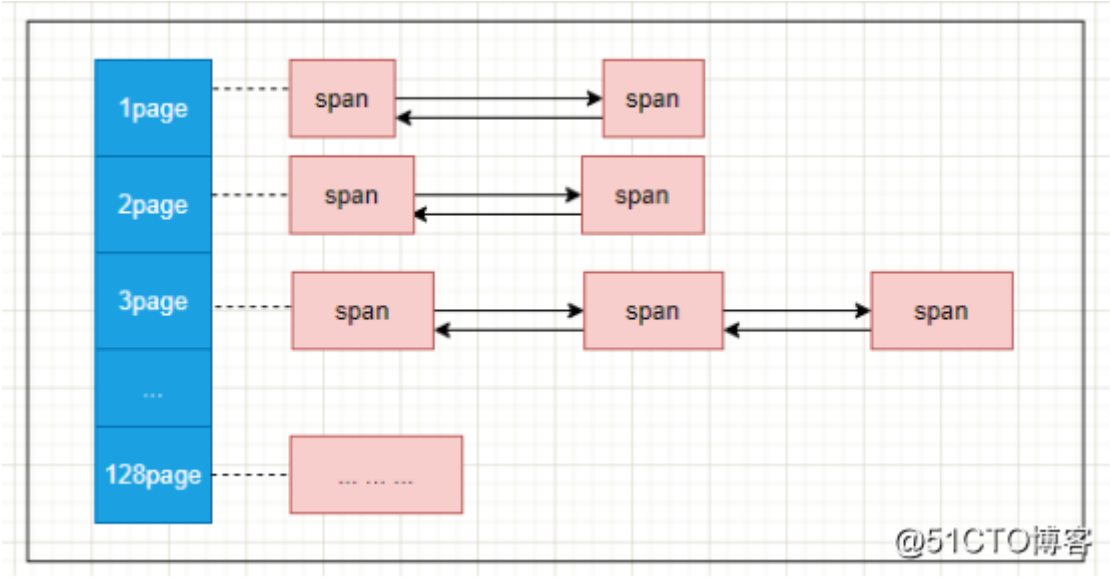
PageHeap保存存储Span的若干链表，CentralCache内存不足时，可以从PageHeap获取Span，然后把Span切割成object。

PageHeap申请内存时按照Page申请，但管理内存的基本单位是Span，Span代表若干连续Page。





PageHeap组织结构如下：



2、Front-end

Front-end处理对特定大小内存的请求，有一个内存缓存用于分配或保存空闲内存。Front-end缓存一次只能由单个线程访问，不需要任何锁，因此大多数分配和释放都很快。

只要有适当大小的缓存内存，Front-end将满足任何请求。如果特定大小的缓存为空，Front-end将从Middle-end请求一批内存来填充缓存。Middle-end包括CentralfReelList和TransferCache。

如果Middle-end内存耗尽，或者用户请求的内存大小大于Front-end缓存的最大值，则请求将转到Back-end，以满足大块内存分配，或重新填充Middle-end的缓存。Back-end也称为PageHeap。

Front-end由两种不同的实现模式：

（1）Per-thread

TCMalloc最初支持对象的Per-thread缓存，但会导致内存占用随着线程数增加而增加。现代应用程序可能有大量的线程，会导致每个线程占用内存累积起来很大，也可能会导致由单个较小线程缓存累积起来的内存占用会很大。

（2）Per-CPU

TCMalloc近期开始支持Per-CPU模式。在Per-CPU模式下，系统中的每个逻辑CPU都有自己的缓存，可以从中分配内存。在x86架构，逻辑CPU相当于一个超线程。

3、Middle-end

Middle-end负责向Front-end提供内存并将内存返回Back-end。Middle-end由Transfer cache和Central free list组成，每个类大小都有一个Transfer cache和一个Central free list。缓存由互斥锁保护，因此访问缓存会产生串行化成本。

（1）Transfer cache

当Front-end请求内存或返回内存时，将访问Transfer cache。

Transfer cache保存一个指向空闲内存的指针数组，可以快速地将对象移动到数组中，或者代表Front-end从数组中获取对象。

当一个线程正在分配另一个线程释放的内存时，Transfer cache就可以得到内存名称。Transfer cache允许内存在两个不同的线程之间快速流动。

如果Transfer cache无法满足内存请求，或者没有足够的空间容纳返回的对象，Transfer cache将访问Central free list。

（2）Central Free List

Central Free List使用spans管理内存，span是一个或多个TCMalloc内存Page的集合。

一个或多个对象的内存请求由Central Free List来满足，方法是从span中提取对象，直到满足请求为止。如果span中没有足够的可用对象，则会从Back-end请求更多的span。

当对象返回到Central Free List时，每个对象都映射到其所属的span（使用pagemap，然后释放到span中）。如果驻留在指定span中的所有对象都返回给span，则整个span将返回给Back-end。

4、Back-end

TCMalloc中Back-end有三项职责：

（1）管理大量未使用的内存块。

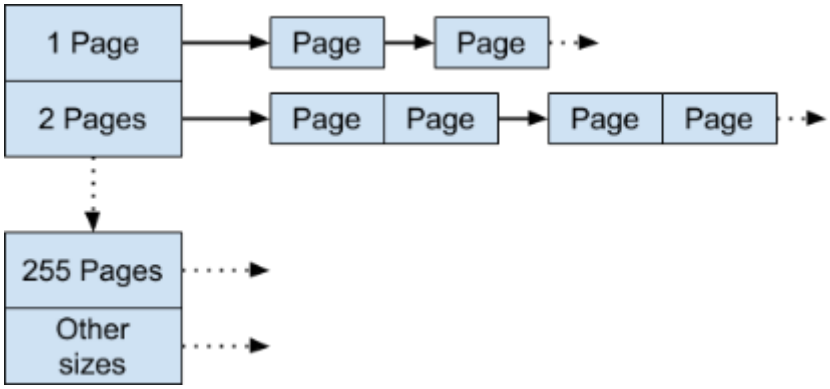
（2）负责在没有合适大小的内存来满足分配请求时从操作系统获取内存。

（3）负责将不需要的内存返回给操作系统。

TCMalloc有两种Back-end：

（1）Legacy Pageheap，管理TCMalloc中Page大小的内存块。

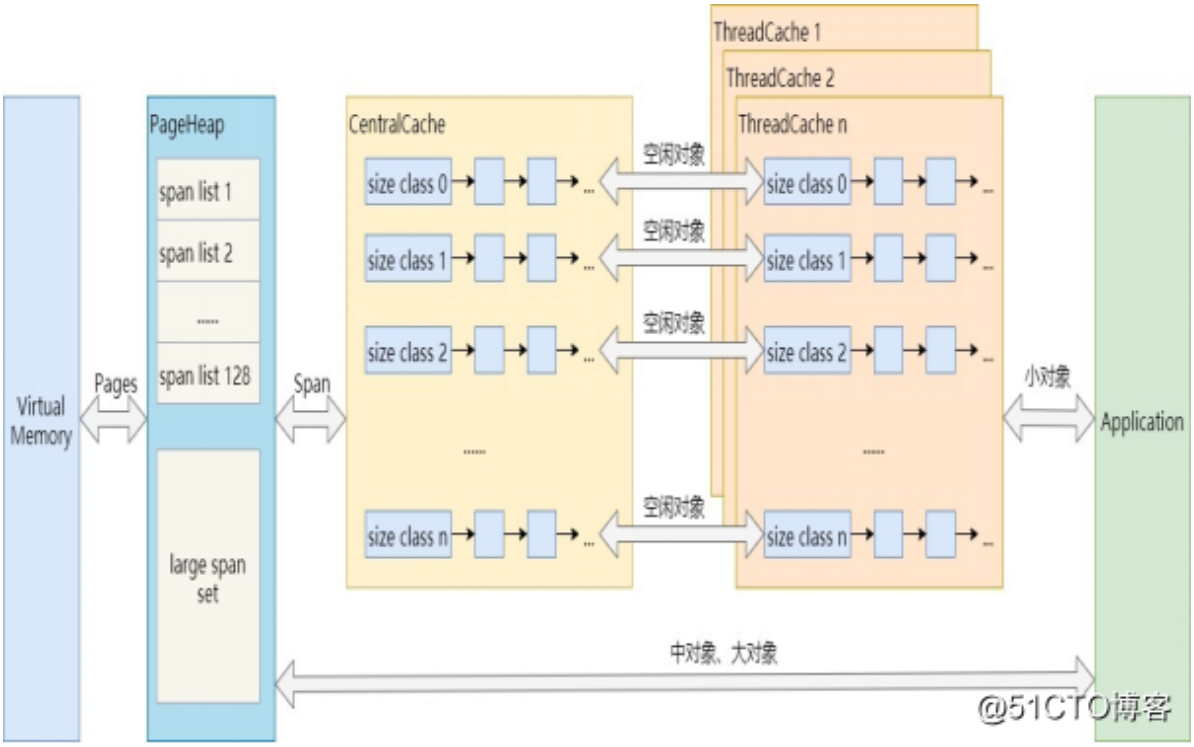
Legacy Pageheap是一个可用内存连续页面的特定长度的空闲列表数组。对于`k<256`，kth条目是由k个TCMalloc页组成的运行的免费列表。第256项是长度大于等于256页的运行的免费列表



（2）支持hugepage的pageheap，以hugepage大小的内存块来管理内存。管理hugepage内存块中内存，使分配器能够通过减少TLB未命中率来提高应用程序性能。

三、TCMalloc内存分配原理

1、TCMalloc内存分配简介



TCMalloc按照所分配内存的大小将内存分配分为三类：小对象分配(0, 256KB]、中对象分配(256KB, 1MB]、大对象分配(1MB, +∞)。

TCMalloc分配小对象分配时，在应用程序和内存之间其实有三层缓存：PageHeap、CentralCache、ThreadCache；TCMalloc分配中对象和大对象时，只有PageHeap缓存。

2、小内存分配

（1）Size Class

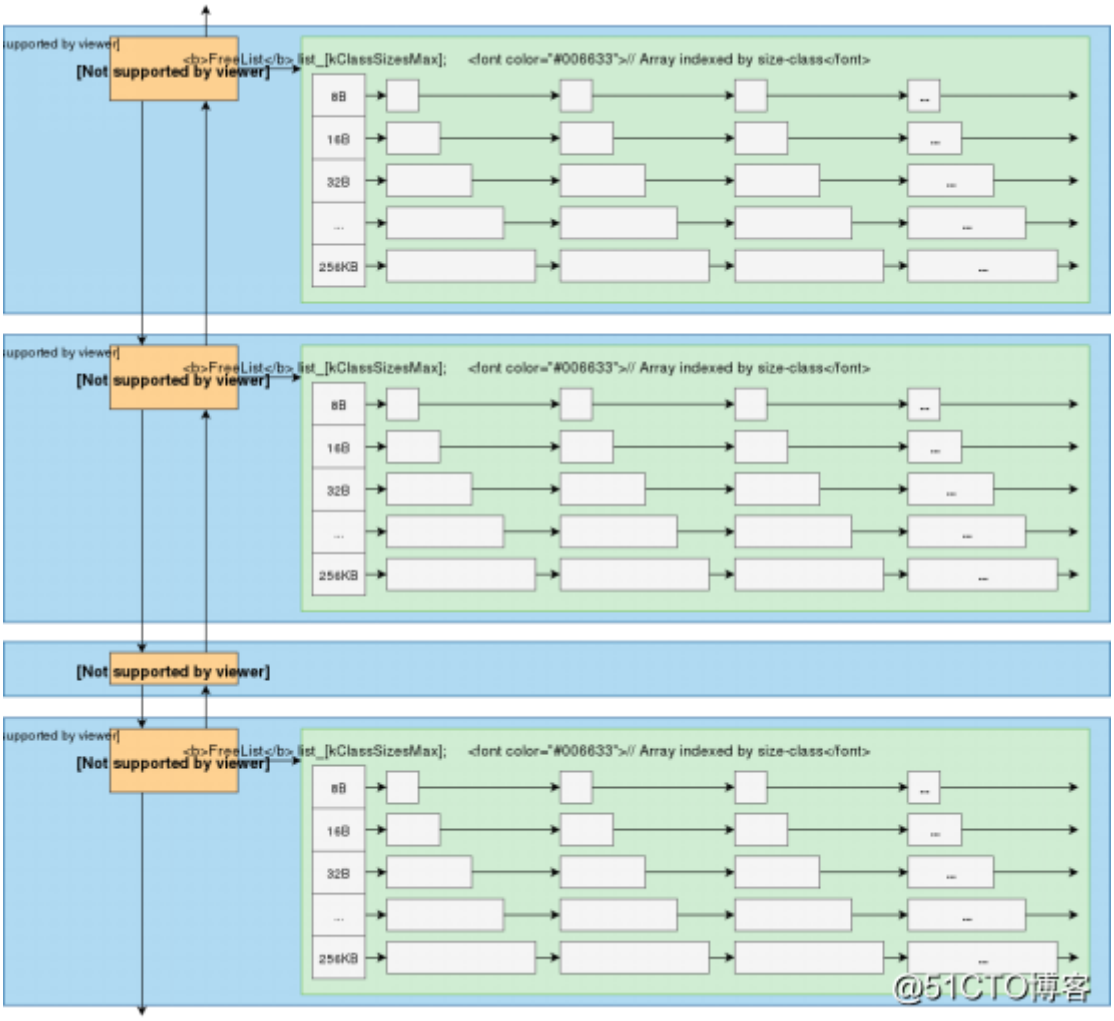
对于小于256KB的小对象，TCMalloc按大小划分85个类别（Size Class），每个size class都对应一个大小，比如8字节，16字节，32字节。应用程序申请内存时，TCMalloc会首先将所申请的内存大小向上取整到size class的大小，比如18字节之间的内存申请都会分配8字节，916字节之间都会分配16字节，以此类推。

（2）ThreadCache

TCMalloc为每个线程保存独立的线程缓存，称为ThreadCache。ThreadCache中对于每个size class都有一个独立的FreeList，缓存n个未被应用程序使用的空闲对象。

TCMalloc对于小对象的分配直接从ThreadCache的FreeList中返回一个空闲对象，小对象的回收也将其重新放回ThreadCache中对应的FreeList中。由于每个线程的ThreadCache是独立的，因此从ThreadCache中取用或回收内存是不需要加锁的，速度很快。

为了方便统计数据，各线程的ThreadCache连接成一个双向链表。ThreadCache结构如下：



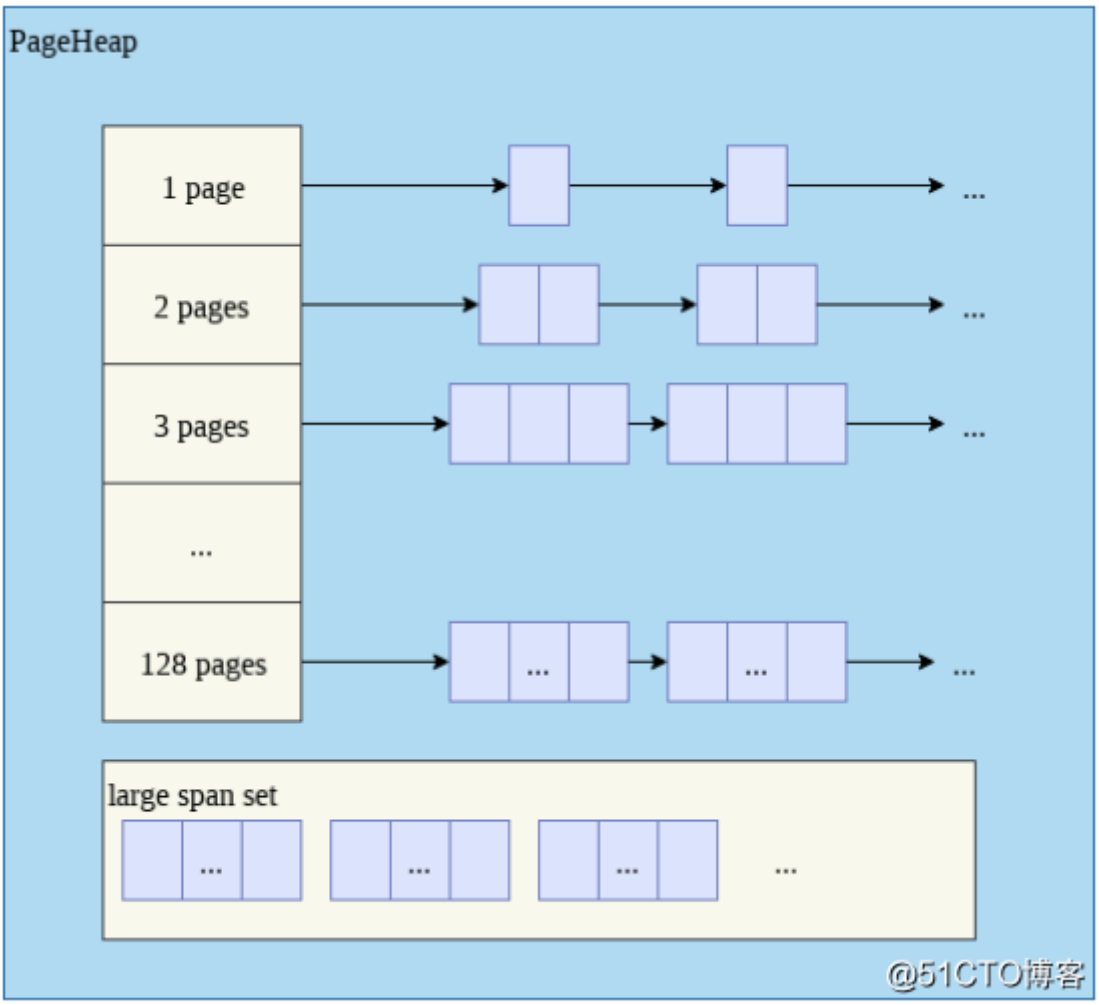
（3）CentralCache

ThreadCache中的空闲对象来自所有线程的公用缓存CentralCache。CentralCache中对于每个size class也都有一个单独的链表来缓存空闲对象，称为CentralFreeList，供各线程的ThreadCache从中获取空闲对象。线程从CentralCache中取用或回收对象，是需要加锁的。为了平摊锁操作的开销，ThreadCache一般从CentralCache中一次性取用或回收多个空闲对象。

（4）PageHeap

当CentralCache中的空闲对象不够用时，CentralCache会向PageHeap申请一块内存（可能来自PageHeap的缓存，也可能向系统申请新的内存），并将其拆分成一系列空闲对象，添加到对应size class的CentralFreeList中。

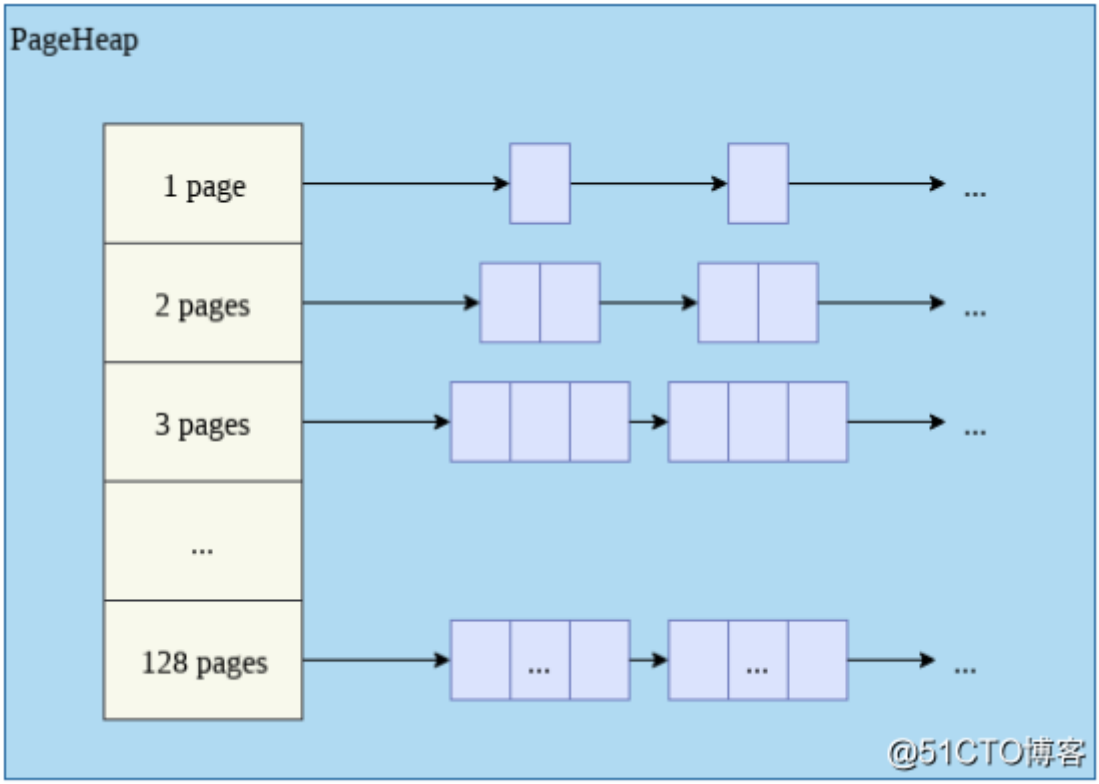
PageHeap内部根据内存块（span）的大小采取了两种不同的缓存策略。128个page以内的span，每个大小都用一个链表来缓存，超过128个page的span，存储于一个有序set（std::set）。



（5）内存回收

应用程序调用free或delete一个小对象时，仅是将其插入到ThreadCache中其size class对应的FreeList中，不需要加锁，因此速度非常快。

只有当满足一定的条件时，ThreadCache中的空闲对象才会重新放回CentralCache中，以供其它线程取用。当满足一定条件时，CentralCache中的空闲对象也会还给PageHeap，PageHeap再还给系统。



假设要分配一块内存，其大小经过向上取整后对应k个page，因此需要从PageHeap取一个大小为k个page的span。分配过程如下：

（1）首先从k个page的span链表开始，到128个page的span链表，按顺序找到第一个非空链表。

（2）取出非空链表中的一个span，假设有n个page，将span拆分成两个span：一个span大小为k个page，作为分配结果返回；另一个span大小为n – k个page，重新插入到n – k个page的span链表中。

（3）如果找不到非空链表，则将分配看做是大对象分配。

4、大内存分配

TCMalloc对于超过1MB（128个page）的大对象分配需要先将所要分配的内存大小向上取整到整数个page，假设是k个page，然后向PageHeap申请一个k个page大小的span。

大对象分配用到的span都是超过128个page的span，其缓存方式不是链表，而是一个按span大小排序的有序set（std::set），以便按大小进行搜索。

假设要分配一块超过1MB的内存，其大小经过向上取整后对应k个page（k>128），或者是要分配一块1MB以内的内存，但无法由中对象分配逻辑来满足，此时k <= 128。分配过程如下：

（1）搜索span set，找到不小于k个page的最小的span，假设span有n个page。

（2）将span拆分为两个span：一个span大小为k个page，作为结果返回；另一个span大小为n – k个page，如果n – k > 128，则将其插入到大span的set中，否则，将其插入到对应的小span链表中。

（3）如果找不到合适的span，则使用sbrk或mmap向系统申请新的内存以生成新的span，并重新执行中对象或大对象的分配算法。

四、TCMalloc使用指南

1、TCMalloc库简介

libtcmalloc_and_profiler.so
libtcmalloc_debug.so
libtcmalloc_minimal_debug.so
libtcmalloc_minimal.so
libtcmalloc.so
libtcmalloc_minimal版本不包含heap profiler和heap checker。

2、动态库方式

通过-ltcmalloc或-ltcmalloc_minimal将TCMalloc链接到应用程序。

通过LD_PRELOAD预载入TCMalloc库可以不用重新编译应用程序即可使用TCMalloc。
`LD_PRELOAD="/usr/lib/libtcmalloc.so"`

3、静态库方式

在编译选项的最后加入/usr/local/lib/libtcmalloc_minimal.a链接静态库。

4、TCMalloc生效

TCMalloc在libc_override.h中实现了覆盖机制，在使用指定-ltcmalloc链接后，应用程序对malloc、free、new、delete等调用就从默认glibc中的函数调用变为TCMalloc库中相应的函数调用。

（1）仅使用GLibc库

在glibc中，内存分配相关的函数都是弱符号（weak symbol），因此TCMalloc只需要定义自己的函数将其覆盖即可。

libc_override_redefine.h中定义如下：

```
void* operator new(size_t size) { return TCMallocInternalNew(size); }
void operator delete(void* p) noexcept { TCMallocInternalDelete(p); }
void* operator new[](size_t size) { return TCMallocInternalNewArray(size); }
void operator delete[](void* p) noexcept { TCMallocInternalDeleteArray(p); }
void* operator new(size_t size, const std::nothrow_t& nt) noexcept {
  return TCMallocInternalNewNothrow(size, nt);
}
void* operator new[](size_t size, const std::nothrow_t& nt) noexcept {
  return TCMallocInternalNewArrayNothrow(size, nt);
}
void operator delete(void* ptr, const std::nothrow_t& nt) noexcept {
  return TCMallocInternalDeleteNothrow(ptr, nt);
}
void operator delete[](void* ptr, const std::nothrow_t& nt) noexcept {
  return TCMallocInternalDeleteArrayNothrow(ptr, nt);
}
extern "C" {
    void* malloc(size_t s) noexcept { return TCMallocInternalMalloc(s); }
    void free(void* p) noexcept { TCMallocInternalFree(p); }
    void sdallocx(void* p, size_t s, int flags) {
      TCMallocInternalSdallocx(p, s, flags);
    }
    void* realloc(void* p, size_t s) noexcept {
      return TCMallocInternalRealloc(p, s);
    }
```

```
    void* calloc(size_t n, size_t s) noexcept {
      return TCMallocInternalCalloc(n, s);
    }
    void cfree(void* p) noexcept { TCMallocInternalCfree(p); }
    void* memalign(size_t a, size_t s) noexcept {
      return TCMallocInternalMemalign(a, s);
    }
    void* valloc(size_t s) noexcept { return TCMallocInternalValloc(s); }
    void* pvalloc(size_t s) noexcept { return TCMallocInternalPvalloc(s); }
    int posix_memalign(void** r, size_t a, size_t s) noexcept {
      return TCMallocInternalPosixMemalign(r, a, s);
    }
    void malloc_stats(void) noexcept { TCMallocInternalMallocStats(); }
    int mallopt(int cmd, int v) noexcept { return TCMallocInternalMallOpt(cmd,
v); }
    #ifdef HAVE_STRUCT_MALLINFO
    struct mallinfo mallinfo(void) noexcept {
      return TCMallocInternalMallocInfo();
    }
    #endif
    size_t malloc_size(void* p) noexcept { return TCMallocInternalMallocSize(p);
}
    size_t malloc_usable_size(void* p) noexcept {
      return TCMallocInternalMallocSize(p);
    }
}  // extern "C"
```

（2）使用GCC编译器编译

如果使用了GCC编译器，则使用其支持的函数属性：alias。

libc_override_gcc_and_weak.h：

```
#define TCMALLOC_ALIAS(tc_fn) \
  __attribute__((alias(#tc_fn), visibility("default")))

void* operator new(size_t size) noexcept(false)
    TCMALLOC_ALIAS(TCMallocInternalNew);
void operator delete(void* p) noexcept TCMALLOC_ALIAS(TCMallocInternalDelete);
void operator delete(void* p, size_t size) noexcept
    TCMALLOC_ALIAS(TCMallocInternalDeleteSized);
void* operator new[](size_t size) noexcept(false)
    TCMALLOC_ALIAS(TCMallocInternalNewArray);
void operator delete[](void* p) noexcept
    TCMALLOC_ALIAS(TCMallocInternalDeleteArray);
void operator delete[](void* p, size_t size) noexcept
    TCMALLOC_ALIAS(TCMallocInternalDeleteArraySized);
void* operator new(size_t size, const std::nothrow_t& nt) noexcept
    TCMALLOC_ALIAS(TCMallocInternalNewNothrow);
void* operator new[](size_t size, const std::nothrow_t& nt) noexcept
    TCMALLOC_ALIAS(TCMallocInternalNewArrayNothrow);
void operator delete(void* p, const std::nothrow_t& nt) noexcept
    TCMALLOC_ALIAS(TCMallocInternalDeleteNothrow);
void operator delete[](void* p, const std::nothrow_t& nt) noexcept
    TCMALLOC_ALIAS(TCMallocInternalDeleteArrayNothrow);

void* operator new(size_t size, std::align_val_t alignment) noexcept(false)
```

```
    TCMALLOC_ALIAS(TCMallocInternalNewAligned);
void* operator new(size_t size, std::align_val_t alignment,
                   const std::nothrow_t&) noexcept
    TCMALLOC_ALIAS(TCMallocInternalNewAligned_nothrow);
void operator delete(void* p, std::align_val_t alignment) noexcept
    TCMALLOC_ALIAS(TCMallocInternalDeleteAligned);
void operator delete(void* p, std::align_val_t alignment,
                     const std::nothrow_t&) noexcept
    TCMALLOC_ALIAS(TCMallocInternalDeleteAligned_nothrow);
void operator delete(void* p, size_t size, std::align_val_t alignment) noexcept
    TCMALLOC_ALIAS(TCMallocInternalDeleteSizedAligned);
void* operator new[](size_t size, std::align_val_t alignment) noexcept(false)
    TCMALLOC_ALIAS(TCMallocInternalNewArrayAligned);
void* operator new[](size_t size, std::align_val_t alignment,
                     const std::nothrow_t&) noexcept
    TCMALLOC_ALIAS(TCMallocInternalNewArrayAligned_nothrow);
void operator delete[](void* p, std::align_val_t alignment) noexcept
    TCMALLOC_ALIAS(TCMallocInternalDeleteArrayAligned);
void operator delete[](void* p, std::align_val_t alignment,
                       const std::nothrow_t&) noexcept
    TCMALLOC_ALIAS(TCMallocInternalDeleteArrayAligned_nothrow);
void operator delete[](void* p, size_t size,
                       std::align_val_t alignemnt) noexcept
    TCMALLOC_ALIAS(TCMallocInternalDeleteArraySizedAligned);

extern "C" {
void* malloc(size_t size) noexcept TCMALLOC_ALIAS(TCMallocInternalMalloc);
void free(void* ptr) noexcept TCMALLOC_ALIAS(TCMallocInternalFree);
void sdallocx(void* ptr, size_t size, int flags) noexcept
    TCMALLOC_ALIAS(TCMallocInternalSdallocx);
void* realloc(void* ptr, size_t size) noexcept
    TCMALLOC_ALIAS(TCMallocInternalRealloc);
void* calloc(size_t n, size_t size) noexcept
    TCMALLOC_ALIAS(TCMallocInternalCalloc);
void cfree(void* ptr) noexcept TCMALLOC_ALIAS(TCMallocInternalCfree);
void* memalign(size_t align, size_t s) noexcept
    TCMALLOC_ALIAS(TCMallocInternalMemalign);
void* aligned_alloc(size_t align, size_t s) noexcept
    TCMALLOC_ALIAS(TCMallocInternalAlignedAlloc);
void* valloc(size_t size) noexcept TCMALLOC_ALIAS(TCMallocInternalValloc);
void* pvalloc(size_t size) noexcept TCMALLOC_ALIAS(TCMallocInternalPvalloc);
int posix_memalign(void** r, size_t a, size_t s) noexcept
    TCMALLOC_ALIAS(TCMallocInternalPosixMemalign);
void malloc_stats(void) noexcept TCMALLOC_ALIAS(TCMallocInternalMallocStats);
int mallopt(int cmd, int value) noexcept
    TCMALLOC_ALIAS(TCMallocInternalMallOpt);
struct mallinfo mallinfo(void) noexcept
    TCMALLOC_ALIAS(TCMallocInternalMallocInfo);
size_t malloc_size(void* p) noexcept TCMALLOC_ALIAS(TCMallocInternalMallocSize);
size_t malloc_usable_size(void* p) noexcept
    TCMALLOC_ALIAS(TCMallocInternalMallocSize);
}  // extern "C"
```

5、TCMalloc调优

默认情况下，TCMaloc会将长时间未用的内存交还系统。tcmalloc_release_rate标识用于控制交回频率，可以在运行时强制调用ReleaseFreeMemory回收未使用内存。

`MallocExtension::instance()->ReleaseFreeMemory();`

通过 SetMemoryReleaseRate来设置tcmalloc_release_rate。如果设置为0，代表永远不交回；数字越大代表交回的频率越大，值在0-10之间，可以通过设置 TCMALLOC_RELEASE_RATE环境变量来设置。

GetMemoryReleaseRate可以查看当前释放的概率值。

TCMALLOC_SAMPLE_PARAMETER：采样时间间隔，默认值为0。

TCMALLOC_RELEASE_RATE：释放未使用内存的频率，默认值为1.0。

TCMALLOC_LARGE_ALLOC_REPORT_THRESHOLD：内存最大分配阈值，默认值为1073741824（1GB）。

TCMALLOC_MAX_TOTAL_THREAD_CACHE_BYTES：分配给线程缓冲的最大内存上限，默认值为16777216（16MB）。

6、TCMalloc测试

malloc.cpp：

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <time.h>

#define MAX_OBJECT_NUMBER       (1024)
#define MAX_MEMORY_SIZE         (1024*100)

struct BufferUnit{
   int   size;
   char* data;
};

struct BufferUnit   buffer_units[MAX_OBJECT_NUMBER];

void MallocBuffer(int buffer_size) {

for(int i=0; i<MAX_OBJECT_NUMBER; ++i)  {
    if (NULL != buffer_units[i].data)   continue;

    buffer_units[i].data = (char*)malloc(buffer_size);
    if (NULL == buffer_units[i].data)  continue;

    memset(buffer_units[i].data, 0x01, buffer_size);
    buffer_units[i].size = buffer_size;
    }
}

void FreeHalfBuffer(bool left_half_flag) {
    int half_index = MAX_OBJECT_NUMBER / 2;
    int min_index = 0;
    int max_index = MAX_OBJECT_NUMBER-1;
```

```
    if  (left_half_flag)
        max_index =  half_index;
    else
        min_index = half_index;

    for(int i=min_index; i<=max_index; ++i) {
        if (NULL == buffer_units[i].data) continue;

        free(buffer_units[i].data);
        buffer_units[i].data =  NULL;
        buffer_units[i].size = 0;
    }
}

int main() {
    memset(&buffer_units, 0x00, sizeof(buffer_units));
    int decrease_buffer_size = MAX_MEMORY_SIZE;
    bool left_half_flag   =   false;
    time_t  start_time = time(0);
    while(1)  {
        MallocBuffer(decrease_buffer_size);
        FreeHalfBuffer(left_half_flag);
        left_half_flag = !left_half_flag;
        --decrease_buffer_size;
        if (0 == decrease_buffer_size) break;
    }
    FreeHalfBuffer(left_half_flag);
    time_t end_time = time(0);
    long elapsed_time = difftime(end_time, start_time);

    printf("Used %ld seconds. \n", elapsed_time);
    return 1;
}
```

使用TCMalloc编译链接：

`g++ malloc.cpp -o test -ltcmalloc`

执行test，耗时334秒。

使用默认GLibc编译链接：

`g++ malloc.cpp -o test`

C++性能优化（十） —— JeMalloc

一、JeMalloc简介

1、JeMalloc简介

JeMalloc 是一款内存分配器，最大的优点在于多线程情况下的高性能以及内存碎片的减少。

GitHub地址：

https://github.com/jemalloc/jemalloc

2、JeMalloc安装

JeMalloc源码下载：

```
git clone https://github.com/jemalloc/jemalloc.git
```

构建工具生成：

```
autogen.sh
```

编译选项配置：

```
configure
```

编译：

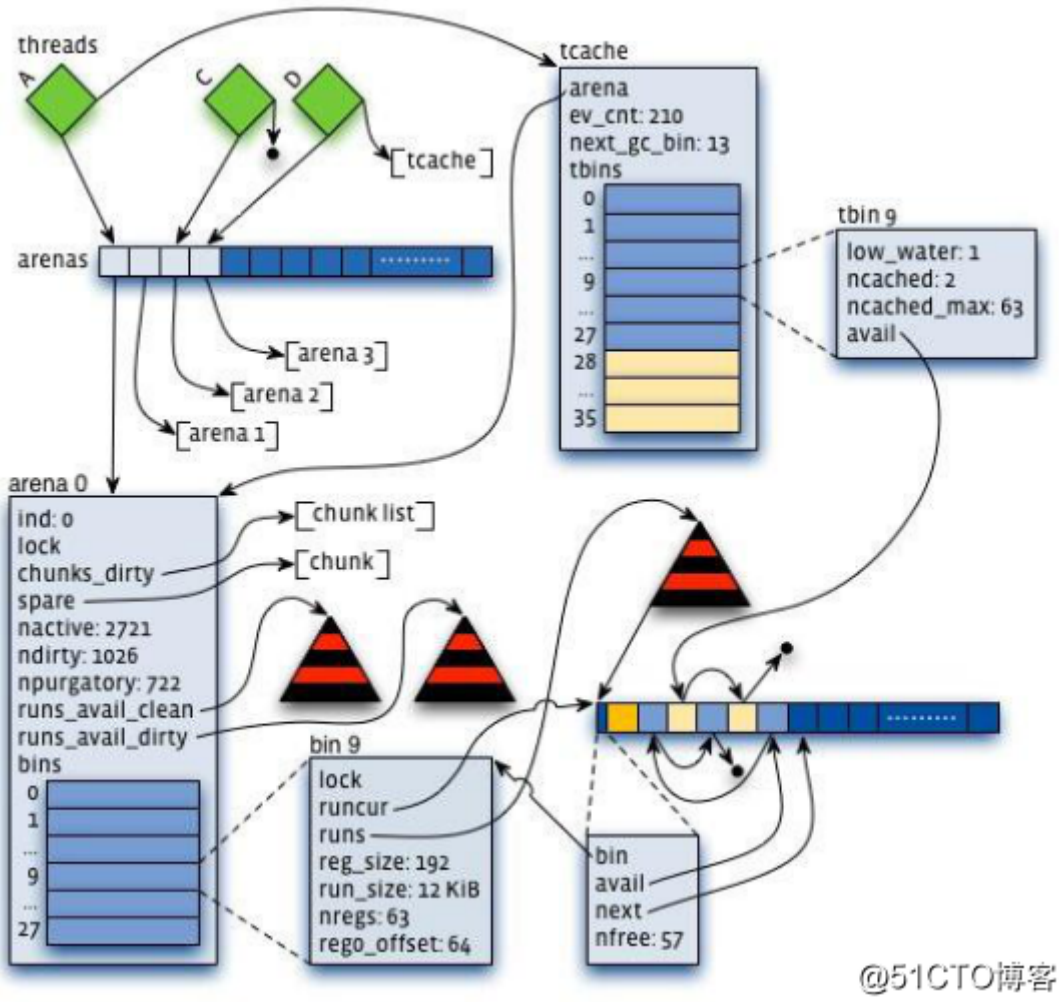
```
make -j4
```

安装：

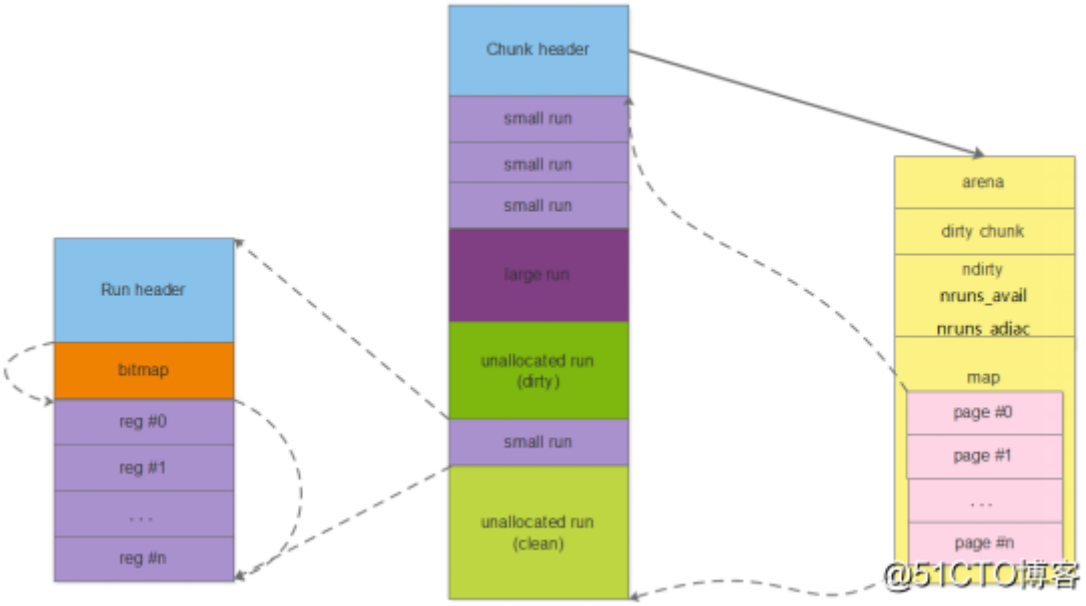
```
make install
```

二、JeMalloc架构

1、JeMalloc架构简介



JeMalloc将内存分成多个相同大小的chunk，数据存储在chunks中；每个chunk分为多个run，run负责请求、分配相应大小的内存并记录空闲和使用的regions的大小。



2、Arena

Arena是JeMalloc的核心分配管理区域，对于多核系统，会默认分配4x逻辑CPU的Arena，线程采取轮询的方式来选择相应的Arena来进行内存分配。
每个arena内都会包含对应的管理信息，记录arena的分配情况。arena都有专属的chunks, 每个chunk的头部都记录chunk的分配信息。在使用某一个chunk的时候，会把chunk分割成多个run，并记录到bin中。不同size class的run属于不同的bin，bin内部使用红黑树来维护空闲的run，run内部使用bitmap来记录分配状态。
JeMalloc使用Buddy allocation 和 Slab allocation 组合作为内存分配算法，使用Buddy allocation将Chunk划分为不同大小的 run，使用 Slab allocation 将run划分为固定大小的 region，大部分内存分配直接查找对应的 run，从中分配空闲的 region，释放则标记region为空闲。
run被释放后会和空闲的、相邻的run进行合并；当合并为整个 chunk 时，若发现有相邻的空闲 chunk，也会进行合并。

3、Chunk

Chunk是JeMalloc进行内存分配的单位，默认大小4MB。Chunk以Page（默认为4KB)为单位进行管理，每个Chunk的前6个Page用于存储后面其它Page的状态，比如是否待分配还是已经分配；而后面其它Page则用于进行实际的分配。

4、Bin

JeMalloc 中 small size classes 使用 slab 算法分配，会有多种不同大小的run，相同大小的run由bin 进行管理。
run是分配的执行者, 而分配的调度者是bin，bin负责记录当前arena中某一个size class范围内所有non-full run的使用情况。当有分配请求时，arena查找相应size class的bin，找出可用于分配的run，再由run分配region。由于只有small region分配需要run，因此bin也只对应small size class。
在arena中， 不同bin管理不同size大小的run，在任意时刻， bin中会针对当前size保存一个run用于内存分配。

5、Run

Run是chunk的一块内存区域，大小是Page的整数倍，由bin进行管理，比如8字节的bin对应的run就只有1个page，可以从里面选取一个8字节的块进行分配。
small classes 从 run 中使用 slab 算法分配，每个 run 对应一块连续的内存，大小为 page size 倍数，划分为相同大小的 region，分配时从run 中分配一个空闲 region，释放时标记region为空闲，重复使

用。
run中采用bitmap记录分配区域的状态，bitmap能够快速计算出第一块空闲区域，且能很好的保证已分配区域的紧凑型。

6、TCache

TCache是线程的私有缓存空间，在分配内存时首先从tcache中分配，避免加锁；当TCache没有空闲空间时才会进入一般的分配流程。
每个TCache内部有一个arena，arena内部包含tbin数组来缓存不同大小的内存块，但没有run。

三、JeMalloc内存分配

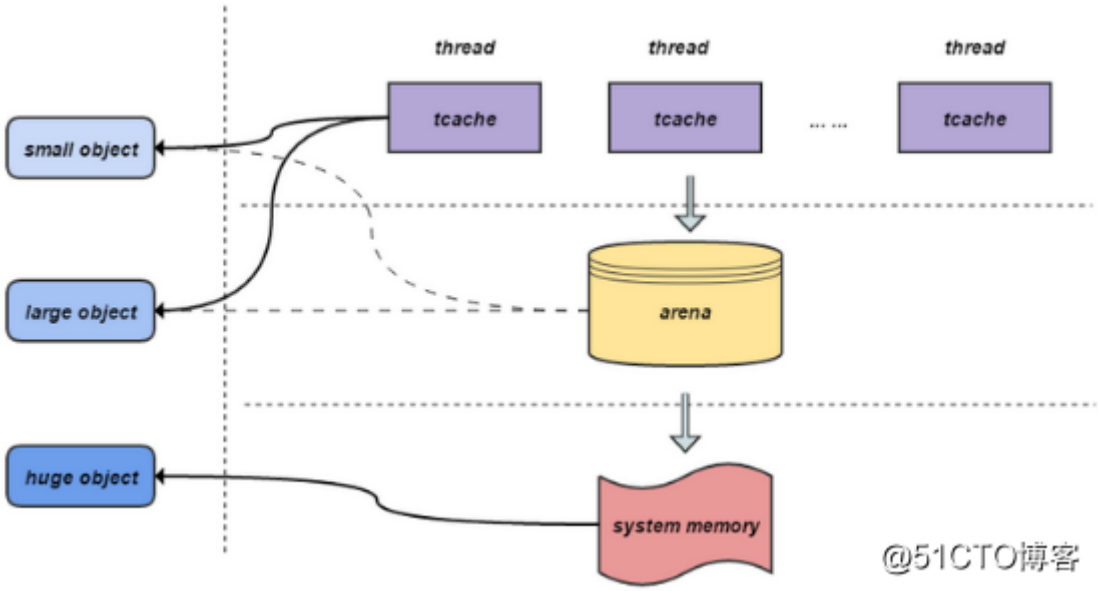
1、JeMalloc内存分配

JeMalloc基于申请内存的大小把内存分配分为三个等级：small、large、huge。
Small objects的size以8字节、16字节、32字节等分隔开的，小于Page大小。
Large objects的size以Page为单位， 等差间隔排列，小于chunk（4MB）的大小。
Huge objects的大小是chunk大小的整数倍。

Category	Spacing	Size
Small	8	[8]
	16	[16, 32, 48, …, 128]
	32	[160, 192, 224, 256]
	64	[320, 384, 448, 512]
	128	[640, 768, 896, 1024]
	256	[1280, 1536, 1792, 2048]
	512	[2560, 3072, 3584]
Large	4 KiB	[4 KiB, 8 KiB, 12 KiB, …, 4072 KiB]
Huge	4 MiB	[4 MiB, 8 MiB, 12 MiB,

JeMalloc通过将内存划分成大小相同的chunk进行管理，chunk的大小为2的k次方，大于Page大小。Chunk起始地址与chunk大小的整数倍对齐，可以通过指针操作在常量时间内找到分配small/large objects的元数据，在对数时间内定位到分配huge objects的元数据。为了获得更好的线程扩展性，JeMalloc采用多个arenas来管理内存，减少了多线程间的锁竞争。每个线程独立管理自己的内存arena，负责small和large的内存分配，线程按第一次分配small或者large内存请求的顺序Round-Robin地选择arena。从某个arena分配的内存块，在释放时一定会回到原arena。

JeMalloc引入线程缓存来解决线程间的同步问题，通过对small和large对象的缓存，实现通常情况下内存的快速申请和释放。



2、small内存分配

如果请求内存size不大于arena的最小的bin，那么通过线程对应的tcache来进行分配。
small objects分配流程如下：

（1）查找对应 size classes 的 bin

（2）从 bin 中获取未满的run。

（3）从 arena 中获取空闲run。

（4）从 run 中返回一个空闲 region。

3、large内存分配

如果请求内存size大于arena的最小的bin，同时不大于tcache能缓存的最大块，也会通过线程对应的tcache来进行分配，但方式不同。

如果tcache对应的tbin里有缓存块，直接分配；如果没有，从chunk里直接找一块相应的page整数倍大小的空间进行分配；

4、Huge内存分配

如果请求分配内存大于chunk（4MB）大小，直接通过mmap进行分配。

四、多线程支持

1、JeMalloc多线程支持

JeMalloc对于多线程内存分配与单线程相同，每个线程从 Arena 中分配内存，但多线程间需要同步和竞争，因此提高多线程内存分配性能方法如下：

（1）减少锁竞争。缩小临界区，使用更细粒度锁。

（2）避免锁竞争。线程间不共享数据，使用局部变量、线程特有数据(tsd)、线程局部存储(tls)等。

2、Arena选择

JeMalloc会创建多个Arena，每个线程由一个Arena 负责。JeMalloc默认创建4x逻辑CPU个Arena。

arena->nthreads 记录负责的线程数量。

每个线程分配时会首先调用arena_choose选择一个arena来负责线程的内存分配。线程选择 arena 的逻辑如下:

（1）如果有空闲的(nthreads==0)已创建arena，则选择空闲arena。

（2）若还有未创建的arena，则选择新创建一个arena。

（3）选择负载最低的arena (nthreads 最小)。

3、线程锁

线程锁尽量使用 spinlock，减少线程间的上下文切换。Linux操作系统可以在编译时通过定义JEMALLOC_OSSPIN宏可以指定使用自选锁。

为了缩小临界区，arena 中提供多个细粒度锁管理不同部分：

（1）arenas_lock: arena 的初始化、分配等

（2）arena->lock: run 和 chunk 的管理

（3）arena->huge_mtx: huge object 的管理

（4）bin->lock: bin 中的操作

4、tsd

当选择完arena后，会将arena绑定到tsd中，直接从tsd中获取arena。

tsd用于保存每个线程本地数据，主要arena和tcache，避免锁竞争。tsd_t中的数据会在第一次访问时延迟初始化，tsd 中各元素使用宏生成对应的 get/set 函数来获取/设置，在线程退出时，会调用相应的 cleanup 函数清理。

5、tcache

tcache 用于 small object和 large object的分配，避免多线程同步。

tcache 使用slab内存分配算法分配内存：

（1）tcache中有多种bin，每个bin管理一个size class。

（2）当分配时，从对应bin中返回一个cache slot。

（3）当释放时，将cache slot返回给对应的bin。

6、线程退出

线程退出时，会调用 tsd_cleanup() 对 tsd 中数据进行清理：

（1）arena，降低arena负载(arena->nthreads–)

（2）tcache，调用tcache_bin_flush_small/large释放 tcache->tbins[]所有元素，释放tcache。
当从一个线程分配的内存由另一个线程释放时，内存还是由原先arena来管理，通过chunk的extent_node_t来获取对应的arena。

五、JeMalloc使用指南

1、JeMalloc库简介

JeMalloc提供了静态库libjemalloc.a和动态库libjemalloc.so，默认安装在/usr/local/lib目录。

2、JeMalloc动态方式

通过-ljemalloc将JeMalloc链接到应用程序。
通过LD_PRELOAD预载入JeMalloc库可以不用重新编译应用程序即可使用JeMalloc。
LD_PRELOAD="/usr/lib/libjemalloc.so"

3、JeMalloc静态方式

在编译选项的最后加入/usr/local/lib/libjemalloc.a链接静态库。

4、JeMalloc生效

jemalloc利用malloc的hook来对代码中的malloc进行替换。

```
JEMALLOC_EXPORT void (*__free_hook)(void *ptr) = je_free;
JEMALLOC_EXPORT void *(*__malloc_hook)(size_t size) = je_malloc;
JEMALLOC_EXPORT void *(*__realloc_hook)(void *ptr, size_t size) = je_realloc;
```

5、JeMalloc测试

malloc.cpp：

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <time.h>

#define MAX_OBJECT_NUMBER       (1024)
#define MAX_MEMORY_SIZE         (1024*100)

struct BufferUnit{
   int   size;
   char* data;
};

struct BufferUnit   buffer_units[MAX_OBJECT_NUMBER];

void MallocBuffer(int buffer_size) {

for(int i=0; i<MAX_OBJECT_NUMBER; ++i)  {
    if (NULL != buffer_units[i].data)   continue;

    buffer_units[i].data = (char*)malloc(buffer_size);
    if (NULL == buffer_units[i].data)  continue;

    memset(buffer_units[i].data, 0x01, buffer_size);
    buffer_units[i].size = buffer_size;
    }
}

void FreeHalfBuffer(bool left_half_flag) {
```

```
    int half_index = MAX_OBJECT_NUMBER / 2;
    int min_index = 0;
    int max_index = MAX_OBJECT_NUMBER-1;
    if  (left_half_flag)
        max_index =  half_index;
    else
        min_index = half_index;

    for(int i=min_index; i<=max_index; ++i) {
        if (NULL == buffer_units[i].data) continue;

        free(buffer_units[i].data);
        buffer_units[i].data =  NULL;
        buffer_units[i].size = 0;
    }
}

int main() {
    memset(&buffer_units, 0x00, sizeof(buffer_units));
    int decrease_buffer_size = MAX_MEMORY_SIZE;
    bool left_half_flag   =   false;
    time_t  start_time = time(0);
    while(1)  {
        MallocBuffer(decrease_buffer_size);
        FreeHalfBuffer(left_half_flag);
        left_half_flag = !left_half_flag;
        --decrease_buffer_size;
        if (0 == decrease_buffer_size) break;
    }
    FreeHalfBuffer(left_half_flag);
    time_t end_time = time(0);
    long elapsed_time = difftime(end_time, start_time);

    printf("Used %ld seconds. \n", elapsed_time);
    return 1;
}
```

使用TCMalloc编译链接：

`g++ malloc.cpp -o test -ljemalloc`

执行test，耗时558秒。

使用默认GLibc编译链接：

`g++ malloc.cpp -o test`

执行test，耗时744秒。

C++性能优化（十一） —— 内存管理器性能分析

一、PTMalloc2

1、PTMalloc2优点

（1）集成在glibc中，Linux主要发行版的通用实现。

2、PTMalloc2缺点

（1）后分配的内存先释放。由于ptmalloc2收缩内存是从top chunk开始，如果与top chunk相邻的chunk不能释放，top chunk 以下的chunk都无法释放。
（2）多线程锁开销大，需要避免多线程频繁分配释放。
（3）内存从thread的arena中分配，不能从一个arena移动到另一个arena。如果多线程使用内存不均衡，容易导致内存的浪费。
（4）每个chunk至少8字节的开销很大。
（5）不定期分配长生命周期的内存容易造成内存碎片，不利于回收。

二、TCMalloc

1、TCMalloc优点

（1）小内存在线程ThreadCache中分配，不加锁(加锁代价大约100ns)。
（2）大内存直接按照大小调用mmap系统调用分配。
（3）大内存加锁使用更高效的自旋锁。
（4）减少内存碎片。

2、TCMalloc缺点

（1）使用大内存频繁时，内存在Central Cache或者Page Heap加锁分配。
（2）TCMalloc对大量小内存的分配过于保守，对于内存需求较大的服务（如推荐系统），小内存上限过低。如果请求量过多，锁冲突严重，CPU使用率将指数暴增。

3、TCMalloc使用场景

当线程数量不定时，使用TCMalloc。

三、JeMalloc

1、JeMalloc优点

（1）采用多个arena来避免线程同步。
（2）使用细粒度锁，大大减少加锁。
（3）TCache GC时对缓存容量进行动态调整。

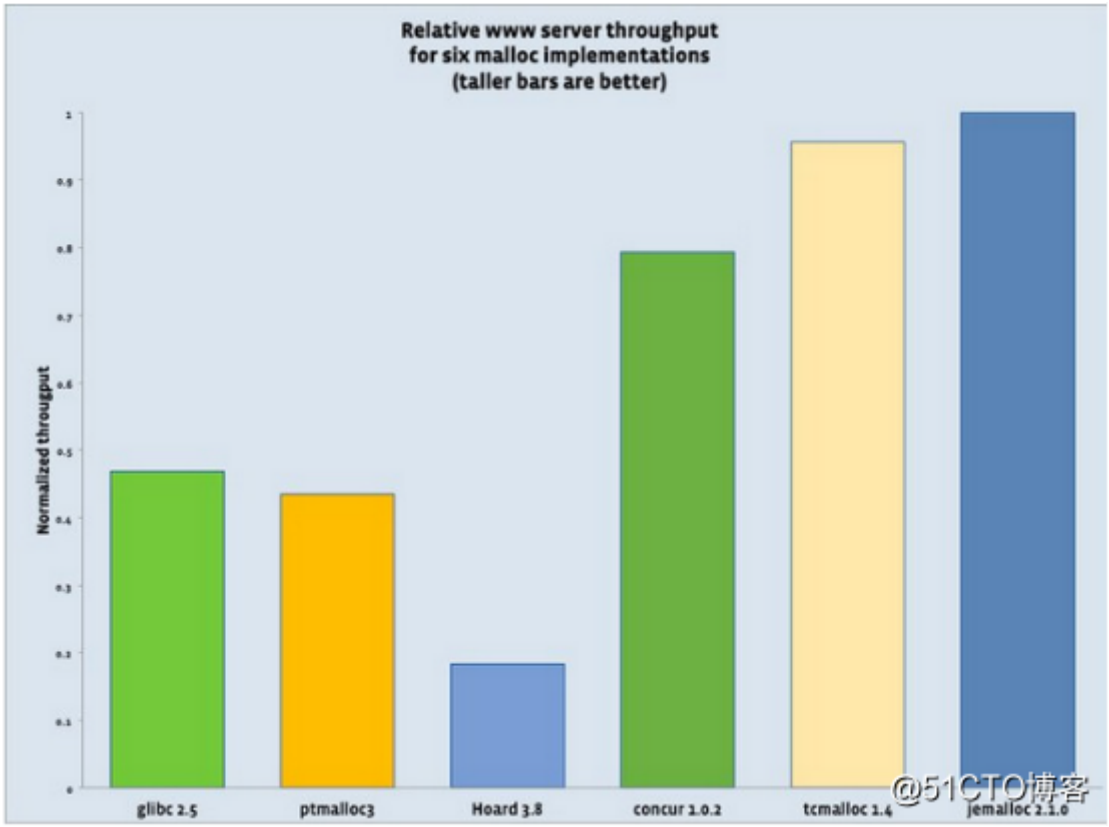
2、JeMalloc缺点

JeMalloc内存分配器的缺点在于arena 间的内存不可见。
（1）某个线程在arena使用了大量内存，但arena 并没有其它线程使用，导致arena 内存无法被回收，占用过多。
（2）两个位于不同arena的线程频繁进行内存申请，导致两个arena的内存出现大量交叉，但连续的内存由于在不同arena而无法进行合并。

3、JeMalloc使用场景

当线程数量固定，不会频繁创建退出时，可以使用jemalloc。

四、不同内存管理器性能比较



C++性能优化（十二）——自旋锁

一、互斥锁

1、互斥锁简介

互斥锁属于sleep-waiting类型锁。Linux Kernel 2.6.x稳定版开始，Linux的互斥锁都是futex (Fast Usermode Mutex)锁。

Futex是一个在Linux上实现锁定和构建高级抽象锁如信号量和POSIX互斥的基本工具。

Futex由Hubertus Franke（IBM Thomas J. Watson 研究中心），Matthew Kirkwood，Ingo Molnar（Red Hat）和 Rusty Russell（IBM Linux 技术中心）等人创建。

Futex是由用户空间的一个对齐的整型变量和附在其上的内核空间等待队列构成。多进程或多线程绝大多数情况下对位于用户空间的futex的整型变量进行操作(汇编语言调用CPU提供的原子操作指令来增加或减少)，而其它情况下则需要通过代价较大的系统调用来对位于内核空间的等待队列进行操作(如唤醒等待的进程/线程或将当前进程/线程放入等待队列)。除了多个线程同时竞争锁的少数情况外，基于futex的lock操作是不需要进行代价昂贵的系统调用操作的。

Futex核心思想是通过将大多数情况下非同时竞争lock的操作放到在用户空间执行，而不是代价昂贵的内核系统调用方式来执行，从而提高了效率。

互斥锁禁止多个线程同时进入受保护的代码临界区（critical section）。在任意时刻，只有一个线程被允许进入代码保护区。互斥锁实际上是count=1情况下的semaphore。

2、互斥锁特点

互斥锁缺点：

（1）等待互斥锁会消耗时间，等待延迟会损害系统的可伸缩性。

（2）优先级倒置。低优先级的线程可以获得互斥锁，因此会阻碍需要同一互斥锁的高优先级线程。

（3）锁护送（lock convoying）。如果持有互斥锁的线程分配的时间片结束，线程被取消调度，则等待同一互斥锁的其它线程需要等待更长时间。

3、互斥锁API

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const
pthread_mutexattr_t *restrict attr);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex, const struct
timespec *restrict abs_timeout);
```

二、自旋锁

1、自旋锁简介

自旋锁（spin lock）属于busy-waiting类型锁。在多处理器环境中，自旋锁最多只能被一个可执行线程持有。如果一个可执行线程试图获得一个被其它线程持有的自旋锁，那么线程就会一直进行忙等待，自旋（空转），等待自旋锁重新可用。如果自旋锁未被争用，请求锁的执行线程便立刻得到自旋锁，继续执行。

多处理器操作系统中某些资源是有限的，不同线程需要互斥访问，因此需要引入锁概念，只有获取锁的线程才能够对资源进行访问。多线程的核心是CPU的时间分片，同一时刻只能有一个线程获取到锁。对于没有获取到锁的线程通常有两种处理方式：自旋锁，没有获取到锁的线程会一直循环等待判断资源是否已经释放锁，不用将线程阻塞起来；互斥锁，把未获取到锁的线程阻塞起来，等待重新调度请求。

自旋锁（spin lock）是指当一个线程在获取锁的时候，如果锁已经被其它线程获取，那么线程将循环等待，然后不断的判断锁是否能够被成功获取，直到获取到锁才会退出循环。

获取锁的线程一直处于活跃状态，但并没有执行任何有效的任务，使用自旋锁会造成busy-waiting。

2、自旋锁的特点

自旋锁不会使线程状态发生切换，一直处于用户态，即线程一直都是active的；不会使线程进入阻塞状态，减少了不必要的上下文切换，执行速度快。

非自旋锁在获取不到锁的时候会进入阻塞状态，从而进入内核态，当获取到锁时需要从内核态恢复，导致线程在用户态与内核态之间来回切换，严重影响锁的性能。

3、自旋锁原理

自旋锁的原理比较简单，如果持有锁的线程能在短时间内释放锁资源，那么等待竞争锁的线程就不需要做内核态和用户态之间的切换进入阻塞状态，只需要等一等(自旋)，等到持有锁的线程释放锁后即可获取，避免用户进程和内核切换的消耗。

自旋锁避免了操作系统进程调度和线程切换，通常适用在时间极短的情况，因此操作系统的内核经常使用自旋锁。但如果长时间上锁，自旋锁会非常耗费性能。线程持有锁时间越长，则持有锁的线程被 OS调度程序中断的风险越大。如果发生中断情况，那么其它线程将保持旋转状态(反复尝试获取锁)，而持有锁的线程并不打算释放锁，导致结果是无限期推迟，直到持有锁的线程可以完成并释放它为止。

自旋锁的目的是占着CPU资源不进行释放，等到获取锁立即进行处理。如果自旋执行时间太长，会有大量的线程处于自旋状态占用CPU资源，进而会影响整体系统的性能，因此可以给自旋锁设定一个自旋时间，等时间一到立即释放自旋锁。

4、自旋锁API

```
#include <pthread.h>

int pthread_spin_destroy(pthread_spinlock_t *lock);
int pthread_spin_init(pthread_spinlock_t *lock, int pshared);

int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t *lock);
```

5、自旋锁与互斥锁

spinlock不会使线程状态发生切换，mutex在获取不到锁的时候会选择sleep。

spinlock优点：没有耗时的系统调用，一直处于用户态，执行速度快。

spinlock缺点：一直占用CPU，而且在执行过程中还会锁bus总线，锁总线时其它处理器不能使用总线。

mutex获取锁分为两阶段，第一阶段在用户态采用spinlock锁总线的方式获取一次锁，如果成功立即返回；否则进入第二阶段，调用系统的futex锁去sleep，当锁可用后被唤醒，继续竞争锁。

mutex优点：不会忙等，得不到锁会sleep。

mutex缺点：sleep时会陷入到内核态，需要昂贵的系统调用。

三、自旋锁实现

1、raw_spinlock

当某个处理器上的内核执行线程申请自旋锁时，如果锁可用，则获得锁，然后执行临界区操作，最后释放锁；如果锁已被占用，线程并不会转入睡眠状态，而是忙等待该锁，一旦锁被释放，则第一个感知此信息的线程将获得锁。

```
typedef struct {
    unsigned int slock;
} raw_spinlock_t;
```

传统自旋锁本质是用一个整数来表示，值为1代表锁未被占用，为0或者为负数表示被占用。

在单处理机环境中可以使用特定的原子级汇编指令swap和test_and_set实现进程互斥，但由于中断只能发生在两条机器指令之间，而同一指令内的多个指令周期不可中断，从而保证swap指令或test_and_set指令的执行不会交叉进行。

多处理器环境中利用test_and_set指令实现进程互斥，硬件需要提供进一步的支持，以保证test_and_set指令执行的原子性，目前多以锁总线形式提供，由于test_and_set指令对内存的两次操作都需要经过总线，在执行test_and_set指令前锁住总线，在执行test_and_set指令后释放总线，即可保证test_and_set指令执行的原子性。

```
static inline void __raw_spin_lock(raw_spinlock_t *lock)
{
    __asm__ __volatile__(
        __raw_spin_lock_string
        :"=m" (lock->slock) : : "memory");
}
static inline void __raw_spin_unlock(raw_spinlock_t *lock)
{
    __asm__ __volatile__(
        __raw_spin_unlock_string
    );
}
```

由于传统自旋锁无序竞争的本质特点，内核执行线程无法保证何时可以取到锁，某些执行线程可能需要等待很长时间，导致锁竞争不公平。
（1）随着处理器个数增加，自旋锁竞争也在加剧，自然导致更长等待时间。释放自旋锁时的重置操作将无效化所有其它正在忙等待的处理器的缓存，那么在处理器拓扑结构中临近自旋锁拥有者的处理器可能会更快地刷新缓存，因而增大获得自旋锁的机率。
（2）由于每个申请自旋锁的处理器均在全局变量slock上忙等待，系统总线将因为处理器间的缓存同步而导致繁重的流量，从而降低了系统整体性能。

2、ticket spinlock

Linux Kernel 2.6.25版本中引入了排队自旋锁，通过保存执行线程申请锁的顺序信息来解决不公平问题。

排队自旋锁仍然使用raw_spinlock_t 数据结构，但是赋予slock字段新含义。为了保存顺序信息，slock字段被分成两部分Owner和Next，分别保存锁持有者和未来锁申请者的票据序号(Ticket Number)，只有Owner和Next相等时，才表明锁处于未使用状态。

排队自旋锁初始化时slock被置为0，即Owner和Next置为0。Linux内核执行线程申请自旋锁时，原子地将Next加1，并将原值返回作为自己的票据序号。如果返回的票据序号等于申请时Owner值，说明自旋锁处于未使用状态，则直接获得锁；否则，线程忙等待检查Owner是否等于自己持有的票据序号，一旦相等，则表明锁轮到自己获取。线程释放锁时，原子地将Owner加1即可，下一个线程将会发现这一变化，从忙等待状态中退出。线程将严格地按照申请顺序依次获取排队自旋锁，从而完全解决了不公平问题。

```
typedef struct arch_spinlock {
    union {
        __ticketpair_t head_tail;
        struct __raw_tickets {
            __ticket_t head, tail;
        } tickets;
    };
} arch_spinlock_t;
```

申请自旋锁时，原子地将tail加1，释放时，head加1。只有head域和tail域的值相等时，才表明锁处于未使用的状态。

```
static inline void __raw_spin_lock(raw_spinlock_t *lock)
{
    asm volatile("\n1:\t"
             LOCK_PREFIX " ; decb %0\n\t"
             "jns 3f\n"
             "2:\t"
             "rep;nop\n\t"
             "cmpb $0,%0\n\t"
             "jle 2b\n\t"
             "jmp 1b\n"
             "3:\n\t"
             : "+m" (lock->slock) : : "memory");
}
static inline void __raw_spin_unlock(raw_spinlock_t *lock)
{
    asm volatile("movb $1,%0" : "+m" (lock->slock) :: "memory");
}
```

在大规模多处理器系统和NUM系统中，排队自旋锁（包括传统自旋锁）存在一个比较严重的性能问题：由于执行线程均在同一个共享变量slock上自旋，申请和释放锁的时候必须对slock进行修改，将导致所有参与排队自旋锁操作的处理器的缓存变得无效。如果排队自旋锁竞争比较激烈的话，频繁的缓存同步操作会导致繁重的系统总线和内存的流量，从而大大降低了系统整体的性能。

3、mcs spinlock

每个锁的申请者（处理器）只在一个本地变量上自旋。MCS Spinlock是一种基于链表结构的自旋锁。

MCS Spinlock的设计目标如下：

（1）保证自旋锁申请者以先进先出的顺序获取锁（FIFO）

（2）只在本地可访问的标志变量上自旋。

（3）在处理器个数较少的系统中或锁竞争并不激烈的情况下，保持较高性能。

（4）自旋锁的空间复杂度（即锁数据结构和锁操作所需的空间开销）为常数。

（5）在没有处理器缓存一致性协议保证的系统中也能很好地工作。

MCS Spinlock采用链表结构将全体锁申请者的信息串成一个单向链表。每个锁申请者必须提前分配一个本地mcs_lock_node，其中至少包括2个字段：本地自旋变量waiting和指向下一个申请者mcs_lock_node结构的指针变量next。waiting初始值为1，申请者自旋等待其直接前驱释放锁；为0时结束自旋。

自旋锁数据结构mcs_lock是一个永远指向最后一个申请者 mcs_lock_node的指针，当且仅当锁处于未使用（无任何申请者）状态时为NULL值。MCS Spinlock依赖原子的swap和CAS（compare_and_swap）操作，如果缺乏CAS支持，MCS Spinlock 就不能保证以先进先出的顺序获取锁。

每个锁有NR_CPUS个元素node数组，mcs_lock_node结构可以在处理器所处节点的内存中分配，从而加快访问速度。

```
typedef struct _mcs_lock_node {
    volatile int waiting;
    struct _mcs_lock_node *volatile next;
} ____cacheline_aligned_in_smp mcs_lock_node;
```

```
typedef mcs_lock_node *volatile mcs_lock;

typedef struct {
    mcs_lock slock;
    mcs_lock_node nodes[NR_CPUS];
} raw_spinlock_t;
static __always_inline void __raw_spin_lock(raw_spinlock_t *lock)
{
    int cpu;
    mcs_lock_node *me;
    mcs_lock_node *tmp;
    mcs_lock_node *pre;

    cpu = raw_smp_processor_id();
    me = &(lock->nodes[cpu]);
    tmp = me;
    me->next = NULL;

    pre = xchg(&lock->slock, tmp);
    if (pre == NULL) {
        /* mcs_lock is free */
        return;
    }

    me->waiting = 1;
    smp_wmb();
    pre->next = me;

    while (me->waiting) {
        asm volatile ("pause");
    }
}
static __always_inline int __raw_spin_trylock(raw_spinlock_t *lock)
{
    int cpu;
    mcs_lock_node *me;

    cpu = raw_smp_processor_id();
    me = &(lock->nodes[cpu]);
    me->next = NULL;

    if (cmpxchg(&lock->slock, NULL, me) == NULL)
        return 1;
    else
        return 0;
}
static __always_inline void __raw_spin_unlock(raw_spinlock_t *lock)
{
    int cpu;
    mcs_lock_node *me;
    mcs_lock_node *tmp;

    cpu = raw_smp_processor_id();
    me = &(lock->nodes[cpu]);
    tmp = me;

    if (me->next == NULL) {
```

```
        if (cmpxchg(&lock->slock, tmp, NULL) == me) {
            /* mcs_lock I am the last. */
            return;
        }
        while (me->next == NULL)
            continue;
    }

    /* mcs_lock pass to next. */
    me->next->waiting = 0;
}
```

mcs spinlock 锁占用空间大。

4、qspinlock

qspinlock在Linux Kernel 4.2引入，基于mcs spinlock设计思想但解决了mcs spinlock接口不一致或空间太大的问题。

qspinlock数据结构体比mcs lock大大减小，与ticket spinlock大小相同。

```
struct __qspinlock {
    union {
        atomic_t val;
#ifdef __LITTLE_ENDIAN
        struct {
            u8  locked;
            u8  pending;
        };
        struct {
            u16 locked_pending;
            u16 tail;
        };
#else
        struct {
            u16 tail;
            u16 locked_pending;
        };
        struct {
            u8  reserved[2];
            u8  pending;
            u8  locked;
        };
#endif

static __always_inline void queued_spin_lock(struct qspinlock *lock)
{
        u32 val;

        val = atomic_cmpxchg_acquire(&lock->val, 0, _Q_LOCKED_VAL);
        if (likely(val == 0))
                return;
        queued_spin_lock_slowpath(lock, val);
}
```

qspinlock采用mcs lock机制， 每一个CPU都定义有一个struct mcs spinlock数据结构，在大规模多处理器系统和NUM架构中， 使用qspinlock可以较好的提高锁的性能。

5、性能比较

写一个spinlock的性能测试驱动，在等待相同时间后比较spinlock 临界区域的值, 从而比较各个锁的性能差异。

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kthread.h>
#include <linux/sched.h>
#include <linux/kernel/h>
#include <linux/spinlock.h>
#include <linux/random.h>
#include <linux/slab.h>
#incude <linux/timer.h>
#include <linux/jiffies.h>
#include <linux/atomic.h>

int spinlock_num;

struct worker {
    int burns;
    struct task_struct *task;
}
static struct worker *workers;
static int threads = 2;
module_param(threads, int, 0);

static spinlock_t lock;
static int runtime = 10;
module_param(runtime, int, 0);

static int bench_running;
static task_struct *monitor_task;

static int rerun, done;
module_param(rerun, int, S_IRUGO|S_ISUSR);
module_param(done, int, S_IRUGO|S_ISUSR);

static int work(void *data)
{
    struct worker *wk = (struct worker*)arg;
    while(!kthread_should_stop()) {
        cond_resched();

    if (!ACCESS_ONCE(bench_running))
        continue;
    spin_lock(&lock)
    spinlock_num++;
    spin_unlock(&lock);
    }
    return 0;
}

static int monitor(void *unused)
{
    int i, c;
    int total, min, max, avg;
```

```
repeat:
    total = 0, min = INT_MAX, max = 0, avg = 0;

    spinlock_num = 0;

    workers = (struct worker *)kzalloc(sizeof(struct worker) * threads,
GFP_KERNEL);
    for (i = 0; i < threads; i++) {
        c = i %num_online_cpus();
        workers[i].task = kthread_create(work, &workers, "locktest/%d:%d", c,
i);
        kthread_bind(workers[i].task, c);
        wake_up_process(workers[i].task);
    }
    bench_running = 0;
    for (i = 0; i < threads; i++) {
        if (workers[i].task)
            kthread_stop(workers[i].task);
    }
    kfree(workers);
    printk("lockresult:%6d %8d %12d\n", num_online_cpus(), threads,
spinlock_num);
    done = 1;
    while(!kthread_should_stop()) {
        schedule_timeout(1);
        if (cmpxchg(&rerun, done, 0)) {
            done = 0;
            goto repeat;
        }
    }
    return 0;
}

static int locktest_init(void)
{
    monitor_task = kthread_run(monitor, NULL, "monitor");
    return 0;
}

static void locktest_exit(void)
{
    kthread_stop(monitor_task);
}

module_init(locktest_init);
module_exit(locktest_exit);
MODULE_LICENSE("GPL");
```

在CPU较少的情况下， qspinlock的性能和ticket spinlock的性能差不多， 在CPU较多的情况下，qspinlock的性能远好于ticket spinlock。

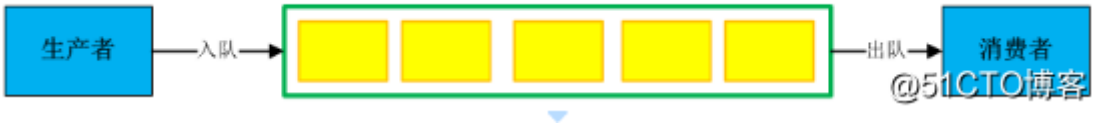
C++性能优化（十三）——无锁队列

一、无锁队列原理

1、队列操作模型

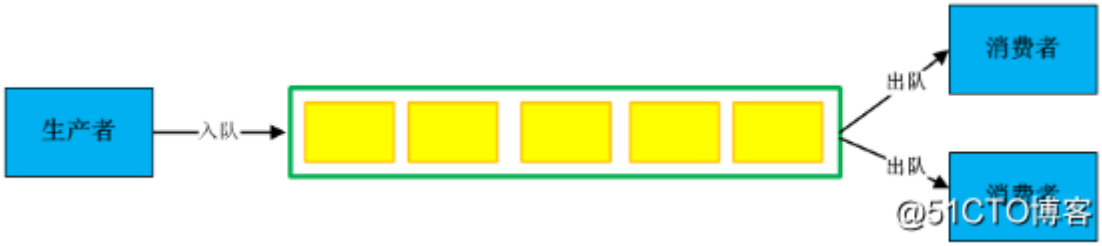
队列是一种非常重要的数据结构，其特性是先进先出（FIFO），符合流水线业务流程。在进程间通信、网络通信间经常采用队列做缓存，缓解数据处理压力。根据操作队列的场景分为：单生产者——单消费者、多生产者——单消费者、单生产者——多消费者、多生产者——多消费者四大模型。根据队列中数据分为：队列中的数据是定长的、队列中的数据是变长的。

（1）单生产者——单消费者



（2）多生产者——单消费者

（3）单生产者——多消费者



（4）多生产者——多消费者

（5）数据定长队列

（6）数据变长队列

2、无锁队列简介

生产环境中广泛使用生产者和消费者模型，要求生产者在生产的同时，消费者可以进行消费，通常使用互斥锁保证数据同步。但线程互斥锁的开销仍然比较大，因此在要求高性能、低延时场景中，推荐使用无锁队列。

3、CAS操作

CAS即Compare and Swap，是所有CPU指令都支持CAS的原子操作（X86中CMPXCHG汇编指令），用于实现实现各种无锁（lock free）数据结构。
CAS操作的C语言实现如下：

```
bool compare_and_swap ( int *memory_location, int expected_value, int new_value)
{
    if (*memory_location == expected_value)
    {
        *memory_location = new_value;
        return true;
    }
    return false;
}
```

CAS用于检查一个内存位置是否包含预期值，如果包含，则把新值复赋值到内存位置。成功返回true，失败返回false。

（1）GGC对CAS支持

GCC4.1+版本中支持CAS原子操作。

```
bool __sync_bool_compare_and_swap (type *ptr, type oldval type newval, ...);
type __sync_val_compare_and_swap (type *ptr, type oldval type newval, ...);
```

（2）Windows对CAS支持

Windows中使用Windows API支持CAS。

```
LONG InterlockedCompareExchange(
  LONG volatile *Destination,
  LONG          ExChange,
  LONG          Comperand
);
```

（3）C11对CAS支持

C11 STL中atomic函数支持CAS并可以跨平台。

```
template< class T >
bool atomic_compare_exchange_weak( std::atomic* obj,T* expected, T desired );
template< class T >
bool atomic_compare_exchange_weak( volatile std::atomic* obj,T* expected, T
desired );
```

其它原子操作如下：

Fetch-And-Add：一般用来对变量做+1的原子操作

Test-and-set：写值到某个内存位置并传回其旧值

二、无锁队列方案

1、boost方案

boost提供了三种无锁方案，分别适用不同使用场景。

boost::lockfree::queue是支持多个生产者和多个消费者线程的无锁队列。

boost::lockfree::stack是支持多个生产者和多个消费者线程的无锁栈。

boost::lockfree::spsc_queue是仅支持单个生产者和单个消费者线程的无锁队列，比boost::lockfree::queue性能更好。

Boost无锁数据结构的API通过轻量级原子锁实现lock-free，不是真正意义的无锁。

Boost提供的queue可以设置初始容量，添加新元素时如果容量不够，则总容量自动增长；但对于无锁数据结构，添加新元素时如果容量不够，总容量不会自动增长。

2、ConcurrentQueue

ConcurrentQueue是基于C实现的工业级无锁队列方案。

`GitHub：https://github.com/cameron314/concurrentqueue`

ReaderWriterQueue是基于C实现的单生产者单消费者场景的无锁队列方案。

`GitHub：https://github.com/cameron314/readerwriterqueue`

3、Disruptor

Disruptor是英国外汇交易公司LMAX基于JAVA开发的一个高性能队列。

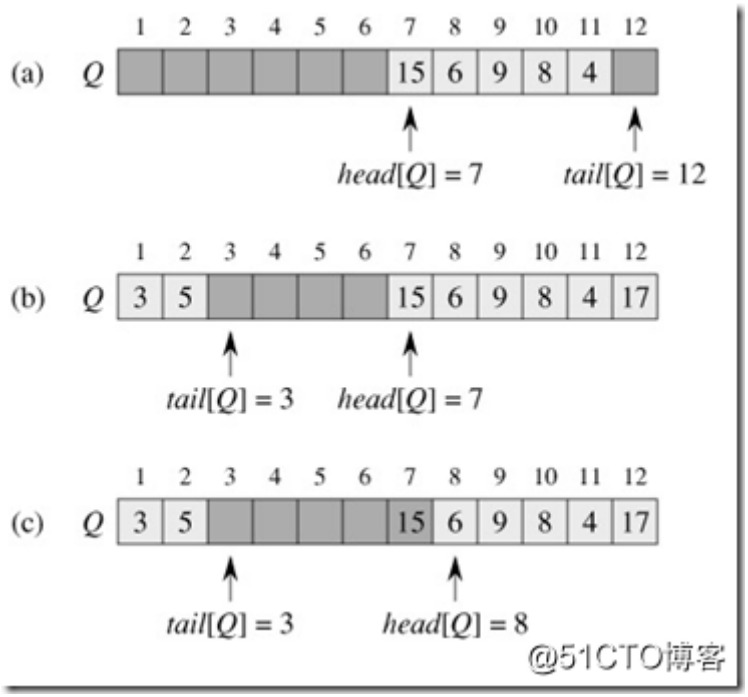
`GitHub：https://github.com/LMAX-Exchange/disruptor`

三、无锁队列实现

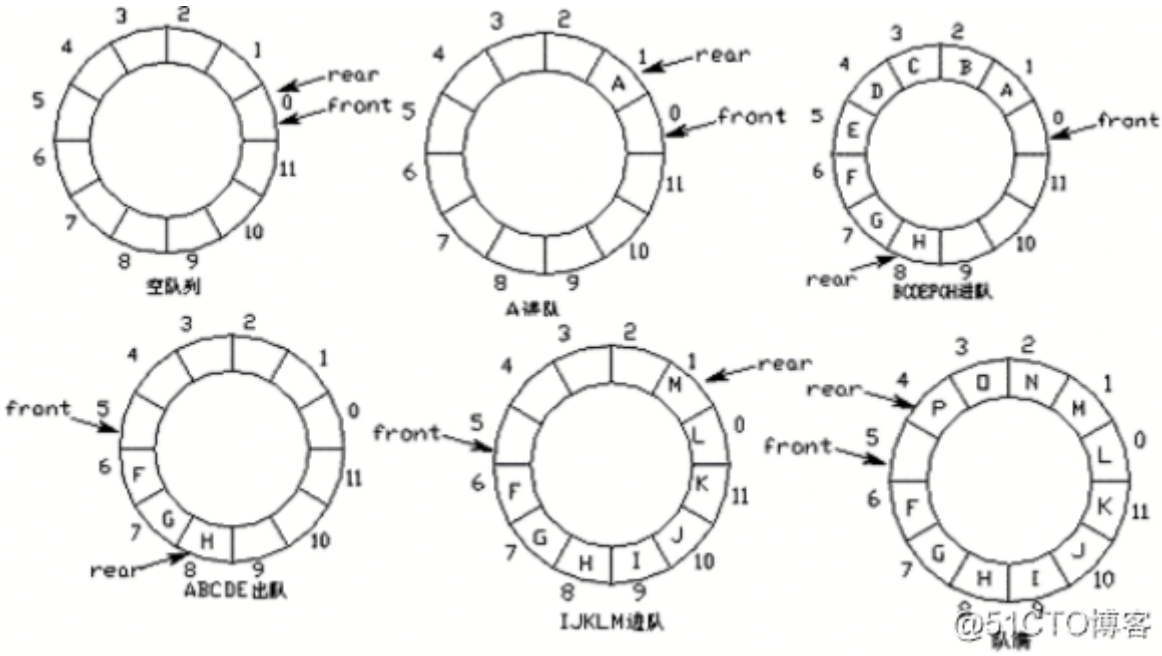
1、环形缓冲区

RingBuffer是生产者和消费者模型中常用的数据结构，生产者将数据追加到数组尾端，当达到数组的尾部时，生产者绕回到数组的头部；消费者从数组头端取走数据，当到达数组的尾部时，消费者绕回到数组头部。

如果只有一个生产者和一个消费者，环形缓冲区可以无锁访问，环形缓冲区的写入index只允许生产者访问并修改，只要生产者在更新index前将新的值保存到缓冲区中，则消费者将始终看到一致的数据结构；读取index也只允许消费者访问并修改，消费者只要在取走数据后更新读index，则生产者将始终看到一致的数据结构。



空队列时，front与rear相等；当有元素进队，则rear后移；有元素出队，则front后移。



空队列时，rear等于front；满队列时，队列尾部空一个位置，因此判断循环队列满时使用(rear-front+maxn)%maxn。

入队操作：

```
x = data[front];
rear = (front+1)%maxn;
```

出队操作：

```
x = data[front];
rear = (front+1)%maxn;
```

2、单生产者单消费者

对于单生产者和单消费者场景，由于read_index和write_index都只会有一个线程写，因此不需要加锁也不需要原子操作，直接修改即可，但读写数据时需要考虑遇到数组尾部的情况。

线程对write_index和read_index的读写操作如下：

（1）写操作。先判断队列时否为满，如果队列未满，则先写数据，写完数据后再修改write_index。

（2）读操作。先判断队列是否为空，如果队列不为空，则先读数据，读完再修改read_index。

3、多生产者单消费者

多生产者和单消费者场景中，由于多个生产者都会修改write_index，所以在不加锁的情况下必须使用原子操作。

4、RingBuffer实现

RingBuffer.hpp文件：

```
#pragma once

template <class T>
class RingBuffer
{
public:
    RingBuffer(unsigned size): m_size(size), m_front(0), m_rear(0)
    {
        m_data = new T[size];
    }

    ~RingBuffer()
    {
        delete [] m_data;
    m_data = NULL;
    }

    inline bool isEmpty() const
    {
        return m_front == m_rear;
    }

    inline bool isFull() const
    {
        return m_front == (m_rear + 1) % m_size;
    }

    bool push(const T& value)
    {
        if(isFull())
        {
            return false;
        }
        m_data[m_rear] = value;
        m_rear = (m_rear + 1) % m_size;
    return true;
    }
```

```
    bool push(const T* value)
    {
        if(isFull())
        {
            return false;
        }
    m_data[m_rear] = *value;
        m_rear = (m_rear + 1) % m_size;
    return true;
    }

    inline bool pop(T& value)
    {
        if(isEmpty())
        {
            return false;
        }
        value = m_data[m_front];
        m_front = (m_front + 1) % m_size;
        return true;
    }
    inline unsigned int front()const
    {
    return m_front;
    }

    inline unsigned int rear()const
    {
    return m_rear;
    }
    inline unsigned int size()const
    {
        return m_size;
    }
private:
    unsigned int m_size;// 队列长度
    int m_front;// 队列头部索引
    int m_rear;// 队列尾部索引
    T* m_data;// 数据缓冲区
};
```

RingBufferTest.cpp测试代码：

```
#include <stdio.h>
#include <thread>
#include <unistd.h>
#include <sys/time.h>
#include "RingBuffer.hpp"


class Test
{
public:
   Test(int id = 0, int value = 0)
   {
    this->id = id;
        this->value = value;
```

```
    sprintf(data, "id = %d, value = %d\n", this->id, this->value);
   }
   void display()
   {
    printf("%s", data);
   }
private:
   int id;
   int value;
   char data[128];
};

double getdetlatimeofday(struct timeval *begin, struct timeval *end)
{
    return (end->tv_sec + end->tv_usec * 1.0 / 1000000) -
           (begin->tv_sec + begin->tv_usec * 1.0 / 1000000);
}

RingBuffer<Test> queue(1 << 12);2u000

#define N (10 * (1 << 20))

void produce()
{
    struct timeval begin, end;
    gettimeofday(&begin, NULL);
    unsigned int i = 0;
    while(i < N)
    {
        if(queue.push(Test(i % 1024, i)))
        {
       i++;
        }
    }
    gettimeofday(&end, NULL);
    double tm = getdetlatimeofday(&begin, &end);
    printf("producer tid=%lu %f MB/s %f msg/s elapsed= %f size= %u\n",
pthread_self(), N * sizeof(Test) * 1.0 / (tm * 1024 * 1024), N * 1.0 / tm, tm,
i);
}

void consume()
{
    sleep(1);
    Test test;
    struct timeval begin, end;
    gettimeofday(&begin, NULL);
    unsigned int i = 0;
    while(i < N)
    {
        if(queue.pop(test))
    {
       // test.display();
       i++;
    }
    }
    gettimeofday(&end, NULL);
    double tm = getdetlatimeofday(&begin, &end);
```

```
    printf("consumer tid=%lu %f MB/s %f msg/s elapsed= %f, size=%u \n",
pthread_self(), N * sizeof(Test) * 1.0 / (tm * 1024 * 1024), N * 1.0 / tm, tm,
i);
}

int main(int argc, char const *argv[])
{
    std::thread producer1(produce);
    std::thread consumer(consume);
    producer1.join();
    consumer.join();
    return 0;
}
```

编译:

```
g++ --std=c++11 RingBufferTest.cpp -o test -pthread
```

单生产者单消费者场景下，消息吞吐量为350万条/秒左右。

5、LockFreeQueue实现

LockFreeQueue.hpp:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdbool.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/mman.h>

#define SHM_NAME_LEN 128
#define MIN(a, b) ((a) > (b) ? (b) : (a))
#define IS_POT(x) ((x) && !((x) & ((x)-1)))
#define MEMORY_BARRIER __sync_synchronize()

template <class T>
class LockFreeQueue
{
protected:
    typedef struct
    {
    int m_lock;
    inline void spinlock_init()
    {
        m_lock = 0;
    }
    inline void spinlock_lock()
    {
        while(!__sync_bool_compare_and_swap(&m_lock, 0, 1)) {}
    }
    inline void spinlock_unlock()
    {
```

```
        __sync_lock_release(&m_lock);
    }
    } spinlock_t;
public:
    // size:队列大小
    // name:共享内存key的路径名称，默认为NULL，使用数组作为底层缓冲区。
    LockFreeQueue(unsigned int size, const char* name = NULL)
    {
    memset(shm_name, 0, sizeof(shm_name));
    createQueue(name, size);
    }
    ~LockFreeQueue()
    {
    if(shm_name[0] == 0)
    {
        delete [] m_buffer;
        m_buffer = NULL;
    }
    else
    {
        if (munmap(m_buffer, m_size * sizeof(T)) == -1) {
        perror("munmap");
        }
        if (shm_unlink(shm_name) == -1) {
        perror("shm_unlink");
        }
    }
    }

    bool isFull()const
    {
#ifdef USE_POT
        return m_head == (m_tail + 1) & (m_size - 1);
#else
    return m_head == (m_tail + 1) % m_size;
#endif
    }

    bool isEmpty()const
    {
    return m_head == m_tail;
    }
    unsigned int front()const
    {
    return m_head;
    }

    unsigned int tail()const
    {
    return m_tail;
    }

    bool push(const T& value)
    {
#ifdef USE_LOCK
        m_spinLock.spinlock_lock();
#endif
        if(isFull())
```

```
        {
#ifdef USE_LOCK
            m_spinLock.spinlock_unlock();
#endif
            return false;
        }
        memcpy(m_buffer + m_tail, &value, sizeof(T));
#ifdef USE_MB
        MEMORY_BARRIER;
#endif

#ifdef USE_POT
        m_tail = (m_tail + 1) & (m_size - 1);
#else
        m_tail = (m_tail + 1) % m_size;
#endif

#ifdef USE_LOCK
        m_spinLock.spinlock_unlock();
#endif
        return true;
    }

    bool pop(T& value)
    {
#ifdef USE_LOCK
        m_spinLock.spinlock_lock();
#endif
        if (isEmpty())
        {
#ifdef USE_LOCK
            m_spinLock.spinlock_unlock();
#endif
            return false;
        }
        memcpy(&value, m_buffer + m_head, sizeof(T));
#ifdef USE_MB
        MEMORY_BARRIER;
#endif

#ifdef USE_POT
        m_head = (m_head + 1) & (m_size - 1);
#else
        m_head = (m_head + 1) % m_size;
#endif

#ifdef USE_LOCK
        m_spinLock.spinlock_unlock();
#endif
        return true;
    }
protected:
    virtual void createQueue(const char* name, unsigned int size)
    {
#ifdef USE_POT
        if (!IS_POT(size))
        {
            size = roundup_pow_of_two(size);
```

```
        }
#endif
        m_size = size;
    m_head = m_tail = 0;
    if(name == NULL)
    {
        m_buffer = new T[m_size];
    }
        else
        {
        int shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
        if (shm_fd < 0)
        {
            perror("shm_open");
        }
        if (ftruncate(shm_fd, m_size * sizeof(T)) < 0)
        {
            perror("ftruncate");
            close(shm_fd);
        }
        void *addr = mmap(0, m_size * sizeof(T), PROT_READ | PROT_WRITE,
MAP_SHARED, shm_fd, 0);
        if (addr == MAP_FAILED)
        {
            perror("mmap");
            close(shm_fd);
        }
        if (close(shm_fd) == -1)
        {
            perror("close");
            exit(1);
        }
        m_buffer = static_cast<T*>(addr);
        memcpy(shm_name, name, SHM_NAME_LEN - 1);
        }
#ifdef USE_LOCK
    spinlock_init(m_lock);
#endif
    }
    inline unsigned int roundup_pow_of_two(size_t size)
    {
    size |= size >> 1;
    size |= size >> 2;
    size |= size >> 4;
    size |= size >> 8;
    size |= size >> 16;
    size |= size >> 32;
    return size + 1;
    }

protected:
    char shm_name[SHM_NAME_LEN];
    volatile unsigned int m_head;
    volatile unsigned int m_tail;
    unsigned int m_size;
#ifdef USE_LOCK
    spinlock_t m_spinLock;
#endif
```

```
    T* m_buffer;
};
```

`#define USE_LOCK`

开启spinlock锁，多生产者多消费者场景

`#define USE_MB`

开启Memory Barrier

`#define USE_POT`

开启队列大小的2的幂对齐

LockFreeQueueTest.cpp测试文件：

```
#include "LockFreeQueue.hpp"
#include <thread>

//#define USE_LOCK

class Test
{
public:
   Test(int id = 0, int value = 0)
   {
    this->id = id;
        this->value = value;
    sprintf(data, "id = %d, value = %d\n", this->id, this->value);
   }
   void display()
   {
    printf("%s", data);
   }
private:
   int id;
   int value;
   char data[128];
};

double getdetlatimeofday(struct timeval *begin, struct timeval *end)
{
    return (end->tv_sec + end->tv_usec * 1.0 / 1000000) -
           (begin->tv_sec + begin->tv_usec * 1.0 / 1000000);
}

LockFreeQueue<Test> queue(1 << 10, "/shm");

#define N ((1 << 20))

void produce()
{
    struct timeval begin, end;
    gettimeofday(&begin, NULL);
    unsigned int i = 0;
    while(i < N)
    {
        if(queue.push(Test(i >> 10, i)))
```

```
        i++;
    }
    gettimeofday(&end, NULL);
    double tm = getdetlatimeofday(&begin, &end);
    printf("producer tid=%lu %f MB/s %f msg/s elapsed= %f size= %u\n",
pthread_self(), N * sizeof(Test) * 1.0 / (tm * 1024 * 1024), N * 1.0 / tm, tm,
i);
}

void consume()
{
    Test test;
    struct timeval begin, end;
    gettimeofday(&begin, NULL);
    unsigned int i = 0;
    while(i < N)
    {
        if(queue.pop(test))
    {
       //test.display();
       i++;
    }
    }
    gettimeofday(&end, NULL);
    double tm = getdetlatimeofday(&begin, &end);
    printf("consumer tid=%lu %f MB/s %f msg/s elapsed= %f size= %u\n",
pthread_self(), N * sizeof(Test) * 1.0 / (tm * 1024 * 1024), N * 1.0 / tm, tm,
i);
}

int main(int argc, char const *argv[])
{
    std::thread producer1(produce);
    //std::thread producer2(produce);
    std::thread consumer(consume);
    producer1.join();
    //producer2.join();
    consumer.join();

    return 0;
}
```

多线程场景下，需要定义USE_LOCK宏，开启锁保护。

编译：

```
g++ --std=c++11 -O3 LockFreeQueueTest.cpp -o test -lrt -pthread
```

```
[user@localhost RingBuffer]$ g++ --std=c++11 -O3  LockFreeQueueTest.cpp -o test -lrt -pthread
[user@localhost RingBuffer]$ ./test
producer tid=139877638125312 492.554030 MB/s 3797649.515631 msg/s elapsed= 0.276112 size= 1048576
consumer tid=139877629732608 492.591460 MB/s 3797938.108516 msg/s elapsed= 0.27[illegible]
[user@localhost RingBuffer]$
```


四、kfifo内核队列

1、kfifo内核队列简介

kfifo是Linux内核的一个FIFO数据结构，采用环形循环队列的数据结构来实现，提供一个无边界的字节流服务，并且使用并行无锁编程技术，即单生产者单消费者场景下两个线程可以并发操作，不需要任何加锁行为就可以保证kfifo线程安全。

2、kfifo内核队列实现

kfifo数据结构定义如下：

```
struct kfifo
{
    unsigned char *buffer;
    unsigned int size;
    unsigned int in;
    unsigned int out;
    spinlock_t *lock;
};

// 创建队列
struct kfifo *kfifo_init(unsigned char *buffer, unsigned int size, gfp_t
gfp_mask, spinlock_t *lock)
{
    struct kfifo *fifo;
    // 判断是否为2的幂
    BUG_ON(!is_power_of_2(size));
    fifo = kmalloc(sizeof(struct kfifo), gfp_mask);
    if (!fifo)
        return ERR_PTR(-ENOMEM);
    fifo->buffer = buffer;
    fifo->size = size;
    fifo->in = fifo->out = 0;
    fifo->lock = lock;

    return fifo;
}
// 分配空间
struct kfifo *kfifo_alloc(unsigned int size, gfp_t gfp_mask, spinlock_t *lock)
{
    unsigned char *buffer;
    struct kfifo *ret;
    // 判断是否为2的幂
    if (!is_power_of_2(size))
    {
        BUG_ON(size > 0x80000000);
        // 向上扩展成2的幂
        size = roundup_pow_of_two(size);
    }
    buffer = kmalloc(size, gfp_mask);
    if (!buffer)
        return ERR_PTR(-ENOMEM);
    ret = kfifo_init(buffer, size, gfp_mask, lock);

    if (IS_ERR(ret))
        kfree(buffer);
    return ret;
}
```

```c
void kfifo_free(struct kfifo *fifo)
{
    kfree(fifo->buffer);
    kfree(fifo);
}

// 入队操作
static inline unsigned int kfifo_put(struct kfifo *fifo, const unsigned char
*buffer, unsigned int len)
{
    unsigned long flags;
    unsigned int ret;
    spin_lock_irqsave(fifo->lock, flags);
    ret = __kfifo_put(fifo, buffer, len);
    spin_unlock_irqrestore(fifo->lock, flags);
    return ret;
}
// 出队操作
static inline unsigned int kfifo_get(struct kfifo *fifo, unsigned char *buffer,
unsigned int len)
{
    unsigned long flags;
    unsigned int ret;
    spin_lock_irqsave(fifo->lock, flags);
    ret = __kfifo_get(fifo, buffer, len);
    //当fifo->in == fifo->out时，buufer为空
    if (fifo->in == fifo->out)
        fifo->in = fifo->out = 0;
    spin_unlock_irqrestore(fifo->lock, flags);
    return ret;
}
// 入队操作
unsigned int __kfifo_put(struct kfifo *fifo, const unsigned char *buffer,
unsigned int len)
{
    unsigned int l;
    //buffer中空的长度
    len = min(len, fifo->size - fifo->in + fifo->out);
    // 内存屏障：smp_mb()，smp_rmb()，smp_wmb()来保证对方观察到的内存操作顺序
    smp_mb();
    // 将数据追加到队列尾部
    l = min(len, fifo->size - (fifo->in & (fifo->size - 1)));
    memcpy(fifo->buffer + (fifo->in & (fifo->size - 1)), buffer, l);
    memcpy(fifo->buffer, buffer + l, len - l);

    smp_wmb();
    //每次累加，到达最大值后溢出，自动转为0
    fifo->in += len;
    return len;
}
// 出队操作
unsigned int __kfifo_get(struct kfifo *fifo, unsigned char *buffer, unsigned int
len)
{
    unsigned int l;
    //有数据的缓冲区的长度
    len = min(len, fifo->in - fifo->out);
```

```
    smp_rmb();
    l = min(len, fifo->size - (fifo->out & (fifo->size - 1)));
    memcpy(buffer, fifo->buffer + (fifo->out & (fifo->size - 1)), l);
    memcpy(buffer + l, fifo->buffer, len - l);
    smp_mb();
    fifo->out += len; //每次累加，到达最大值后溢出，自动转为0
    return len;
}

static inline void __kfifo_reset(struct kfifo *fifo)
{
    fifo->in = fifo->out = 0;
}

static inline void kfifo_reset(struct kfifo *fifo)
{
    unsigned long flags;
    spin_lock_irqsave(fifo->lock, flags);
    __kfifo_reset(fifo);
    spin_unlock_irqrestore(fifo->lock, flags);
}

static inline unsigned int __kfifo_len(struct kfifo *fifo)
{
    return fifo->in - fifo->out;
}

static inline unsigned int kfifo_len(struct kfifo *fifo)
{
    unsigned long flags;
    unsigned int ret;
    spin_lock_irqsave(fifo->lock, flags);
    ret = __kfifo_len(fifo);
    spin_unlock_irqrestore(fifo->lock, flags);
    return ret;
}
```

3、kfifo设计要点

（1）保证buffer size为2的幂

`kfifo->size`值在调用者传递参数size的基础上向2的幂扩展，目的是使`kfifo->size`取模运算可以转化为位与运算（提高运行效率）。`kfifo->in % kfifo->size`转化为`kfifo->in & (kfifo->size - 1)`保证size是2的幂可以通过位运算的方式求余，在频繁操作队列的情况下可以大大提高效率。

（2）使用spin_lock_irqsave与spin_unlock_irqrestore 实现同步。

Linux内核中有spin_lock、spin_lock_irq和spin_lock_irqsave保证同步。

```
static inline void __raw_spin_lock(raw_spinlock_t *lock)
{
    preempt_disable();
    spin_acquire(&lock->dep_map, 0, 0, _RET_IP_);
    LOCK_CONTENDED(lock, do_raw_spin_trylock, do_raw_spin_lock);
}

static inline void __raw_spin_lock_irq(raw_spinlock_t *lock)
```

```
{
    local_irq_disable();
    preempt_disable();
    spin_acquire(&lock->dep_map, 0, 0, _RET_IP_);
    LOCK_CONTENDED(lock, do_raw_spin_trylock, do_raw_spin_lock);
}
```

spin_lock比spin_lock_irq速度快，但并不是线程安全的。spin_lock_irq增加调用local_irq_disable函数，即禁止本地中断，是线程安全的，既禁止本地中断，又禁止内核抢占。

spin_lock_irqsave是基于spin_lock_irq实现的一个辅助接口，在进入和离开临界区后，不会改变中断的开启、关闭状态。

如果自旋锁在中断处理函数中被用到，在获取自旋锁前需要关闭本地中断，spin_lock_irqsave实现如下：

A、保存本地中断状态；

B、关闭本地中断；

C、获取自旋锁。

解锁时通过 spin_unlock_irqrestore完成释放锁、恢复本地中断到原来状态等工作。

（3）线性代码结构

代码中没有任何if-else分支来判断是否有足够的空间存放数据，kfifo每次入队或出队只是简单的 +len 判断剩余空间，并没有对kfifo->size 进行取模运算，所以kfifo->in和kfifo->out总是一直增大，直到unsigned in超过最大值时绕回到0这一起始端，但始终满足：`kfifo->in - kfifo->out <= kfifo->size`。

（4）使用Memory Barrier

mb()：适用于多处理器和单处理器的内存屏障。

rmb()：适用于多处理器和单处理器的读内存屏障。

wmb()：适用于多处理器和单处理器的写内存屏障。

smp_mb()：适用于多处理器的内存屏障。

smp_rmb()：适用于多处理器的读内存屏障。

smp_wmb()：适用于多处理器的写内存屏障。

Memory Barrier使用场景如下：

A、实现同步原语（synchronization primitives）

B、实现无锁数据结构（lock-free data structures）

C、驱动程序

程序在运行时内存实际访问顺序和程序代码编写的访问顺序不一定一致，即内存乱序访问。内存乱序访问行为出现是为了提升程序运行时的性能。内存乱序访问主要发生在两个阶段：

A、编译时，编译器优化导致内存乱序访问（指令重排）。

B、运行时，多CPU间交互引起内存乱序访问。

Memory Barrier能够让CPU或编译器在内存访问上有序。Memory barrier前的内存访问操作必定先于其后的完成。Memory Barrier包括两类：

A、编译器Memory Barrier。

B、CPU Memory Barrier。

通常，编译器和CPU引起内存乱序访问不会带来问题，但如果程序逻辑的正确性依赖于内存访问顺序，内存乱序访问会带来逻辑上的错误。

在编译时，编译器对代码做出优化时可能改变实际执行指令的顺序（如GCC的O2或O3都会改变实际执行指令的顺序）。

在运行时，CPU虽然会乱序执行指令，但在单个CPU上，硬件能够保证程序执行时所有的内存访问操作都是按程序代码编写的顺序执行的，Memory Barrier没有必要使用（不考虑编译器优化）。

为了更快执行指令，CPU采取流水线的执行方式，编译器在编译代码时为了使指令更适合CPU的流水线执行方式以及多CPU执行，原本指令就会出现乱序的情况。在乱序执行时，CPU真正执行指令的顺序由可用的输入数据决定，而非程序员编写的顺序。

https://github.com/scorpiostudio/RingBuffer