4种常用Java线程锁的特点，性能比较、使用场景

题目标签

学习时长：20分钟

题目难度：中等

知识点标签：synchronized、ReentrantLock、Semaphore、AtomicInteger

题目描述

4中常用Java线程锁的特点，性能比较及使用场景

1. 面试题分析

根据题目要求我们可以知道：

- 多线程的缘由
- 多线程并发面临的问题
- 4种Java线程锁(线程同步)
- Java线程锁总结

> 分析需要全面并且有深度

容易被忽略的坑

- 分析片面
- 没有深入

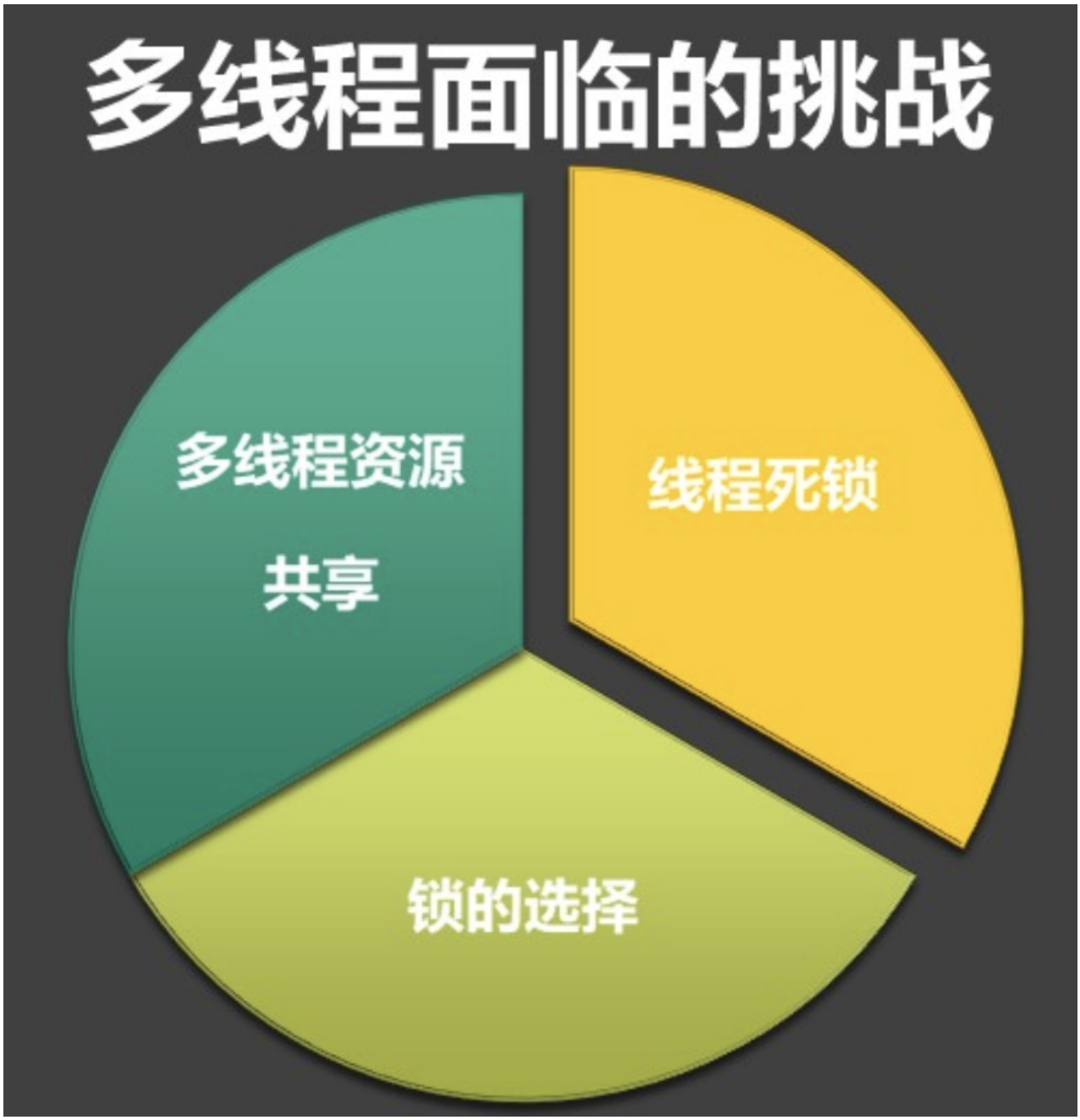
2. 多线程的缘由

在出现了进程之后，操作系统的性能得到了大大的提升。虽然进程的出现解决了操作系统的并发问题，但是人们仍然不满足，人们逐渐对实时性有了要求。

使用多线程的理由之一是和进程相比，它是一种非常**花销小，切换快**，更”节俭”的多任务操作方式。

在Linux系统下，启动一个新的进程必须分配给它独立的地址空间，建立众多的数据表来维护它的代码段、堆栈段和数据段，这是一种”昂贵”的多任务工作方式。而在进程中的同时运行多个线程，它们彼此之间使用相同的地址空间，共享大部分数据，启动一个线程所花费的空间远远小于启动一个进程所花费的空间，而且，线程间彼此切换所需的时间也远远小于进程间切换所需要的时间。

3. 多线程并发面临的问题



由于多个线程是**共同占有所属进程的资源和地址空间**的，那么就会存在一个问题：

如果多个线程要同时访问某个资源，怎么处理?

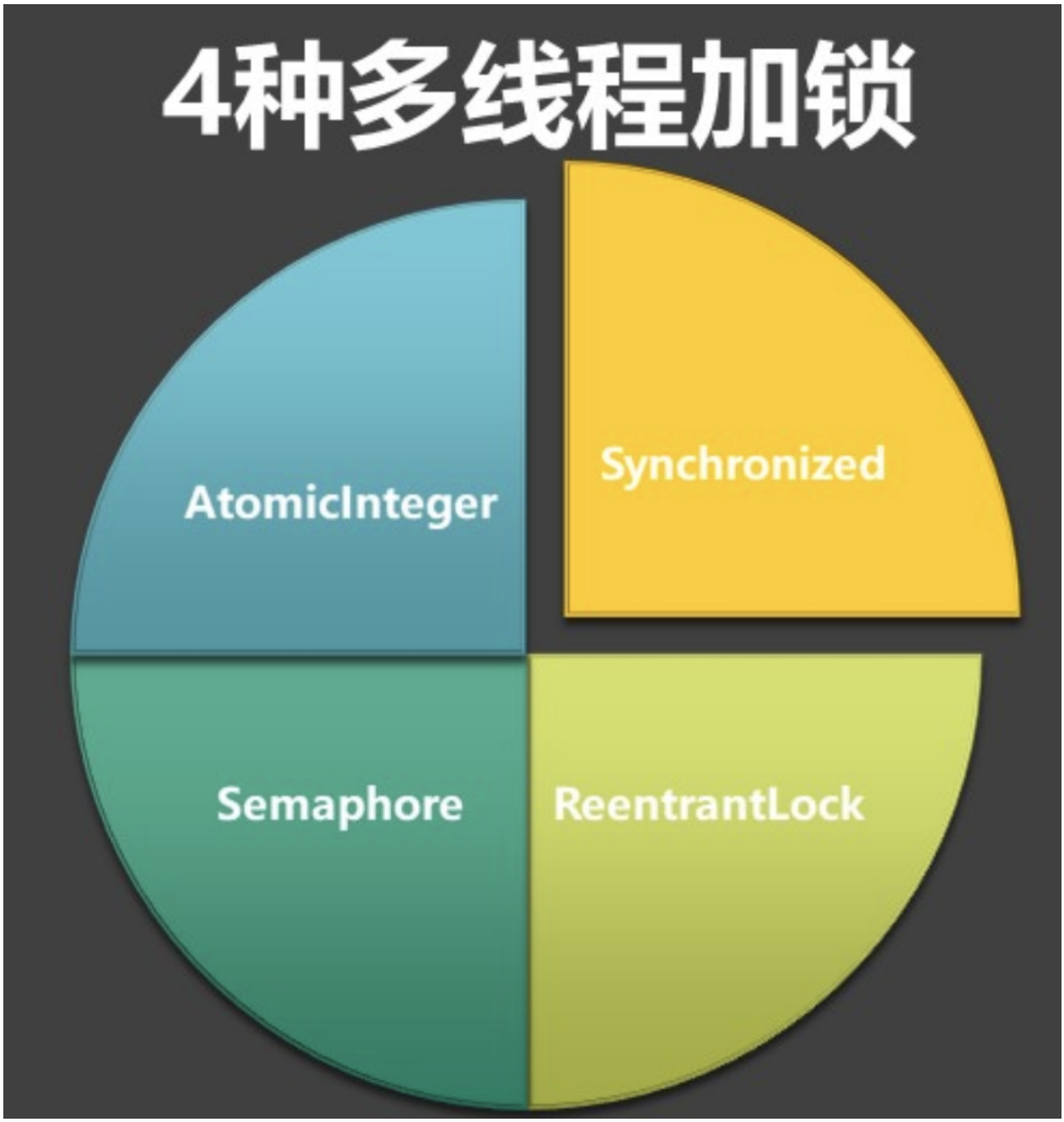
在Java并发编程中，经常遇到多个线程访问同一个 共享资源 ，这时候作为开发者必须考虑如何维护数据一致性，这就是Java锁机制(同步问题)的来源。

Java提供了多种多线程锁机制的实现方式，常见的有：

1. synchronized
2. ReentrantLock
3. Semaphore
4. AtomicInteger等

每种机制都有优缺点与各自的适用场景，必须熟练掌握他们的特点才能在Java多线程应用开发时得心应手。

4. 4种Java线程锁(线程同步)



1.synchronized

在Java中synchronized关键字被常用于维护数据一致性。

synchronized机制是给共享资源上锁，只有拿到锁的线程才可以访问共享资源，这样就可以强制使得对共享资源的访问都是顺序的。

Java开发人员都认识synchronized，使用它来实现多线程的同步操作是非常简单的，只要在需要同步的对方的**方法、类或代码块**中加入该关键字，它能够保证在同一个时刻最多只有一个线程执行同一个对象的同步代码，可保证修饰的代码在执行过程中不会被其他线程干扰。使用synchronized修饰的代码具有原子性和可见性，在需要进程同步的程序中使用的频率非常高，可以满足一般的进程同步要求。

```
synchronized (obj) {
//方法
.....
}
```

synchronized实现的机理依赖于软件层面上的JVM，因此其性能会随着Java版本的不断升级而提高。

到了Java1.6，synchronized进行了很多的优化，有适应自旋、锁消除、锁粗化、轻量级锁及偏向锁等，效率有了本质上的提高。在之后推出的Java1.7与1.8中，均对该关键字的实现机理做了优化。

需要说明的是，当线程通过synchronized等待锁时是不能被Thread.interrupt()中断的，因此程序设计时必须检查确保合理，否则可能会造成线程死锁的尴尬境地。

最后，尽管Java实现的锁机制有很多种，并且有些锁机制性能也比synchronized高，但还是强烈推荐在多线程应用程序中使用该关键字，因为实现方便，后续工作由JVM来完成，可靠性高。只有在确定锁机制是当前多线程程序的性能瓶颈时，才考虑使用其他机制，如ReentrantLock等。

2.ReentrantLock

可重入锁，顾名思义，这个锁可以被线程多次重复进入进行获取操作。

ReentantLock继承接口Lock并实现了接口中定义的方法，除了能完成synchronized所能完成的所有工作外，还提供了诸如可响应中断锁、可轮询锁请求、定时锁等避免多线程死锁的方法。

Lock实现的机理依赖于特殊的CPU指定，可以认为不受JVM的约束，并可以通过其他语言平台来完成底层的实现。在并发量较小的多线程应用程序中，ReentrantLock与synchronized性能相差无几，但在高并发量的条件下，synchronized性能会迅速下降几十倍，而ReentrantLock的性能却能依然维持一个水准。

因此我们建议在高并发量情况下使用ReentrantLock。

ReentrantLock引入两个概念：**公平锁与非公平锁**。

公平锁指的是锁的分配机制是公平的，通常先对锁提出获取请求的线程会先被分配到锁。反之，JVM按随机、就近原则分配锁的机制则称为不公平锁。

ReentrantLock在构造函数中提供了是否公平锁的初始化方式，默认为非公平锁。这是因为，非公平锁实际执行的效率要远远超出公平锁，除非程序有特殊需要，否则最常用非公平锁的分配机制。

ReentrantLock通过方法lock()与unlock()来进行加锁与解锁操作，与synchronized会被JVM自动解锁机制不同，ReentrantLock加锁后需要手动进行解锁。为了避免程序出现异常而无法正常解锁的情况，**使用ReentrantLock必须在finally控制块中进行解锁操作**。通常使用方式如下所示：

```
Lock lock = new ReentrantLock();
try {
lock.lock();
//...进行任务操作5 }
finally {
lock.unlock();
}
```

3.Semaphore

上述两种锁机制类型都是“互斥锁”，学过操作系统的都知道，互斥是进程同步关系的一种特殊情况，相当于只存在一个临界资源，因此同时最多只能给一个线程提供服务。但是，在实际复杂的多线程应用程序中，可能存在多个临界资源，这时候我们可以借助Semaphore信号量来完成多个临界资源的访问。

Semaphore基本能完成ReentrantLock的所有工作，使用方法也与之类似，通过acquire()与release()方法来获得和释放临界资源。

经实测，Semaphone.acquire()方法默认为可响应中断锁，与ReentrantLock.lockInterruptibly()作用效果一致，也就是说在等待临界资源的过程中可以被Thread.interrupt()方法中断。

此外，Semaphore也实现了可轮询的锁请求与定时锁的功能，除了方法名tryAcquire与tryLock不同，其使用方法与ReentrantLock几乎一致。Semaphore也提供了公平与非公平锁的机制，也可在构造函数中进行设定。

Semaphore的锁释放操作也由手动进行，因此与ReentrantLock一样，为避免线程因抛出异常而无法正常释放锁的情况发生，**释放锁的操作也必须在finally代码块中完成**。

4.AtomicInteger

首先说明，此处AtomicInteger是一系列相同类的代表之一，常见的还有AtomicLong、AtomicLong等，他们的实现原理相同，区别在与运算对象类型的不同。

我们知道，在多线程程序中，诸如++i
或
i++等运算不具有原子性，是不安全的线程操作之一。通常我们会使用synchronized将该操作变成一个原子操作，但JVM为此类操作特意提供了一些同步类，使得使用更方便，且使程序运行效率变得更高。通过相关资料显示，通常AtomicInteger的性能是ReentantLock的好几倍。

5. Java线程锁总结

1.synchronized：

在资源竞争不是很激烈的情况下，偶尔会有同步的情形下，synchronized是很合适的。原因在于，编译程序通常会尽可能的进行优化synchronize，另外可读性非常好。

2.ReentrantLock:

在资源竞争不激烈的情形下，性能稍微比synchronized差点点。但是当同步非常激烈的时候，synchronized的性能一下子能下降好几十倍，而ReentrantLock确还能维持常态。

高并发量情况下使用ReentrantLock。

3.Atomic:

和上面的类似，不激烈情况下，性能比synchronized略逊，而激烈的时候，也能维持常态。激烈的时候，Atomic的性能会优于ReentrantLock一倍左右。但是其有一个缺点，就是只能同步一个值，一段代码中只能出现一个Atomic的变量，多于一个同步无效。因为他不能在多个Atomic之间同步。

所以，我们写同步的时候，优先考虑synchronized，如果有特殊需要，再进一步优化。ReentrantLock和Atomic如果用的不好，不仅不能提高性能，还可能带来灾难。

以上就是Java线程锁的详解，除了从编程的角度应对高并发，更多还需要从架构设计的层面来应对高并发场景，例如：Redis缓存、CDN、异步消息等，详细的内容如下。

6. 扩展内容

- AQS的实现原理
- 线程池的实现原理、优点与风险、以及四种线程池实现
- CountDownLatch、Semaphore等4大并发工具类详解