1 进程和线程的区别

1、线程是CPU调度的基本单位；进程是能独立运行的基本单位，是资源分配的基本单位; 2、创建和销毁线程的成本比进程低； 3、进程拥有自己独立的虚拟地址空间；一个进程中的多个线程共享进程的虚拟地址空间；4、线程占用的资源比进程少； 5、线程缺少访问控制，进程中的一个线程出错，会终止掉整个进程。

1.1 进程和线程优点

1.1.1 线程优点

1、线程间通信比较容易，成本低，调度切换快； 2、创建销毁成本低； 3、线程间切换容易； 4、等待慢速O/I时，可以干其他事。

1.1.2 进程优点

1、安全更强的容错性，一个进程挂了不会影响其他进程； 2、容易调试。

1.2 进程和线程的应用场景

1.2.1 线程应用场景

通信（注意加锁）；等待慢速O/I时，可以干其他事

1.2.2 进程应用场景

需要安全稳定时用进程。能用多进程尽量不要用多线程。

2 进程状态及转换

2.1 进程转态

1、新建态： 刚刚创建的进程，操作系统还没有把它加入可执行组中，通常是进程控制块已经创建但还没有加载到内存中的进程。 2、就绪态： 进程已经做好了准备，只要有机会就开始执行。 3、运行态： 该进程正在执行。 4、阻塞态： 进程在某些事情发生前不能执行，等待阻塞进程的事件完成。 5、退出态： 操作系统从可执行组中释放出的进程，或由于自身原因停止运行。

2.2 进程状态转换

新建态->就绪态：提交 就绪态->运行态：进程调度 运行态->就绪态：时间耗尽 运行态->阻塞态：事件请求（进程请求一个无法立即得到的资源） 阻塞态->就绪态：事件发生（当等待的事件、资源发生时） 运行态->退出态：释放

3 进程调度算法的特点和引用场景

3.1 时间片轮转调度算法（PR）：

给每个进程固定的执行时间，根据进程到达的先后顺序让进程在单位时间片内执行，执行完成后调度下一个进程执行。 时间片轮转调度不考虑进程等待时间和执行时间，属于抢占式调度。 优点是兼顾长短作业；缺点是平均等待时间较长，上下文切换较费时。 适用于分时系统。

3.2 前来先服务算法（FCFS）：

根据进程到达的前后顺序执行进程，不考虑等待时间和执行时间，会产饥饿现象。属于非抢占式调度。 优点是公平，实现简单；缺点是不利于短作业。

3.3 优先级调度算法：

在进程等待队列中选择优先级最高的来执行。 常用于批处理系统中，还可用于实时系统中。

3.4 多级反馈队列调度算法：

将时间片轮转与优先级调度相结合，把进程按优先级分成不同队列，先按优先级调度，如果优先级相同，按时间片轮转。 优点是兼顾长短作业，有较好的的响应时间，可行性强，适用于各种作业环境。

3.5 高响应比优先调度算法：

根据"响应比=（进程执行时间+进程等待时间）/进程执行时间"这个公式得到的响应比来进行调度。高响应比优先算法在等待时间相同的情况下，执行时间越短，响应比越高，满足段任务优先，同时响应比会随着等待时间的增加而增大，优先级会提高，能够避饥饿现象。 优点是兼顾长短作业。缺点是计算响应比开销大，适用于批处理系统。

4 线程实现的方法

4.1 继承Thread类

继承Thread类的子类，并重写Thread类的run()方法，创建子类对象（即线程对象），调用线程对象的start()方法来启动线程。

```
public class ThreadDemo extends Thread{
    public void run(){
            System.out.println(getName());
    }
    public static void main(String[] args){
       ThreadDemo t1 = new ThreadDemo();
       ThreadDemo t2 = new ThreadDemo();
       t1.start();
       t2.start();
    }
}
```

4.2 实现Runnable接口创建线程

如果类已经extends另一个类无法直接继承Thread，此时可以实现一个Runnable接口。定义Runnable接口的实现类，并重新改接口的run()方法：

```
public class ThreadDemo implements Runnable{
    public void run(){
        System.out.println(Thread.currentThread().getName());
    }
    public static void main(String[] args){
        ThreadDemo t1 = new ThreadDemo();
        ThreadDemo t2 = new ThreadDemo();
        new Thread(t1).start();
        new Thread(t2).start();
    }
}
```

4.3 实现Callable接口通过FutureTask包装器来创建Thread线程

Callable的接口定义如下（也只有一个方法）：

```
public interface Callable<V>{
    V call() throws Exception;
}
```

具体类的实现

```
public class SomeCallable<V> extends OtherClass implements Callable<V>{
    @override
    public V call() throws Exception{
        return null;
    }
    public static void main(String[] agrs){
        Callable<V> oneCallable = new SomeCallbale();
        //用Callable<V>创建一个FutureTask<V>对象
        FutureTask<V> oneTask = new FutureTask<V>(oneCallable);
        //FutureTask是一个包装器，他通过接受Callable来创建，同时实现了Runnable和Future
接口
        //有FutureTask创建一个Thread对象
        Thread oneThread = new Thread(oneTask);
        oneTask.start();
    }
}
```

4.4 使用ExecutorService、Callable、Future实现有返回结果的线程

可返回值的接口必须实现Callable接口，无返回值的任务必须实现Runnable接口； 执行Callable任务后，可以获取一个Future对象，在该对象上调用get就可以获取到Callable任务返回的Object。 再结合线程池接口ExecutorService就可以实现又返回结果的多线程。

```
int taskSize = 5;
// 创建一个线程池
ExecutorService pool = Executors.newFixedThreadPool(taskSize);
```

5 协程的作用

协程看起来是子程序，但在子程序内部可以中断，转而执行别的子程序，在适当的时候再返回接着执行。在一个子程序中中断，去执行其他子程序，不是函数调用类似于CPU的中断

```
def A():
    print '1'
    print '2'
    print '3'

def B():
    print 'x'
    print 'y'
    print 'z'
```

在执行A的过程中可以随时中断执行B，B也有可能随时中断执行A，结果可能是：1 2 x y x z

协程的优势： 协程是在一个线程中执行，最大的优势就是协程极高的执行效率，因为子程序切换不是线程切换，而是由程序自身控制。 第二个优势是不需要多线程的锁机制，因为只有一个线程，也不存在同时写变量冲突，在协程中控制共享资源不加锁，只需要判断转态，所以在执行效率比线程高。

6 常见线程同步问题

涉及进程同步的一些概念

同步和互斥： 临界区资源：指一次只允许一个进程访问使用的共享资源成为临界区资源。 同步：指为完成某种任务简历的一个或多个进程，这些进程在合作的过程中需要协调工作次序进行有序访问而出现等待产生的制约关系。 互斥：指两个或多个进程访问临界区资源时只能有一个进程访问，其他进程等待的互相制约的关系。

信号量和互斥量： 信号量：本身是一个计数器，使用P（减一），V（加一）两个操作来实现计数的加减，当计数不大于0时，则进程进入休眠状态，它用于为多个进程提供共享数据对象的访问。 互斥量：如果信号量只存在两个状态，那就不需要计数，简化为加锁和解锁两个状态，这就是互斥量。

6.1 生产者与消费者问题

问题描述：两个线程——生产者和消费者共享固定大小的缓冲区。生产者的作用是产生数据放入缓冲区中，消费者在缓冲区中消耗这些数据。问题的关键是保证生产者不会再缓冲区满的时候加入数据，消费者不会再缓冲

区为空的时候消耗数据。 问题分析：生产者和消费者进程对缓冲区的访问时互斥关系，而生产者和消费者本身又存在同步关系，身缠之后才能消费。所以对缓冲区设置一个互斥量，再设置两个信号量实现同步。 问题解决：

```
semaphore mutex = 1;  //互斥量
semaphore full = 0;   //满缓冲区单元
semaphore empty = N;  //空缓冲区单元
//生产者
prodecer(){
    while(1){
        P(empty);   //缓冲区剩余空间减一
        P(mutex);
        add_sourece++;
        V(mutex);
        V(full);    缓冲区已占空间加一
    }
}
//消费者
consumer(){
    while(1){
        P(full);     //缓冲区已占空间减一
        P(mutex);    //互斥量减一
        add_source--;
        V(mutex);    //互斥量加一
        V(empty);    //缓冲区剩余空间加一
    }
}
```

6.2 读者和写者问题

问题描述：有读者和写者两个并发进程共享一个数据，两个或两个以上的读进程可以访问数据，但是一个写进程访问数据与其他进程都互斥。 问题分析：读者和写者是互斥关系，写者和写者是互斥关系，读者和读者是同步关系。所以需要一个互斥量实现读写和写写互斥，一个读者访问计数实现对计数的互斥。 问题解决： 1、读写公平：读者与写者公平抢占资源，但是只要之前已经排队的读者，就算写者获取资源也要等所有等待的读者进程结束。

```
//公平读写
int count = 0;
semaphore mutex = 1; //读者计数器
semaphore rw = 1;    //资源访问锁
semaphore w = 1;     //读写公平抢占锁
//写作者
writer(){
    while(1){
        P(w);
        P(rw);
        writing sth;
        V(rw);
        V(w);
```

```
    }
}
//读者
reader(){
    while(1){
        P(w);
        P(mutex);
        if(count==0)
            P(rw);
        count++;
        V(mutex);
        V(w);
        reading sth;
        P(mutex);
        count--;
        if(count--)
            V(rw);
        V(mutex);
    }
}
```

6.3 哲学家就餐问题

问题描述：一张圆桌上有5个哲学家，每两个哲学家之间有一双筷子，哲学家只有同时拿起左右两支筷子才可以用餐，用晚餐筷子放回原处。 问题分析：五个哲学家就是五个进程，五根筷子是要回去的资源。可以定义互斥数组用来表示五根筷子的互斥访问，为了防止哲学家合区一根筷子死锁，需要添加一定的限制条件。一种方法是限制仅当左右筷子都可以使用时才拿起，需要一个互斥量来限制筷子不会出现竞争。

```
semaphore chopstick[5] = {1,1,1,1,1};
semaphore mutex = 1;
pi(){
    while(1){
        P(mutex);
        P(chopstick[i]);
        P(chopstick[(i+1)%5]);
        V(mutex);
        eating;
        V(chopstick[i]);
        V(chopstick[(i+1)%5]);
    }
}
```

7 进程通信方法的特点以及使用场景

实现短线程通信的模型有两种：共享内存和消息传递。 题目：有两个进程A和B，A进程向集合中添加"abc"，一共添加十次，当添加5次后，希望B线程执行相关动作。

7.1 使用volatile关键字

基于volatile关键字实现线程通信是时使用共享内存的思想，多个线程同时监听一个变量，当这个变量发生变化时，线程能够感知并执行相关业务。

```
public class TestSync{
    //定义共享变量，使用volatile关键字修饰
    static volatile boolean notice = false;

    public static void main(String[] agrs){
        List<String> list = new ArrayList<>();
        //实现A线程
        Thread threadA = new Thread(()->{
            for(int i=0;i<10;i++){
                list.add("abc");
                try{
                    Thread.sleep(500);
                }catch(InterruptedException e){
                    e.printStackTrace();
                }
                if(list.size()==5){
                    notice = true;
                }
            }
        });
        //实现B线程
        Thread threadB = new Thread(()->{
            while(true){
                if(notice){
                    System.out.println("B线程执行业务");
                    break;
                }
            }
        });
        threadB.start();          //先启动B线程
        try{
            Thread.sleep(1000);
        }catch(Exception){
            e.printStackTrace();
        }
        threadA.start();
    }
}
```

7.2 使用Object类的wait()和notify()方法

Object类提供了线程间通信的方法：wait()、notify()、notifyAl()，它们是多线程通信的基础，而这种实现方式的思想自然是线程间通信。 注意：wait和notify必须配合synchronized使用，wait方法释放锁，notify方法不释放锁。 wait()、notify()和notifyAll()方法时object的本地final方法，无法重写。 wait()是使当前线程阻塞，会释放当前锁，让出CPU进入等待态； notify()被执行时唤醒一个正处于等待的线程，然后继续执行知道完成synchronized代码块中的代码（所以一般放在代码块的最后执行）或中途遇到wait()释放锁。notifyAll()会唤醒所有等待态的线程。

```
public class SynTest{
    public static void main(String[] agrs){
        //定义一个锁对象
        Object lock = new Object();
        List<String> list = new ArrayList<>();
        //实现线程A
        Thread threadA = new Thread(()->{
            synchroized(lock){
                for(int i=1;i<=10;i++){
                    list.add("abc");
                    try{
                        Thread.sleep(100);
                    }catch(Exception e){
                        e.printStackTrace();
                    }
                    if(i==5){
                        lock.notify();    //唤醒其他等待进程B
                    }
                }
            }
        });
        //实现进程B
        Thread threadB = new Thread(()->{
            while(true){
                synchroized(lock){
                    if(list.size()!=5){
                        try{
                            lock.wait();    //阻塞当前进程
                        }catch(Exception e){
                            e.printStackTrace();
                        }
                    }
                }
                System.out.println("B线程被唤醒，开始执行自己的业务");
            }
        });
        threadB.start();    //先启动B线程
        try{
            Thread.sleep();
        }catch(Exception e){
            e.printStackTrace();
        }
        threadA.start();    //再启动A线程
    }
}
```

在线程A发出notify()唤醒通知之后，依然是走完了自己线程的业务之后，线程B才开始执行，这也正好说明了，notify()方法不释放锁，而wait()方法释放锁

7.3 使用JUC工具类CountDownLatch

在java.util.concurrent包下提供了很多并行编码的开发工具类，CountDownLatch基于AQS框架，相当于维护一个线程共享变量state。

```
public class SynTest{
    public static void main(String[] args){
        CountDownLatch countDownLatch = new CountDownLatch(1);
        List<String> list = new ArrayList<>();
        //实现A线程
        Thread threadA = new Thread(()->{
            for(int i=1;i<=5;i++){
                list.add("abc");
                try{
                    Thread.sleep(100);
                }catch(Exception e){
                    e.printStackTrace();
                }
                if(list.size()==5){
                    countDownLatch.countDown();
                }
            }
        });
        //实现进程B
        Thread threadB = new Thread(()->{
            while(true){
                if(list.size()!=5){
                    try{
                        countDownLacth.wait();
                    }catch(Exception e){
                        e.printStackTrace();
                    }
                }
                //省略执行自己的业务的代码
                //..........
            }
        });
    }
}
```

8 Synchronized的用法

8.1 作用于实例方法

使用作用范围为整个函数，这里的实例锁是调用该实例方法的对象（不包括静态方法）：

```
public class SynTest implements Runnable{
    int count = 0; //共享资源
    @Override
    public void run(){
        for(int i=0;i<5;i++){
```

```
            increaseCount();
            System.out.println(Thread.currentThread().getName()+" "+count++);
        }
    }
    public static void main(String[] args){
            SynTest synTest1 = new TestSyn();
            //SynTest synTest2 = new TestSyn();
            Thread thread1 = new Thread(synTest1,"thead1");
            Thread thread2 = new Thread(synTest1,"thread2");
            thread1.start();
            thread2.start();
    }
}
```

运行结果为：

```
    thread1:0
    ...
    thread1:9
    ...
    thread2:0
    ...
    thread2:9
```

结果说明当thread1 执行run方法时，thread2是访问不了的。但这个实例方法时synTest1的，如果有两个实例对象，分别使用实例对象的方法则并不会保证线程安全，如下所示。

```
public static void main(String[] args){
        SynTest synTest1 = new TestSyn();
        SynTest synTest2 = new TestSyn();
        Thread thread1 = new Thread(synTest1,"thead1");
        Thread thread2 = new Thread(synTest2,"thread2");
        thread1.start();
        thread2.start();
}
```

8.2 作用于静态方法

静态方法是不属于当前实例的，而是属于类的，所以这个锁就是类的class对象锁。

```
public class SynTest implements Runnable{
    static int count = 0;

    public synchronized void run(){
        increaseCount();
    }
    public static synchronized void increaseCount(){
```

```
        for(int i=0;i<5;i++){
            System.out.println(Thread.currentThread().getName()+" "+count++);
            try{
                Thread.sleep(1000);
            }catch(InterruptedException){
                e.printStackTrack();
            }
        }
    }
    public static void main(String[] args){
        SynTest synTest1 = new SynTest();
        SynTest synTest2 = new SynTest();
        Thread thread1 = new Thread(synTest1,"thead1");
        Thread thread2 = new Thread(synTest2,"thread2");
        thread1.start();
        thread2.start();
    }
    /**
     * 输出结果
     thread1:0
     thread1:1
     thread1:2
     thread1:3
     thread1:4
     thread2:5
     thread2:6
     thread2:7
     thread2:8
     thread2:9
     */
}
```

可以看出虽然后两个实例对象，分别调用但依然保持了线程同步。因为synchronized修饰的是静态方法，不属于当前实例，而是属于类。 注意：当一个线程A调用一个实例对象的非static synchronized方法，则线程B调用这个实例对象所属类的static synchronized方法并不会发生互斥，是允许的。因为访问静态synchronized方法占用的锁是当前类的class对象，而非静态synchronized方法占用的是当前实例对象锁。

8.3 作用于代码块

使用场景：在某些情况下我们的方法体会比较大，存在一些比较耗时的操作，但需要同步的操作又只有一小部分，所以我们可以通过使用同步代码块的方法对需要同步的代码进行包裹，这样无需对整个代码块进行同步操作。所以同步代码块的作用范围为synchronized(object){}

```
public class SynTest implements Runnable{
    static int count = 0;
    private byte[] mByte = new byte[0];

    @Override
    public synchronized void run(){
        increaseCount();
    }
```

```
    private void increaseCount(){
        //省略耗时的其他操作
        synchronized(mybyte){
            for(int i=0;i<5;i++){
                System.out.println(Thread.currentThread().getName()+" "+count++);
                try{
                    Thread.sleep(1000);
                }catch(InterruptedException){
                    e.printStackTrack();
                }
            }
        }
    }
    public static void main(String[] args){
        SynTest synTest1 = new SynTest();
        SynTest synTest2 = new SynTest();
        Thread thread1 = new Thread(synTest1,"thead1");
        Thread thread2 = new Thread(synTest2,"thread2");
        thread1.start();
        thread2.start();
    }
}
```

上面代码中产生了两个实力，分别调用各自代码块，县城并不是安全的。如果要线程安全，如要在一个实例中进行多线程调用。

9 死锁

9.1 什么死锁

并发执行带来的问题死锁，死锁指的是：两个或两个以上进程（线程）在运行时争夺资源而造成的一种僵局，若无外力，这些进程（线程）将无法向前推进。 饥饿：指一个进程一直得不到资源。 饥饿和死锁都是由于进程竞争资源而引起的。饥饿一般不占有资源，死锁进程一定占有资源。

9.2 资源类型

可抢占资源：进程在得到这类资源后，该资源可以被其他进程抢占。这类资源不会引起死锁。CPU和主存属于可抢占资源。 不可抢占资源：一旦系统把该类资源分配某个进程后，不能将它强行收回，只能进程用完后释放。磁带机，打印机属于不可抢占资源。

9.3 死锁产生的原因

- 竞争不可抢夺资源引起死锁 通常系统的不可抢夺资源的数量不满足多个进程运行的需求，是的进程在运行中因抢夺资源而陷入僵局。
- 竞争可消耗资源引起死锁
- 进程推进顺序不当引起死锁 多个进程不是竞争同一个资源，而是等待对方的资源而导致死锁。

9.4 死锁产生的必要条件

- 互斥条件 进程对所分配的资源进行排他性控制，及一段时间内一个资源只能被一个进程占用。其他进程只能等待。
- 不可剥夺条件 进程获得资源在未使用完前，不能被其他进程强行夺走。只能自己主动释放。
- 请求与保持条件 进程在有用资源的同时，需要新的资源请求，但该资源被其他进程占有，此时该进程被阻塞等待，同时也不会释放自己的资源。
- 循环等待条件 存在一种进程资源的循环等待链，链中每一个进程获得资源同时被链中下一个进程请求。

以上是死锁产生的必要条件，即如果系统发生是死锁，这些条件一定成立。只要上述条件一个不成立就不会发生死锁。

9.5 处理死锁的方法

- 预防死锁：通过限制条件，破坏产生死锁的四个必要条件之一或几个，来防止死锁的产生。
- 避免死锁：在资源的动态分配过程中，通过某种策略防止系统进入不安全状态，避免死锁的发生。
- 检测死锁：允许系统的运行过程中产生死锁，但可设置检测方法并及时检测死锁的发生，并采取适当的措施清除。
- 解除死锁：当检测出死锁后，采用适当的方法清除死锁。

9.6 死锁的处理经验

（1）对于java程序员来说最简单的防止死锁的方式是对竞争的资源引入序号，如果一个进程需要几个资源，那么他必须先得到小序号的资源，再申请大序号的资源。可以在java代码中增加同步关键字的使用。 （2）确保在峰值并发时有足够大的资源池。 （3）避免执行数据库调用或在占有java虚拟机锁时，执行其他与java虚拟机无关的操作。

10 虚拟内存

10.1 什么是虚拟内存

在系统中所有进程是共享CPU和主存这些资源的。当进程数较多，所需要的资源也会增加，可能会导致部分程序没有主存空间。此外，资源是共享，可能会导致某个进程重写了另外一个进程的内存，导致程序运行逻辑错误。 所以为了更加有效的管理内存，系统提供了一种对主存的抽象概念，虚拟内存（VM）。它为每一个进程提供了一个大的、一致的和私有的地址空间。 基于局部性原理，在程序装入时，可以将程序的一部分装入内存，而将其他部分留在外存，就可以启动程序执行。在程序执行时，当访问的资源不再内存中时，可以将其调入内存。另一方面，程序将内存中暂时不使用的资源换出到外存中。这样系统好像提供了一个比实际内存大的多的存储器，称为虚拟存储器。 虚拟内存实际并不存在，只是由系统提供了部分装入、请求调如何置换功能对用户透明后，给用户的感觉变大。 虚拟内存提供了三个重要的能力：缓存，内存管理，内存保护。

10.2 分页系统实现虚拟内存原理

在请求分页系统中，只要将当前需要的一部分页面转装入内存就可以启动作业。在作业过程中，当访问的页面不在内存中时，在通过页面置换算法调换页面。

10.3 页面置换算法

- 先进先出页面置换算法（FIFO） 优先淘汰最早进入内存的页面，亦即在内存中驻留时间最久的页面。该算法实现简单，只需把调入内存的页面根据先后次序链接成队列，设置一个指针总指向最早的页面。

- 最近最久未使用置换算法（LRU） 选择最长时间没有访问的页面进行淘汰。 一般使用LinkedHashMap来实现，LinkedHashMap底层是双向链表+HashMap实现的，本身实现了一个方法removeEldestEntry用于判断是否需要移除最不常用的数，方法默认返回false所以需要重写该方法。

```
public class LRU<K,V>{
    private static final float hashLoadFactory = 0.75f;
    private LinkedHashMap<K,V> map;
    private int cacheSize;
    public LRU(int cacheSize){
        this.cacheSize = cacheSize;
        int capacity = (int)Math.ceil(cacheSize/hashLoadFactory)+1;
        map = new LinkedHashMap<K,V>(capacity,hashLoadFactory,true){
            private static final long serialVersionUID = 1L;
            @Override
            protected boolean removeEldestEntry(Map.Entry eldest){
                return size()>LRU.this.cacheSize;
            }
        };
    }
    public synchronized V get(K key){
        return map.get(key);
    }
    public synchronized void put(K key,V value){
        map.put(key,value);
    }
}
```

11 分页管理和分段管理

11.1 概念

- 分页存储管理是将一个进程的逻辑地址空间分成若干大小相等的片，称为页面或者页，并加以编号。相应的也罢内存空间分成与页面相同大小的若干个存储块，称为物理块或页框，同样进行编号。在为进程分配内存时，将以块为单位的进程中的若干个页面分别装入到多个不相邻的物理页中。
- 分段存储管理，作业的地址空间被划分为若干个段，每个段定义了一组逻辑信息。例如有助程序段MAIN，子程序段X，数据段D等。每个段都有自己的名字，通常可以用短号代替名字，每个段从0开始编址，并采用一段连续的地址空间。段的长度有相应的逻辑信息组的长度决定，因而个段的长度不同。某逻辑地址有段号和段内地址所组成。

11.2 区别

（1）页是信息的物理存储单位，分页是为了实现离散的分配方式，以消减内存的碎片，提高内存的利用率。分页仅仅是系统管理的需要并不是用户需求；段是信息的逻辑单位，它是一组意义相对完整的信息，分段的目的是为了更好的满足用户的需求。 （2）页的大小固定且有系统决定，有系统把逻辑地址个划分为页号和页内地址两部分，由机器硬件实现，且系统中只能有一种大小的页面；段的长度不固定，取决于用户编写程序的逻辑，通常有编译程序对源程序进程编译时，根据信息的性质划分。 （3）分页式存储的作业地址空间是一维

的，即单一的线性逻辑地址空间，程序员只需要一个记忆符表示一个地址空间；分段式存储的地址空间是二维的，标识一个地址需要给出段名和段内地址。

12 动态链接和静态链接

12.1 静态链接

- 优点：（1）代码装载速度快，执行速度略比动态链接快；（2）只需要开发计算机里有正确的.LIB文件，在以二进制发布程序时不考虑用户计算机上的.LIB文件是否存在版本问题，可避免DLL地狱等问题。
- 不足：（1）使用静态链接生成的可执行文件体积较大，包含相同的公共代码，造成浪费。

12.2 动态链接

- 优点：（1）节省内存并减少页面交换；（2）DLL文件和EXE文件独立，只要输出接口不变（名称、参数、返回值类型和调用约定不变），更换DLL文件不会对EXE文件造成影响，极大的提高了可维护性和可执行性；（3）不同编程语言编写的程序只要安照函数调用约定就可以调用统一个DLL函数；（4）使用于大规模的软件开发，是开发过程独立、耦合度小，便于不同开发者和组织之间进行开发和测试。
- 不足：（1）使用动态链接的应用程序不是自完备的，它以来的DLL模块也要存在，如果使用载入时动态链接，程序启动时大仙DLL不在，系统将会终止应用并给出错误信息。而使用运行时动态链接，系统不会终止，但由于DLL的到处函数不可用会导致程序加载失败；（2）速度比静态加载慢，如果新模块和旧模块不兼容，那么需要该模块运行的软件统统不能用。