关注公众号，获取更多 BAT 面试题答案和架构干货

一、java 集合类必考题目

Java 集合类框架的最佳实践有哪些?

根据应用的需要正确选择要使用的集合的类型对性能非常重要，比如：假如元素的数量是固定的，而且能事先知道，我们就应该用 Array 而不是 ArrayList。

有些集合类允许指定初始容量。因此，如果我们能估计出存储的元素的数目，我们可以设置初始容量来避免重新计算 hash 值或者是扩容。

为了类型安全，可读性和健壮性的原因总是要使用泛型。同时，使用泛型还可以避免运行时的 ClassCastException。

使用 JDK 提供的不变类(immutable class)作为 Map 的键可以避免为我们自己的类实现 hashCode()和 equals()方法。

编程的时候接口优于实现。

底层的集合实际上是空的情况下，返回长度是 0 的集合或者是数组，不要返回 null。

HashMap、HashTable 和 currentHashMap 的区别及优缺点

HashMap：线程不安全，所以性能高，可以通过继承 collection 来调用方法实现线程安全。

Hashtable:线程安全

concurrentHashMap:线程安全的，在多线程下效率更高。、

注：hashtable:使用一把锁处理并发问题，当有多个线程访问时，需要多个线程竞争一把锁，导致阻塞。

concurrentHashMap 则使用分段，相当于把一个 hashmap 分成多个，然后每个部分分配一把锁，这样就可以支持多线程访问。

HashSet 和 TreeSet 有什么区别?

HashSet 是由一个 hash 表来实现的，因此，它的元素是无序的。add()，remove()，contains()方法的时间复杂度是 O(1)。

另一方面，TreeSet 是由一个树形的结构来实现的，它里面的元素是有序的。因此，add()，remove()，contains()方法的时间复杂度是 O(logn)。

ConcurrentHashMap 和 Hashtable 的区别

Hashtable 和 ConcurrentHashMap 有什么分别呢？它们都可以用于多线程的环境，但是当 Hashtable 的大小增加到一定的时候，性能会急剧下降，因为迭代时需要被锁定很长的时间。因为 ConcurrentHashMap 引入了分割(segmentation)，不论它变得多么大，仅仅需要锁定 map 的某个部分，而其它的线程不需要等到迭代完成才能访问 map。简而言之，在迭代的过程中，ConcurrentHashMap 仅仅锁定 map 的某个部分，而 Hashtable 则会锁定整个 map。

HashMap 的工作原理是近年来常见的 Java 面试题。几乎每个 Java 程序员都知道 HashMap，都知道哪里要用 HashMap，知道 Hashtable 和 HashMap 之间的区别，那么为何这道面试题如此特殊呢？是因为这道题考察的深度很深。这题经常出现在高级或中高级面试中。

Java 中的 HashMap 的工作原理是什么?

Java 中的 HashMap 是以键值对(key-value)的形式存储元素的。HashMap 需要一个 hash 函数，它使用 hashCode()和 equals()方法来向集合/从集合添加和检索元素。当调用 put()方法的时候，HashMap 会计算 key 的 hash 值，然后把键值对存储在集合中合适的索引上。如果 key 已经存在了，value 会被更新成新值。HashMap 的一些重要的特性是它的容量(capacity)，负载因子(load factor)和扩容极限(threshold resizing)。

hashCode()和 equals()方法的重要性体现在什么地方?

Java 中的 HashMap 使用 hashCode()和 equals()方法来确定键值对的索引，当根据键获取值的时候也会用到这两个方法。如果没有正确的实现这两个方法，两个不同的键可能会有相同的 hash 值，因此，可能会被集合认为是相等的。而且，这两个方法也用来发现重复元素。所以这两个方法的实现对 HashMap 的精确性和正确性是至关重要的。

HashMap 和 Hashtable 有什么区别?

HashMap 和 Hashtable 都实现了 Map 接口，因此很多特性非常相似。但是，他们有以下不同点：

HashMap 允许键和值是 null，而 Hashtable 不允许键或者值是 null。

Hashtable 是同步的，而 HashMap 不是。因此，HashMap 更适合于单线程环境，而 Hashtable 适合于多线程环境。

HashMap 提供了可供应用迭代的键的集合，因此，HashMap 是快速失败的。另一方面，Hashtable 提供了对键的列举(Enumeration)。

一般认为 Hashtable 是一个遗留的类。

数组(Array)和列表(ArrayList)有什么区别？什么时候应该使用 Array 而不是 ArrayList?

下面列出了 Array 和 ArrayList 的不同点：

Array 可以包含基本类型和对象类型，ArrayList 只能包含对象类型。

Array 大小是固定的，ArrayList 的大小是动态变化的。

ArrayList 提供了更多的方法和特性，比如：addAll()，removeAll()，iterator()等等。

对于基本类型数据，集合使用自动装箱来减少编码工作量。但是，当处理固定大小的基本数据类型的时候，这种方式相对比较慢。

ArrayList 和 LinkedList 有什么区别?

ArrayList 和 LinkedList 都实现了 List 接口，他们有以下的不同点：

ArrayList 是基于索引的数据接口，它的底层是数组。它可以以 O(1)时间复杂度对元素进行随机访问。与此对应，LinkedList 是以元素列表的形式存储它的数据，

每一个元素都和它的前一个和后一个元素链接在一起，在这种情况下，查找某个元素的时间复杂度是 O(n)。

相对于 ArrayList，LinkedList 的插入，添加，删除操作速度更快，因为当元素被添加到集合任意位置的时候，不需要像数组那样重新计算大小或者是更新索引。

LinkedList 比 ArrayList 更占内存，因为 LinkedList 为每一个节点存储了两个引用，一个指向前一个元素，一个指向下一个元素。

也可以参考 ArrayList vs. LinkedList。

一、 Java 线程池使用说明

一简介

线程的使用在 java 中占有极其重要的地位，在 jdk1.4 极其之前的 jdk 版本中，关于线程池的使用是极其简陋的。在 jdk1.5 之后这一情况有了很大的改观。Jdk1.5 之后加入了 java.util.concurrent 包，这个包中主要介绍 java 中线程以及线程池的使用。为我们在开发中处理线程的问题提供了非常大的帮助。

二：线程池

线程池的作用：

线程池作用就是限制系统中执行线程的数量。

根据系统的环境情况，可以自动或手动设置线程数量，达到运行的最佳效果；少了

浪费了系统资源，多了造成系统拥挤效率不高。用线程池控制线程数量，其他线程排队等候。一个任务执行完毕，再从队列的中取最前面的任务开始执行。若队列中没有等待进程，线程池的这一资源处于等待。当一个新任务需要运行时，如果线程池中有等待的工作线程，就可以开始运行了；否则进入等待队列。

为什么要用线程池:

1.减少了创建和销毁线程的次数，每个工作线程都可以被重复利用，可执行多个任务。

2.可以根据系统的承受能力，调整线程池中工作线线程的数目，防止因为消耗过多的内存，而把服务器累趴下(每个线程需要大约 1MB 内存，线程开的越多，消耗的内存也就越大，最后死机)。

Java 里面线程池的顶级接口是 Executor，但是严格意义上讲 Executor 并不是一个线程池，而只是一个执行线程的工具。真正的线程池接口是 ExecutorService。

比较重要的几个类：

ExecutorService	真正的线程池接口。
ScheduledExecutorService	能和 Timer/TimerTask 类似，解决那些需要任务重复执行的问题。
ThreadPoolExecutor	ExecutorService 的默认实现。
ScheduledThreadPoolExecutor	继承 ThreadPoolExecutor 的 ScheduledExecutorService 接口实现，周期性任务调度的类实现。

要配置一个线程池是比较复杂的，尤其是对于线程池的原理不是很清楚的情况下，很有可能配置的线程池不是较优的，因此在 Executors 类里面提供了一些静态工厂，生成一些常用的线程池。

1. newSingleThreadExecutor

创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。

2.newFixedThreadPool

创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。

3. newCachedThreadPool

创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，那么就会回收部分空闲（60 秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说 JVM）能够创建的最大线程大小。

4.newScheduledThreadPool

创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。

实例

1：newSingleThreadExecutor

MyThread.java

```
Public class MyThread extends Thread {
    @Override
    publicvoid run() {
        System.out.println(Thread.currentThread().getName() + "正在执行。。。");
    }
}
```

TestSingleThreadExecutor.java

```
Public class TestSingleThreadExecutor {
    publicstaticvoid main(String[] args) {
        //创建一个可重用固定线程数的线程池
        ExecutorService pool = Executors. newSingleThreadExecutor();
        //创建实现了 Runnable 接口对象，Thread 对象当然也实现了 Runnable 接口
```

```
        Thread t1 = new MyThread();
        Thread t2 = new MyThread();
        Thread t3 = new MyThread();
        Thread t4 = new MyThread();
        Thread t5 = new MyThread();
        //将线程放入池中进行执行
        pool.execute(t1);
        pool.execute(t2);
        pool.execute(t3);
        pool.execute(t4);
        pool.execute(t5);
        //关闭线程池
        pool.shutdown();
    }
}
```

输出结果

```
pool-1-thread-1 正在执行。。。
pool-1-thread-1 正在执行。。。
pool-1-thread-1 正在执行。。。
pool-1-thread-1 正在执行。。。
pool-1-thread-1 正在执行。。。
```

2newFixedThreadPool

TestFixedThreadPool.Java

```
publicclass TestFixedThreadPool {
    publicstaticvoid main(String[] args) {
        //创建一个可重用固定线程数的线程池
        ExecutorService pool = Executors.newFixedThreadPool(2);
        //创建实现了 Runnable 接口对象，Thread 对象当然也实现了 Runnable 接口
        Thread t1 = new MyThread();
        Thread t2 = new MyThread();
        Thread t3 = new MyThread();
        Thread t4 = new MyThread();
        Thread t5 = new MyThread();
        //将线程放入池中进行执行
```

```
        pool.execute(t1);
        pool.execute(t2);
        pool.execute(t3);
        pool.execute(t4);
        pool.execute(t5);
        //关闭线程池
        pool.shutdown();
    }
}
```

输出结果

```
pool-1-thread-1 正在执行。。。
pool-1-thread-2 正在执行。。。
pool-1-thread-1 正在执行。。。
pool-1-thread-2 正在执行。。。
pool-1-thread-1 正在执行。。。
```

3 newCachedThreadPool

TestCachedThreadPool.java

```
publicclass TestCachedThreadPool {
    publicstaticvoid main(String[] args) {
        //创建一个可重用固定线程数的线程池
        ExecutorService pool = Executors.newCachedThreadPool();
        //创建实现了 Runnable 接口对象，Thread 对象当然也实现了 Runnable 接口
        Thread t1 = new MyThread();
        Thread t2 = new MyThread();
        Thread t3 = new MyThread();
        Thread t4 = new MyThread();
        Thread t5 = new MyThread();
        //将线程放入池中进行执行
        pool.execute(t1);
        pool.execute(t2);
        pool.execute(t3);
        pool.execute(t4);
        pool.execute(t5);
```

```
        //关闭线程池
        pool.shutdown();
    }
}
```

输出结果：

```
pool-1-thread-2 正在执行。。。
pool-1-thread-4 正在执行。。。
pool-1-thread-3 正在执行。。。
pool-1-thread-1 正在执行。。。
pool-1-thread-5 正在执行。。。
```

4newScheduledThreadPool

TestScheduledThreadPoolExecutor.java

```
publicclass TestScheduledThreadPoolExecutor {
    publicstaticvoid main(String[] args) {
        ScheduledThreadPoolExecutor exec = new ScheduledThreadPoolExecutor(1);
        exec.scheduleAtFixedRate(new Runnable() {//每隔一段时间
                    @Override
                    publicvoid run() {
                        //throw new RuntimeException();
                        System.out.println("================");
                    }
                }, 1000, 5000, TimeUnit.MILLISECONDS);
        exec.scheduleAtFixedRate(new Runnable() {//每隔一段时间打印系统时间，证明两者是互不影响的
                    @Override
                    publicvoid run() {
                        System.out.println(System.nanoTime());
                    }
                }, 1000, 2000, TimeUnit.MILLISECONDS);
    }
}
```

输出结果

```
================
8384644549516
```

```
8386643829034
8388643830710
================
8390643851383
8392643879319
8400643939383
```

三：ThreadPoolExecutor 详解

ThreadPoolExecutor 的完整构造方法的签名是：

ThreadPoolExecutor(int corePoolSize, int maximumPoolSize,

long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory, RejectedExecutionHandler handler) .

corePoolSize - 池中所保存的线程数，包括空闲线程。

maximumPoolSize-池中允许的最大线程数。

keepAliveTime - 当线程数大于核心时，此为终止前多余的空闲线程等待新任务的最长时间。

unit - keepAliveTime 参数的时间单位。

workQueue - 执行前用于保持任务的队列。此队列仅保持由 execute 方法提交的 Runnable 任务。

threadFactory - 执行程序创建新线程时使用的工厂。

handler - 由于超出线程范围和队列容量而使执行被阻塞时所使用的处理程序。

ThreadPoolExecutor 是 Executors 类的底层实现。

在 JDK 帮助文档中，有如此一段话：

“强烈建议程序员使用较为方便的 Executors 工厂方法

Executors.newCachedThreadPool()（无界线程池，可以进行自动线程回

收）、Executors.newFixedThreadPool(int)（固定大小线程池）

Executors.newSingleThreadExecutor()（单个后台线程）

它们均为大多数使用场景预定义了设置。”

下面介绍一下几个类的源码：

ExecutorService newFixedThreadPool (int nThreads):固定大小线程池。

可以看到，corePoolSize 和 maximumPoolSize 的大小是一样的（实际上，后面会介绍，如果使用无界 queue 的话 maximumPoolSize 参数是没有意义的），keepAliveTime 和 unit 的设值表名什么？-就是该实现不想 keep alive！最后的 BlockingQueue 选择了 LinkedBlockingQueue，该 queue 有一个特点，他是无界的。

```
1.  public static ExecutorService newFixedThreadPool(int nThreads) {
2.      return new ThreadPoolExecutor(nThreads, nThreads,
3.                                    0L, TimeUnit.MILLISECONDS,
4.                                    new LinkedBlockingQueue<Runnable>());
5.  }
```

ExecutorService newSingleThreadExecutor()：单线程

```
1.  public static ExecutorService newSingleThreadExecutor() {
2.      return new FinalizableDelegatedExecutorService
3.          (new ThreadPoolExecutor(1, 1,
4.                                  0L, TimeUnit.MILLISECONDS,
5.                                  new LinkedBlockingQueue<Runnable>()));
6.  }
```

ExecutorService newCachedThreadPool()：无界线程池，可以进行自动线程回收

这个实现就有意思了。首先是无界的线程池，所以我们可以发现 maximumPoolSize 为 big big。其次 BlockingQueue 的选择上使用 SynchronousQueue。可能对于该 BlockingQueue 有些陌生，简单说：该 QUEUE 中，每个插入操作必须等待另一个线程的对应移除操作。

```
1.  public static ExecutorService newCachedThreadPool() {
2.          return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
3.                                        60L, TimeUnit.SECONDS,
4.                                        new SynchronousQueue<Runnable>());
5.      }
```

先从 BlockingQueue<Runnable> workQueue 这个入参开始说起。在 JDK 中，其实已经说得很清楚了，一共有三种类型的 queue。

所有 BlockingQueue 都可用于传输和保持提交的任务。可以使用此队列与池大小进行交互：

如果运行的线程少于 corePoolSize，则 Executor 始终首选添加新的线程，而不进行排队。(如果当前运行的线程小于 corePoolSize，则任务根本不会存放，添加到 queue 中，而是直接抄家伙（thread）开始运行)

如果运行的线程等于或多于 corePoolSize，则 Executor 始终首选将请求加入队列，**而不添加新的线程**。

如果无法将请求加入队列，则创建新的线程，除非创建此线程超出 maximumPoolSize，在这种情况下，任务将被拒绝。

queue 上的三种类型。

排队有三种通用策略：

直接提交。工作队列的默认选项是 SynchronousQueue，它将任务直接提交给线程而不保持它们。在此，如果不存在可用于立即运行任务的线程，则试图把任务加入队列将失败，因此会构造一个新的线程。此策略可以避免在处理可能具有内部依赖性的请求集时出现锁。直接提交通常要求无界 maximumPoolSizes 以避免拒绝新提交的任务。

当命令以超过队列所能处理的平均数连续到达时，此策略允许无界线程具有增长的可能性。

无界队列。使用无界队列（例如，不具有预定义容量的 LinkedBlockingQueue）将导致在所有 corePoolSize 线程都忙时新任务在队列中等待。这样，创建的线程就不会超过 corePoolSize。（因此，maximumPoolSize 的值也就无效了。）当每个任务完全独立于其他任务，即任务执行互不影响时，适合于使用无界队列；例如，在 Web 页服务器中。这种排队可用于处理瞬态突发请求，当命令以超过队列所能处理的平均数连续到达时，此策略允许无界线程具有增长的可能性。

有界队列。当使用有限的 maximumPoolSizes 时，有界队列（如 ArrayBlockingQueue）有助于防止资源耗尽，但是可能较难调整和控制。队列大小和最大池大小可能需要相互折衷：使用大型队列和小型池可以最大限度地降低 CPU 使用率、操作系统资源和上下文切换开销，但是可能导致人工降低吞吐量。如果任务频繁阻塞（例如，如果它们是 I/O 边界），则系统可能为超过您许可的更多线程安排时间。使用小型队列通常要求较大的池大小，CPU 使用率较高，但是可能遇到不可接受的调度开销，这样也会降低吞吐量。

BlockingQueue 的选择。

例子一：使用直接提交策略，也即 SynchronousQueue。

首先 SynchronousQueue 是无界的，也就是说他存数任务的能力是没有限制的，但是由于该 Queue 本身的特性，**在某次添加元素后必须等待其他线程取走后才能继续添加。**在这里不是核心线程便是新创建的线程，但是我们试想一样下，下面的场景。我们使用一下参数构造 ThreadPoolExecutor：

```
1.  new ThreadPoolExecutor(
2.          2, 3, 30, TimeUnit.SECONDS,
```

```
3.              new   SynchronousQueue<Runnable>(),
4.              new RecorderThreadFactory("CookieRecorderPool"),
5.              new ThreadPoolExecutor.CallerRunsPolicy());
```

```
new ThreadPoolExecutor(
 2, 3, 30, TimeUnit.SECONDS,
 new SynchronousQueue<Runnable>(),
 new RecorderThreadFactory("CookieRecorderPool"),
 new ThreadPoolExecutor.CallerRunsPolicy());
```

当核心线程已经有 2 个正在运行.

1. 此时继续来了一个任务（A），根据前面介绍的“如果运行的线程等于或多于 corePoolSize，则 Executor 始终首选将请求加入队列，**而不添加新的线程**。”,所以 A 被添加到 queue 中。
2. 又来了一个任务（B），且核心 2 个线程还没有忙完，OK，接下来首先尝试 1 中描述，但是由于使用的 SynchronousQueue，所以一定无法加入进去。
3. 此时便满足了上面提到的“如果无法将请求加入队列，则创建新的线程，除非创建此线程超出 maximumPoolSize，在这种情况下，任务将被拒绝。”，所以必然会新建一个线程来运行这个任务。
4. 暂时还可以，但是如果这三个任务都还没完成，连续来了两个任务，第一个添加入 queue 中，后一个呢？queue 中无法插入，而线程数达到了 maximumPoolSize，所以只好执行异常策略了。

所以在使用 SynchronousQueue 通常要求 maximumPoolSize 是无界的，这样就可以避免上述情况发生（如果希望限制就直接使用有界队列）。对于使用

SynchronousQueue 的作用 jdk 中写的很清楚：此策略可以避免在处理可能具有内部依赖性的请求集时出现锁。

什么意思？如果你的任务 A1，A2 有内部关联，A1 需要先运行，那么先提交 A1，再提交 A2，当使用 SynchronousQueue 我们可以保证，A1 必定先被执行，在 A1 么有被执行前，A2 不可能添加入 queue 中。

例子二：使用无界队列策略，即 LinkedBlockingQueue

这个就拿 **newFixedThreadPool** 来说，根据前文提到的规则：

如果运行的线程少于 corePoolSize，则 Executor 始终首选添加新的线程，而不进行排队。那么当任务继续增加，会发生什么呢?

如果运行的线程等于或多于 corePoolSize，则 Executor 始终首选将请求加入队列，而不添加新的线程。OK，此时任务变加入队列之中了，那什么时候才会添加新线程呢?

如果无法将请求加入队列，则创建新的线程，除非创建此线程超出 maximumPoolSize，在这种情况下，任务将被拒绝。这里就很有意思了，可能会出现无法加入队列吗？不像 SynchronousQueue 那样有其自身的特点，对于无界队列来说，总是可以加入的（资源耗尽，当然另当别论）。换句说，永远也不会触发产生新的线程！corePoolSize 大小的线程数会一直运行，忙完当前的，就从队列中拿任务开始运行。所以要防止任务疯长，比如任务运行的实行比较长，而添加任务的速度远远超过处理任务的时间，而且还不断增加，不一会儿就爆了。

例子三：有界队列，使用 ArrayBlockingQueue。

这个是最为复杂的使用，所以 JDK 不推荐使用也有些道理。与上面的相比，最大的特点便是可以防止资源耗尽的情况发生。

举例来说，请看如下构造方法：

```
1.  new ThreadPoolExecutor(
2.          2, 4, 30, TimeUnit.SECONDS,
3.          new ArrayBlockingQueue<Runnable>(2),
4.          new RecorderThreadFactory("CookieRecorderPool"),
5.          new ThreadPoolExecutor.CallerRunsPolicy());
```

new ThreadPoolExecutor(
2, 4, 30, TimeUnit.SECONDS,
new ArrayBlockingQueue<Runnable>(2),
new RecorderThreadFactory("CookieRecorderPool"),
new ThreadPoolExecutor.CallerRunsPolicy());

假设，所有的任务都永远无法执行完。

对于首先来的 A,B 来说直接运行，接下来，如果来了 C,D，他们会被放到 queue 中，如果接下来再来 E,F，则增加线程运行 E，F。但是如果再来任务，队列无法再接受了，线程数也到达最大的限制了，所以就会使用拒绝策略来处理。

keepAliveTime

jdk 中的解释是：当线程数大于核心时，此为终止前多余的空闲线程等待新任务的最长时间。

有点拗口，其实这个不难理解，在使用了“池”的应用中，大多都有类似的参数需要配置。比如数据库连接池，DBCP 中的 maxIdle，minIdle 参数。

什么意思？接着上面的解释，后来向老板派来的工人始终是“借来的”，俗话说“有借就有还”，但这里的问题就是什么时候还了，如果借来的工人刚完成一个任务就还回去，后来发现任务还有，那岂不是又要去借？这一来一往，老板肯定头也大死了。

合理的策略：既然借了，那就多借一会儿。直到“某一段”时间后，发现再也用不到这些工人时，便可以还回去了。这里的某一段时间便是 keepAliveTime 的含义，TimeUnit 为 keepAliveTime 值的度量。

RejectedExecutionHandler

另一种情况便是，即使向老板借了工人，但是任务还是继续过来，还是忙不过来，这时整个队伍只好拒绝接受了。

RejectedExecutionHandler 接口提供了对于拒绝任务的处理的自定方法的机会。在 ThreadPoolExecutor 中已经默认包含了 4 中策略，因为源码非常简单，这里直接贴出来。

CallerRunsPolicy：线程调用运行该任务的 execute 本身。此策略提供简单的反馈控制机制，能够减缓新任务的提交速度。

```
1.  public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
2.          if (!e.isShutdown()) {
3.              r.run();
4.          }
5.      }
```

```
public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
        if (!e.isShutdown()) {
            r.run();
        }
    }
```

这个策略显然不想放弃执行任务。但是由于池中已经没有任何资源了，那么就直接使用调用该 execute 的线程本身来执行。

AbortPolicy：处理程序遭到拒绝将抛出运行时 RejectedExecutionException

```
1.  public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
2.          throw new RejectedExecutionException();
3.      }
```

```
public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
    throw new RejectedExecutionException();
}
```

这种策略直接抛出异常，丢弃任务。

DiscardPolicy：不能执行的任务将被删除

```
1.  public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
2.      }
```

```
public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
}
```

这种策略和 AbortPolicy 几乎一样，也是丢弃任务，只不过他不抛出异常。

DiscardOldestPolicy：如果执行程序尚未关闭，则位于工作队列头部的任务将被删除，然后重试执行程序（如果再次失败，则重复此过程）

```
1.  public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
2.          if (!e.isShutdown()) {
3.              e.getQueue().poll();
4.              e.execute(r);
5.          }
6.      }
```

```
public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
    if (!e.isShutdown()) {
        e.getQueue().poll();
        e.execute(r);
```

```
        }
    }
```

该策略就稍微复杂一些，在 pool 没有关闭的前提下首先丢掉缓存在队列中的最早的任务，然后重新尝试运行该任务。这个策略需要适当小心。

设想:如果其他线程都还在运行，那么新来任务踢掉旧任务，缓存在 queue 中，再来一个任务又会踢掉 queue 中最老任务。

总结:

keepAliveTime 和 maximumPoolSize 及 BlockingQueue 的类型均有关系。如果 BlockingQueue 是无界的，那么永远不会触发 maximumPoolSize，自然 keepAliveTime 也就没有了意义。

反之，如果核心数较小，有界 BlockingQueue 数值又较小，同时 keepAliveTime 又设的很小，如果任务频繁，那么系统就会频繁的申请回收线程。

```
public static ExecutorService newFixedThreadPool(int
nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  0L, TimeUnit.MILLISECONDS,
                                  new
LinkedBlockingQueue<Runnable>());
  }
```

还有很多其他的方法:

比如：getQueue() 、getPoolSize() 、getActiveCount()、getCompletedTaskCount()

等获取与线程池相关属性的方法，有兴趣的朋友可以自行查阅 API。

二、 多线程面试题目

多线程同步有哪几种方法?

Synchronized 关键字，Lock 锁实现，分布式锁等。

什么是死锁？如何避免死锁?

死锁就是两个线程相互等待对方释放对象锁。

多线程之间如何进行通信?

wait/notify

什么是线程池?

很简单，简单看名字就知道是装有线程的池子，我们可以把要执行的多线程交给线程池来处理，和连接池的概念一样，通过维护一定数量的线程池来达到多个线程的复用。

线程池的好处

我们知道不用线程池的话，每个线程都要通过 new Thread(xxRunnable).start()的方式来创建并运行一个线程，线程少的话这不会是问题，而真实环境可能会开启多个线程让系统和程序达到最佳效率，当线程数达到一定数量就会耗尽系统的 CPU 和内存资源，也会造成 GC 频繁收集和停顿，因为每次创建和销毁一个线程都是要消耗系统资源的，如果为每个任务都创建线程这无疑是一个很大的性能瓶颈。所以，线程池中的线程复用极大节省了系统资源，当线程一段时间不再有任务处理时它也会自动销毁，而不会长驻内存。

什么是活锁、饥饿、无锁、死锁?

死锁、活锁、饥饿是关于多线程是否活跃出现的运行阻塞障碍问题，如果线程出现了这三种情况，即线程不再活跃，不能再正常地执行下去了。

死锁

死锁是多线程中最差的一种情况，多个线程相互占用对方的资源的锁，而又相互等对方释放锁，此时若无外力干预，这些线程则一直处理阻塞的假死状态，形成死锁。

举个例子，A 同学抢了 B 同学的钢笔，B 同学抢了 A 同学的书，两个人都相互占用对方的东西，都在让对方先还给自己自己再还，这样一直争执下去等待对方还而又得不到解决，老师知道此事后就让他们相互还给对方，这样在外力的干预下他们才解决，当然这只是个例子没有老师他们也能很好解决，计算机不像人如果发现这种情况没有外力干预还是会一直阻塞下去的。

活锁

活锁这个概念大家应该很少有人听说或理解它的概念，而在多线程中这确实存在。活锁恰恰与死锁相反，死锁是大家都拿不到资源都占用着对方的资源，而活锁是拿到资源却又相互释放不执行。当多线程中出现了相互谦让，都主动将资源释放给别的线程使用，这样这个资源在多个线程之间跳动而又得不到执行，这就是活锁。

饥饿

我们知道多线程执行中有线程优先级这个东西，优先级高的线程能够插队并优先执行，这样如果优先级高的线程一直抢占优先级低线程的资源，导致低优先级线程无法得到执行，这就是饥饿。当然还有一种饥饿的情况，一个线程一直占着一个资源不放而导致其他线程得不到执行，与死锁不同的是饥饿在以后一段时间内还是能够得到执行的，如那个占用资源的线程结束了并释放了资源。

无锁

无锁，即没有对资源进行锁定，即所有的线程都能访问并修改同一个资源，但同时只有一个线程能修改成功。无锁典型的特点就是一个修改操作在一个循环内进行，线程会不断的尝试修改共享资源，如果没有冲突就修改成功并退出否则就会继续下一次循环尝试。所以，如果有多个线程修改同一个值必定会有一个线程能修改成功，而其他修改失败的线程会不断重试直到修改成功。之前的文章我介绍过 JDK 的 CAS 原理及应用即是无锁的实现。

可以看出，无锁是一种非常良好的设计，它不会出现线程出现的跳跃性问题，锁使用不当肯定会出现系统性能问题，虽然无锁无法全面代替有锁，但无锁在某些场合下是非常高效的。

Synchronized 有哪几种用法?

锁类、锁方法、锁代码块。

Fork/Join 框架是干什么的?

大任务自动分散小任务，并发执行，合并小任务结果。

、Java 中用到了什么线程调度算法?

抢占式。一个线程用完 CPU 之后，操作系统会根据线程优先级、线程饥饿情况等数据算出一个总的优先级并分配下一个时间片给某个线程执行。

三、 Spinrg 框架

什么是 SpringIOC 容器？

SpringIOC 负责创建对象，管理对象（通过依赖注入（DI），装配对象，配置对象，并且管理这些对象的整个生命周期。

IOC 的优点是什么？

IOC 或依赖注入把应用的代码量降到最低。它使应用容易测试，单元测试不再需要单例和 JNDI 查找机制。最小的代价和最小的侵入性使松散耦合得以实现。IOC 容器支持加载服务时的饿汉式初始化和懒加载。

ApplicationContext 通常的实现是什么？

- **FileSystemXmlApplicationContext**：此容器从一个 XML 文件中加载 beans 的定义，XMLBean 配置文件的全路径名必须提供给它的构造函数。
- **ClassPathXmlApplicationContext**：此容器也从一个 XML 文件中加载 beans 的定义，这里，你需要正确设置 classpath 因为这个容器将在 classpath 里找 bean 配置。
- **WebXmlApplicationContext**：此容器加载一个 XML 文件，此文件定义了一个 WEB 应用的所有 bean。

Bean 工厂和 Applicationcontexts 有什么区别？

Applicationcontexts 提供一种方法处理文本消息，一个通常的做法是加载文件资源（比如镜像），它们可以向注册为监听器的 bean 发布事件。另外，在容器或容器内的对象上执行的那些不得不由 bean 工厂以程序化方式处理的操作，可以在 Applicationcontexts 中以声明的方式处理。Applicationcontexts 实现了 MessageSource 接口，该接口的实现以可插拔的方式提供获取本地化消息的方法。

什么是 Spring 的依赖注入？

依赖注入，是 IOC 的一个方面，是个通常的概念，它有多种解释。这概念是说你不用创建对象，而只需要描述它如何被创建。你不在代码里直接组装你的组件和服务，但是要在配置文件里描述哪些组件需要哪些服务，之后一个容器（IOC 容器）负责把他们组装起来。

有哪些不同类型的 IOC（依赖注入）方式？

- **构造器依赖注入：**构造器依赖注入通过容器触发一个类的构造器来实现的，该类有一系列参数，每个参数代表一个对其他类的依赖。
- **Setter 方法注入：**Setter 方法注入是容器通过调用无参构造器或无参 static 工厂方法实例化 bean 之后，调用该 bean 的 setter 方法，即实现了基于 setter 的依赖注入。

哪种依赖注入方式你建议使用，构造器注入，还是 Setter 方法注入？

你两种依赖方式都可以使用，构造器注入和 Setter 方法注入。最好的解决方案是用构造器参数实现强制依赖，setter 方法实现可选依赖。

SpringBeans

什么是 Springbeans?

Springbeans 是那些形成 Spring 应用的主干的 java 对象。它们被 SpringIOC 容器初始化，装配，和管理。这些 beans 通过容器中配置的元数据创建。比如，以 XML 文件中<bean/>的形式定义。

Spring 框架定义的 beans 都是单件 beans。在 beantag 中有个属性"singleton"，如果它被赋为 TRUE，bean 就是单件，否则就是一个 prototypebean。默认是 TRUE，所以所有在 Spring 框架中的 beans 缺省都是单件。

一个 SpringBean 定义包含什么？

一个 SpringBean 的定义包含容器必知的所有配置元数据，包括如何创建一个 bean，它的生命周期详情及它的依赖。

如何给 Spring 容器提供配置元数据?

这里有三种重要的方法给 Spring 容器提供配置元数据。

XML 配置文件。

基于注解的配置。

基于 java 的配置。

你怎样定义类的作用域?

当定义一个<bean>在 Spring 里，我们还能给这个 bean 声明一个作用域。它可以通过 bean 定义中的 scope 属性来定义。如，当 Spring 要在需要的时候每次生产一个新的 bean 实例，bean 的 scope 属性被指定为 prototype。另一方面，一个 bean 每次使用的时候必须返回同一个实例，这个 bean 的 scope 属性必须设为 singleton。

解释 Spring 支持的几种 bean 的作用域。

Spring 框架支持以下五种 bean 的作用域：

- **singleton:**bean 在每个 Springioc 容器中只有一个实例。
- **prototype**：一个 bean 的定义可以有多个实例。
- **request**：每次 http 请求都会创建一个 bean，该作用域仅在基于 web 的 SpringApplicationContext 情形下有效。
- **session**：在一个 HTTPSession 中，一个 bean 定义对应一个实例。该作用域仅在基于 web 的 SpringApplicationContext 情形下有效。
- **global-session**：在一个全局的 HTTPSession 中，一个 bean 定义对应一个实例。该作用域仅在基于 web 的 SpringApplicationContext 情形下有效。

缺省的 Springbean 的作用域是 Singleton.

Spring 框架中的单例 bean 是线程安全的吗?

不，Spring 框架中的单例 bean 不是线程安全的。

解释 Spring 框架中 bean 的生命周期。

- Spring 容器从 XML 文件中读取 bean 的定义，并实例化 bean。
- Spring 根据 bean 的定义填充所有的属性。
- 如果 bean 实现了 BeanNameAware 接口，Spring 传递 bean 的 ID 到 setBeanName 方法。
- 如果 Bean 实现了 BeanFactoryAware 接口，Spring 传递 beanfactory 给 setBeanFactory 方法。
- 如果有任何与 bean 相关联的 BeanPostProcessors，Spring 会在 postProcesserBeforeInitialization()方法内调用它们。
- 如果 bean 实现 IntializingBean 了，调用它的 afterPropertySet 方法，如果 bean 声明了初始化方法，调用此初始化方法。
- 如果有 BeanPostProcessors 和 bean 关联，这些 bean 的 postProcessAfterInitialization() 方法将被调用。
- 如果 bean 实现了 DisposableBean，它将调用 destroy()方法。

哪些是重要的 bean 生命周期方法？你能重载它们吗？

有两个重要的 bean 生命周期方法，第一个是 setup，它是在容器加载 bean 的时候被调用。第二个方法是 teardown 它是在容器卸载类的时候被调用。

Thebean 标签有两个重要的属性（init-method 和 destroy-method）。用它们你可以自己定制初始化和注销方法。它们也有相应的注解（@PostConstruct 和@PreDestroy）。

什么是 Spring 的内部 bean？

当一个 bean 仅被用作另一个 bean 的属性时，它能被声明为一个内部 bean，为了定义 innerbean，在 Spring 的基于 XML 的配置元数据中，可以在<property/>或<constructor-arg/>元素内使用<bean/>元素，内部 bean 通常是匿名的，它们的 Scope 一般是 prototype。

在 Spring 中如何注入一个 java 集合？

Spring 提供以下几种集合的配置元素：

- <list>类型用于注入一列值，允许有相同的值。
- <set>类型用于注入一组值，不允许有相同的值。
- <map>类型用于注入一组键值对，键和值都可以为任意类型。
- <props>类型用于注入一组键值对，键和值都只能为 String 类型。

什么是 bean 装配?

装配，或 bean 装配是指在 Spring 容器中把 bean 组装到一起，前提是容器需要知道 bean 的依赖关系，如何通过依赖注入来把它们装配到一起。

什么是 bean 的自动装配？

Spring 容器能够自动装配相互合作的 bean，这意味着容器不需要<constructor-arg>和<property>配置，能通过 Bean 工厂自动处理 bean 之间的协作。

解释不同方式的自动装配。

有五种自动装配的方式，可以用来指导 Spring 容器用自动装配方式来进行依赖注入。

- **no**：默认的方式是不进行自动装配，通过显式设置 ref 属性来进行装配。
- **byName**：通过参数名自动装配，Spring 容器在配置文件中发现 bean 的 autowire 属性被设置成 byname，之后容器试图匹配、装配和该 bean 的属性具有相同名字的 bean。
- **byType:**：通过参数类型自动装配，Spring 容器在配置文件中发现 bean 的 autowire 属性被设置成 byType，之后容器试图匹配、装配和该 bean 的属性具有相同类型的 bean。如果有多个 bean 符合条件，则抛出错误。
- **constructor：这个方式类似于** byType，但是要提供给构造器参数，如果没有确定的带参数的构造器参数类型，将会抛出异常。
- **autodetect**：首先尝试使用 constructor 来自动装配，如果无法工作，则使用 byType 方式。

自动装配有哪些局限性?

自动装配的局限性是：

- **重写**：你仍需用<constructor-arg>和<property>配置来定义依赖，意味着总要重写自动装配。
- **基本数据类型**：你不能自动装配简单的属性，如基本数据类型，String 字符串，和类。
- **模糊特性**：自动装配不如显式装配精确，如果有可能，建议使用显式装配。

你可以在 Spring 中注入一个 null 和一个空字符串吗？

可以。

Spring 注解

什么是基于 Java 的 Spring 注解配置?给一些注解的例子.

基于 Java 的配置，允许你在少量的 Java 注解的帮助下，进行你的大部分 Spring 配置而非通过 XML 文件。

以@Configuration 注解为例，它用来标记类可以当做一个 bean 的定义，被 SpringIOC 容器使用。另一个例子是@Bean 注解，它表示此方法将要返回一个对象，作为一个 bean 注册进 Spring 应用上下文。

什么是基于注解的容器配置?

相对于 XML 文件，注解型的配置依赖于通过字节码元数据装配组件，而非尖括号的声明。

开发者通过在相应的类，方法或属性上使用注解的方式，直接组件类中进行配置，而不是使用 xml 表述 bean 的装配关系。

怎样开启注解装配?

注解装配在默认情况下是不开启的，为了使用注解装配，我们必须在 Spring 配置文件中配置<context:annotation-config/>元素。

@Required 注解

这个注解表明 bean 的属性必须在配置的时候设置，通过一个 bean 定义的显式的属性值或通过自动装配，若@Required 注解的 bean 属性未被设置，容器将抛出 BeanInitializationException。

@Autowired 注解

@Autowired 注解提供了更细粒度的控制，包括在何处以及如何完成自动装配。它的用法和@Required 一样，修饰 setter 方法、构造器、属性或者具有任意名称和/或多个参数的 PN 方法。

四、 Java 虚拟机

一、什么是 Java 虚拟机

Java 虚拟机是一个想象中的机器,在实际的计算机上通过软件模拟来实现。Java 虚拟机有自己想象中的硬件,如处理器、堆栈、寄存器等,还具有相应的指令系统。

二、为什么使用 Java 虚拟机

Java 语言的一个非常重要的特点就是与平台的无关性。而使用 Java 虚拟机是实现这一特点的关键。一般的高级语言如果要在不同的平台上运行,至少需要编译成不同的目标代码。而引入 Java 语言虚拟机后,Java 语言在不同平台上运行时不需要重新编译。Java 虚拟机屏蔽了与具体平台相关的信息,使得 Java 语言编译程序只需生成在 Java 虚拟机上运行的目标代码(字节码),就可以在多种平台上不加修改地运行。Java 虚拟机在执行字节码时,把字节码解释成具体平台上的机器指令执行。

三、Java 虚拟机的生命周期

一个运行中的 Java 虚拟机有着一个清晰的任务：执行 Java 程序。程序开始执行时它才运行，程序结束时它就停止。假如你同时运行三个 Java 程序，就会有三个运行中的 Java 虚拟机。

Java 虚拟机总是开始于一个 main()方法，这个方法必须是公有 public、返回 void、直接接收一个字符串数组。在程序执行时,你必须给 Java 虚拟机指明这个包含有 main()方法的类名。

Main()方法是程序的起点，它被执行的线程初始化为程序的初始线程。程序中其它的线程都由他来启动。Java 中的线程分为两种：守护线程 （daemon）和普通线程（non-daemon）。守护线程是 Java 虚拟机自己使用的线程，比如负责垃圾收集的线程就是一个守护线程。当然，你也可以把自己的程序设置为守护线程。包含 Main()方法的初始线程不是守护线程。

只要 Java 虚拟机中还有普通的线程在执行，Java 虚拟机就不会停止。如果有足够的权限，你可以调用 exit()方法终止程序。

三、Java 虚拟机的体系结构

在 Java 虚拟机的规范中定义了一系列的子系统、内存区域、数据类型和使用指南。这些组件构成了 Java 虚拟机的内部结构，他们不仅仅为 Java 虚拟机的实现提供了清晰的内部结构，更是严格规定了 Java 虚拟机实现的外部行为。

每一个 Java 虚拟机都由一个类加载器子系统（class loader subsystem），负责加载程序中的类型（类 class 和接口 interface），并赋予唯一的名字。每一个 Java 虚拟机都有一个执行引擎（execution engine）负责执行被加载类中包含的指令。

程序的执行需要一定的内存空间，如字节码、被加载类的其他额外信息、程序中的对象、方法的参数、返回值、本地变量、处理的中间变量等等。Java 虚拟机将这些信息统统保存在数据区（data area）中。虽然每个 Java 虚拟机的实现中都包含数据区，但是 Java 虚拟机规范对数据区的规定却非常的抽象。

每一个 Java 虚拟机都包含方法区（method area）和堆（heap），他们都被整个程序共享。Java 虚拟机加载并解析一个类以后，将从类文件中解析出来的信息保存与方法区中。程序执行时创建的 对象都保存在堆中。

当一个线程被创建时，会被分配只属于它自己的 PC 寄存器“pc register”（程序计数器）和 Java 堆栈（Java stack）。当线程不调用本地方法时，PC 寄存器中保存线程执行的下一条指令。Java 堆栈保存了一个线程调用方法时的状态，包括本地变量、调用方法的 参数、返回值、处理的中间变量。调用本地方法时的状态保存在本地方法堆栈中（native method stacks），可能再寄存器或者其他非平台独立的内存中。

四、数据类型（Data Types）

所有 Java 虚拟机中使用的数据都有确定的数据类型，数据类型和操作都在 Java 虚拟机规范中严格定义。Java 中的数据类型分为原始数据类型 （primitive types）和引

用数据类型（reference type）。引用类型依赖于实际的对象，但不是对象本身。原始数据类型不依赖于任何东西，它们就是本身表示的数据。

所有 Java 程序语言中的原始数据类型，都是 Java 虚拟机的原始数据类型，除了布尔型（boolean）。当编译器将 Java 源代码编译为字节码时，使用整型（int）或者字节型 （byte）去表示布尔型。在 Java 虚拟机中使用整数 0 表示布尔型的 false，使用非零整数表示布尔型的 true，布尔数组被表示为字节数组，虽然它们可能会以字节数组或者字节块（bit fields）保存在堆中。

除了布尔型，其它的原始类型都是 Java 虚拟机中的数据类型。在 Java 中数据类型被分为：整形的 byte，short，int，long；char 和浮点型的 float，double。Java 语言中的数据类型在任何主机上都有同样的范围。

引用类型可能被创建为：类类型（class type），接口类型（interface type），数组类型（array type）。他们都引用被动态创建的对象。当引用类型引用 null 时，说明没有引用任何对象。

Java 虚拟机规范只定义了每一种数据类型表示的范围，没有定义在存储时每种类型占用的空间。他们如何存储由 Java 虚拟机的实现者自己决定。关于浮点型更多信息参见 14 章“Floating Point Arithmetic”。

五、字节长度

Java 虚拟机中最小的数据单元式字（word），其大小由 Java 虚拟机的实现者定义。但是一个字的大小必须足够容纳 byte，short，int， char，float，returnValue，reference；两个字必须足够容纳 long，double。所以虚拟机的实现者至少提供的字不能小于 31bits 的字，但是最好选择特定平台上最有效率的字长。

在运行时，Java 程序不能决定所运行机器的字长。字长也不会影响程序的行为，他只是在 Java 虚拟机中的一种表现方式。

六、类加载器子系统

Java 虚拟机中的类加载器分为两种：原始类加载器（primordial class loader）和类加载器对象（class loader objects）。

原始类加载器是 Java 虚拟机实现的一部分，类加载器对象是运行中的程序的一部分。不同类加载器加载的类被不同的命名空间所分割。

类加载器调用了许多 Java 虚拟机中其他的部分和 java.lang 包中的很多类。比如，类加载对象就是 java.lang.ClassLoader 子类 的实例，ClassLoader 类中的方法可以访问虚拟机中的类加载机制；每一个被 Java 虚拟机加载的类都会被表示为一个 java.lang.Class 类的实例。像其他对象一样，类加载器对象和 Class 对象都保存在堆中，被加载的信息被保存在方法区中。

1、加载、连接、初始化（Loading, Linking and Initialization）

类加载子系统不仅仅负责定位并加载类文件，他按照以下严格的步骤作了很多其他的事情：（具体的信息参见第七章的“类的生命周期”）

1）、加载：寻找并导入指定类型（类和接口）的二进制信息

2）、连接：进行验证、准备和解析

①验证：确保导入类型的正确性

②准备：为类型分配内存并初始化为默认值

③解析：将字符引用解析为直接饮用

3）、初始化：调用 Java 代码，初始化类变量为合适的值

2、原始类加载器（The Primordial Class Loader）

每个 Java 虚拟机都必须实现一个原始类加载器，他能够加载那些遵守类文件格式并且被信任的类。但是，Java 虚拟机的规范并没有定义如何加载类，这由 Java 虚拟机实现者自己决定。对于给定类型名的类型，原始莱加载器必须找到那个类型名加“.class”的文件并加载入虚拟机中。

3、类加载器对象

虽然类加载器对象是 Java 程序的一部分，但是 ClassLoader 类中的三个方法可以访问 Java 虚拟机中的类加载子系统。

1）、protected final Class defineClass(...)：使用这个方法可以出入一个字节数组，定义一个新的类型。

2）、protected Class findSystemClass(String name)：加载指定的类，如果已经加载，就直接返回。

3）、protected final void resolveClass(Class c)：defineClass()方法只是加载一个类，这个方法负责后续的动态连接和初始化。

4、命名空间

当多个类加载器加载了同一个类时，为了保证他们名字的唯一性，需要在类名前加上加载该类的类加载器的标识。

七、方法区（The Method Area）

在 Java 虚拟机中，被加载类型的信息都保存在方法区中。

程序中的所有线程共享一个方法区，所以访问方法区信息的方法必须是线程安全的。如果你有两个线程都去加载一个叫 Lava 的类，那只能由一个线程被容许去加载这个类，另一个必须等待。

在程序运行时，方法区的大小是可变的，程序在运行时可以扩展。有些 Java 虚拟机

的实现也可以通过参数也订制方法区的初始大小，最小值和最大值。

方法区也可以被垃圾收集。因为程序中的类由类加载器动态加载，所有类可能变成没有被引用（unreferenced）的状态。当类变成这种状态时，他就可 能被垃圾收集掉。没有加载的类包括两种状态，一种是真正的没有加载，另一个种是“unreferenced”的状态。详细信息参见第七章的类的生命周期 （The Lifetime of a Class）。

1、类型信息（Type Information）

每一个被加载的类型，在 Java 虚拟机中都会在方法区中保存如下信息：

1）、类型的全名（The fully qualified name of the type）

2）、类型的父类型的全名（除非没有父类型，或者弗雷形式 java.lang.Object）（The fully qualified name of the typeís direct superclass）

3）、给类型是一个类还是接口（class or an interface）（Whether or not the type is a class ）

4）、类型的修饰符（public，private，protected，static，final，volatile，transient 等）（The typeís modifiers）

5）、所有父接口全名的列表（An ordered list of the fully qualified names of any direct superinterfaces）

类型全名保存的数据结构由虚拟机实现者定义。除此之外，Java 虚拟机还要为每个类型保存如下信息：

1）、类型的常量池（The constant pool for the type）

2）、类型字段的信息（Field information）

3）、类型方法的信息（Method information）

4）、所有的静态类变量（非常量）信息（All class (static) variables declared

in the type, except constants）

5）、一个指向类加载器的引用（A reference to class ClassLoader）

6）、一个指向 Class 类的引用（A reference to class Class）

1）、类型的常量池（The constant pool for the type）

常量池中保存中所有类型是用的有序的常量集合，包含直接常量（literals）如字符串、整数、浮点数的常量，和对类型、字段、方法的符号引用。常量池 中每一个保存的常量都有一个索引，就像数组中的字段一样。因为常量池中保存中所有类型使用到的类型、字段、方法的字符引用，所以它也是动态连接的主要对 象。

2）、类型字段的信息（Field information）

字段名、字段类型、字段的修饰符（public，private，protected，static，final，volatile，transient 等）、字段在类中定义的顺序。

3）、类型方法的信息（Method information）

方法名、方法的返回值类型（或者是 void）、方法参数的个数、类型和他们的顺序、字段的修饰符（public，private，protected，static，final，volatile，transient 等）、方法在类中定义的顺序

如果不是抽象和本地本法还需要保存

方法的字节码、方法的操作数堆栈的大小和本地变量区的大小（稍候有详细信息）、异常列表（详细信息参见第十七章“Exceptions”。）

4）、类（静态）变量（Class Variables）

类变量被所有类的实例共享，即使不通过类的实例也可以访问。这些变量绑定在类上（而不是类的实例上），所以他们是类的逻辑数据的一部分。在 Java 虚拟机使用

这个类之前就需要为类变量（non-final）分配内存

常量（final）的处理方式于这种类变量（non-final）不一样。每一个类型在用到一个常量的时候，都会复制一份到自己的常量池中。常量也像类变 量一样保存在方法区中，只不过他保存在常量池中。（可能是，类变量被所有实例共享，而常量池是每个实例独有的）。Non-final 类变量保存为定义他的 类型数据（data for the type that declares them）的一部分，而 final 常量保存为使用他的类型数据（data for any type that uses them）的一部分。

5）、指向类加载器的引用（A reference to class ClassLoader）

每一个被 Java 虚拟机加载的类型，虚拟机必须保存这个类型是否由原始类加载器或者类加载器加载。那些被类加载器加载的类型必须保存一个指向类加载器的引 用。当类加载器动态连接时，会使用这条信息。当一个类引用另一个类时，虚拟机必须保存那个被引用的类型是被同一个类加载器加载的，这也是虚拟机维护不同命 名空间的过程。详情参见第八章“The Linking Model”

6）、指向 Class 类的引用（A reference to class Class）

Java 虚拟机为每一个加载的类型创建一个 java.lang.Class 类的实例。你也可以通过 Class 类的方法：

public static Class forName(String className)来查找或者加载一个类，并取得相应的 Class 类的实例。通过这个 Class 类的实例，我们可以访问 Java 虚拟机方法区中的信息。具体参照 Class 类的 JavaDoc。

2、方法列表（Method Tables）

为了更有效的访问所有保存在方法区中的数据，这些数据的存储结构必须经过仔细的设计。所有方法区中，除了保存了上边的那些原始信息外，还有一个为了加快存 取

速度而设计的数据结构，比如方法列表。每一个被加载的非抽象类，Java 虚拟机都会为他们产生一个方法列表，这个列表中保存了这个类可能调用的所有实例 方法的引用，报错那些父类中调用的方法。

八、堆

当 Java 程序创建一个类的实例或者数组时，都在堆中为新的对象分配内存。虚拟机中只有一个堆，所有的线程都共享它。

1、垃圾收集（Garbage Collection）

垃圾收集是释放没有被引用的对象的主要方法。它也可能会为了减少堆的碎片，而移动对象。在 Java 虚拟机的规范中没有严格定义垃圾收集，只是定义一个 Java 虚拟机的实现必须通过某种方式管理自己的堆。详情参见第九章“Garbage Collection”。

2、对象存储结构（Object Representation）

Java 虚拟机的规范中没有定义对象怎样在堆中存储。每一个对象主要存储的是他的类和父类中定义的对象变量。对于给定的对象的引用，虚拟机必须嫩耨很快的 定位到这个对象的数据。另为，必须提供一种通过对象的引用方法对象数据的方法，比如方法区中的对象的引用，所以一个对象保存的数据中往往含有一个某种形式 指向方法区的指针。

一个可能的堆的设计是将堆分为两个部分：引用池和对象池。一个对象的引用就是指向引用池的本地指针。每一个引用池中的条目都包含两个部分：指向对象池中对 象数据的指针和方法区中对象类数据的指针。这种设计能够方便 Java 虚拟机堆碎片的整理。当虚拟机在对象池中移动一个对象的时候，只需要修改对应引用池中 的指针地址。但是每次访问对象的数据都需要处理两次指针。下图演示了这种堆的设计。在第九章的“垃圾收集”中的 HeapOfFish Applet 演示了这种设计。

另一种堆的设计是：一个对象的引用就是一个指向一堆数据和指向相应对象的偏移指

针。这种设计方便了对象的访问，可是对象的移动要变的异常复杂。下图演示了这种设计

当程序试图将一个对象转换为另一种类型时，虚拟机需要判断这种转换是否是这个对象的类型，或者是他的父类型。当程序适用 instanceof 语句的时候也 会做类似的事情。当程序调用一个对象的方法时，虚拟机需要进行动态绑定，他必须判断调用哪一个类型的方法。这也需要做上面的判断。

无论虚拟机实现者使用哪一种设计，他都可能为每一个对象保存一个类似方法列表的信息。因为他可以提升对象方法调用的速度，对提升虚拟机的性能非常重要，但 是虚拟机的规范中比没有要求必须实现类似的数据结构。下图描述了这种结构。图中显示了一个对象引用相关联的所有的数据结构，包括：

1)、一个指向类型数据的指针

2)、一个对象的方法列表。方法列表是一个指向所有可能被调用对象方法的指针数组。方法数据包括三个部分：操作码堆栈的大小和方法堆栈的本地变量区；方法的字节码；异常列表。

每一个 Java 虚拟机中的对象必须关联一个用于同步多线程的 lock(mutex)。同一时刻，只能有一个对象拥有这个对象的锁。当一个拥有这个这个对象 的锁，他就可以多次申请这个锁，但是也必须释放相应次数的锁才能真正释放这个对象锁。很多对象在整个生命周期中都不会被锁，所以这个信息只有在需要时才需 要添加。很多 Java 虚拟机的实现都没有在对象的数据中包含“锁定数据”，只是在需要时才生成相应的数据。除了实现对象的锁定，每一个对象还逻辑关联到一 个“wait set”的实现。锁定帮组线程独立处理共享的数据，不需要妨碍其他的线程。“wait set”帮组线程协作完成同一个目标。“wait set”往往通过 Object 类的 wait()和 notify()方法来实现。

垃圾收集也需要堆中的对象是否被关联的信息。Java 虚拟机规范中指出垃圾收集一个运

行一个对象的 finalizer 方法一次，但是容许 finalizer 方法重新引用这个对象，当这个对象再次不被引用时，就不需要再次调用 finalize 方法。所以虚拟机也需要保存 finalize 方法 是否运行过的信息。更多信息参见第九章的“垃圾收集”

3、数组的保存（Array Representation）

在 Java 中，数组是一种完全意义上的对象，他和对象一样保存在堆中、有一个指向 Class 类实例的引用。所有同一维度和类型的数组拥有同样的 Class，数组的长 度不做考虑。对应 Class 的名字表示为维度和类型。比如一个整型数据的 Class 为“[I”，字节型三维数组 Class 名为“[[[B”，两维对象数据 Class 名为“[[Ljava.lang.Object”。

多维数组被表示为数组的数组，如下图：

数组必须在堆中保存数组的长度，数组的数据和一些对象数组类型数据的引用。通过一个数组引用的，虚拟机应该能够取得一个数组的长度，通过索引能够访问特定 的数据，能够调用 Object 定义的方法。Object 是所有数据类的直接父类。更多信息参见第六章“类文件”。

九、PC 寄存器（程序计数器）（The Program Counter）

每一个线程开始执行时都会被创建一个程序计数器。程序计数器只有一个字长（word），所以它能够保存一个本地指针和 returnValue。当线程执行 时，程序计数器中存放了正在执行指令的地址，这个地址可以使一个本地指针，也可以使一个从方法字节码开始的偏移指针。如果执行本地方法，程序计数器的值没 有被定义。

十、Java 堆栈（The Java Stack）

当一个线程启动时，Java 虚拟机会为他创建一个 Java 堆栈。Java 堆栈用一些离散的 frame 类纪录线程的状态。Java 虚拟机堆 Java 堆栈的操作只有两种：压入和弹出 frames。

线程中正在执行的方法被称为当前方法（current method），当前方法所对应的 frame

被称为当前帧（current frame）。定义当前方法的类被称为当前类（current class），当前类的常量池被称为当前常量池（current constant pool.）。当线程执行时，Java 虚拟机会跟踪当前类和当前常量池。但线程操作保存在帧中的数据时，他只操作当前帧的数据。

当线程调用一个方法时，虚拟机会生成一个新的帧，并压入线程的 Java 堆栈。这个新的帧变成当前帧。当方法执行时，他使用当前帧保存方法的参数、本地变 量、中间结构和其他数据。方法有两种退出方式：正常退出和异常推出。无论方法以哪一种方式推出，Java 虚拟机都会弹出并丢弃方法的帧，上一个方法的帧变 为当前帧。

所有保存在帧中的数据都只能被拥有它的线程访问，线程不能访问其他线程的堆栈中的数据。所以，访问方法的本地变量时，不需要考虑多线程同步。

和方法区、堆一样，Java 堆栈不需要连续的内存空间，它可以被保存在一个分散的内存空间或者堆上。堆栈具体的数据和长度都有 Java 虚拟机的实现者自己定义。一些实现可能提供了执行堆栈最大值和最小值的方法。

十一、堆栈帧（The Stack Frame）

堆栈帧包含三部分：本地变量、操作数堆栈和帧数据。本地变量和操作数堆栈的大小都是一字（word）为单位的，他们在编译就已经确定。帧数据的大小取决于 不同的实现。当程序调用一个方法时，虚拟机从类数据中取得本地变量和操作数堆栈的大小，创建一个合适大小和帧，然后压入 Java 堆栈中。

1、本地变量（Local Variables）

本地变量在 Java 堆栈帧中被组织为一个从 0 计数的数组，指令通过提供他们的索引从本地变量区中取得相应的值。Int,float,reference, returnValue 占一个字，byte,short,char 被转换成 int 然后存储，long 和 doubel 占两个字。

指令通过提供两个字索引中的前一个来取得 long,doubel 的值。比如一个 long 的值存

储在索引 3，4 上，指令就可以通过 3 来取得这个 long 类型的值。

本地变量区中包含了方法的参数和本地变量。编译器将方法的参数以他们申明的顺序放在数组的前面。但是编译器却可以将本地变量任意排列在本地变量数组中，甚至两个本地变量可以公用一个地址，比如，当两个本地变量在两个不交叠的区域内，就像循环变量 i,j。

虚拟机的实现者可以使用任何结构来描述本地变量区中的数据，虚拟机规范中没有定义如何存储 long 和 doubel。

2、操作数堆栈（Operand Stack）

向本地变量一样，操作数堆栈也被组织为一个以字为单位的数组。但是不像本地变量那样通过索引访问，而是通过 push 和 pop 值来实现访问的。如果一个指令 push 一个值到堆栈中，那么下一个指令就可以 pop 并且使用这个值。

操作数堆栈不像程序计数器那样不可以被指令直接访问，指令可以直接访问操作数堆栈。Java 虚拟机是一个以堆栈为基础，而不是以寄存器为基础的，因为它的 指令从堆栈中取得操作数，而不是同寄存器中。当然，指令也可以从其他地方去的操作数，比如指令后面的操作码，或者常量池。但是 Java 虚拟机指令主要是从 操作数堆栈中取得他们需要的操作数。

Java 虚拟机将操作数堆栈视为工作区，很多指令通过先从操作数堆栈中 pop 值，在处理完以后再将结果 push 回操作数堆栈。一个 add 的指令执行过程如 下图所示：先执行 iload_0 和 iload_1 两条指令将需要相加的两个数，从本地方法区中取出，并 push 到操作数堆栈中；然后执行 iadd 指令，现 pop 出两个值，相加，并将结果 pusp 进操作数堆栈中；最后执行 istore_2 指令，pop 出结果，赋值到本地方法区中。

3、帧数据（Frame Data）

处理本地变量和操作数堆栈以外，java 堆栈帧还包括了为了支持常量池，方法返回值和

异常分发需要的数据，他们被保存在帧数据中。

当虚拟机遇到使用指向常量池引用的指令时，就会通过帧数据中指向常量区的指针来访问所需要的信息。前面提到过，常量区中的引用在最开始时都是符号引用。即使当虚拟机检查这些引用时，他们也是字符引用。所以虚拟机需要在这时转换这个引用。

当一个方法正常返回时，虚拟机需要重建那个调用这个方法的方法的堆栈帧。如果执行完的方法有返回值，虚拟机就需要将这个值 push 进调用方法的哪个操作数堆栈中。

帧数据中也包含虚拟机用来处理异常的异常表的引用。异常表定义了一个被 catch 语句保护的一段字节码。每一个异常表中的个体又包含了需要保护的字节玛的 范围，和异常被捕捉到时需要执行的字节码的位置。当一个方法抛出一个异常时，Java 虚拟机就是用异常表去判断如何处理这个异常。如果虚拟机找到了一个匹 配的 catch，他就会将控制权交给 catch 语句。如果没有找到匹配的 catch，方法就会异常返回，然后再调用的方法中继续这个过程。

除了以上的三个用途外，帧数据还可能包含一些依赖于实现的数据，比如调试的信息。

十二、本地方法堆栈

本地方法区依赖于虚拟机的不同实现。虚拟机的实现者可以自己决定使用哪一种机制去执行本地方法。

任何本地方法接口（Native Method Interface）都使用某种形式的本地方法堆栈。

十三、执行引擎

一个 java 虚拟机实现的核心就是执行引擎。在 Java 虚拟机规范，执行引擎被描述为一系列的指令。对于每一个指令，规范都描述了他们应该做什么，但是没有说要如何去做。

1、指令集

在 Java 虚拟机中一个方法的字节码流就是一个指令的序列。每一个指令由一个字节的

操作码（Opcode）和可能存在的操作数（Operands）。操作 码指示去做什么，操作数提供一些执行这个操作码可能需要的额外的信息。一个抽象的执行引擎每次执行一个指令。这个过程发生在每一个执行的线程中。

有时，执行引擎可能会遇到一个需要调用本地方法的指令，在这种情况下，执行引擎会去试图调用本地方法，但本地方法返回时，执行引擎会继续执行字节码流中的下一个指令。本地方法也可以看成对 Java 虚拟机中的指令集的一种扩充。

决定下一步执行那一条指令也是执行引擎工作的一部分。执行引擎有三种方法去取得下一条指令。多数指令会执行跟在他会面的指令；一些像 goto， return 的指令，会在他们执行的时候决定他们的下一条指令；当一个指令抛出异常时，执行引擎通过匹配 catch 语句来决定下一条应该执行的指令。

平台独立性、网络移动性、安全性左右了 Java 虚拟机指令集的设计。平台独立性是指令集设计的主要影响因素之一。基于堆栈的结构使得 Java 虚拟机可以在 更多的平台上实现。更小的操作码，紧凑的结构使得字节码可以更有效的利用网络带宽。一次性的字节码验证，使得字节码更安全，而不影响太多的性能。

2、执行技术

许多种执行技术可以用在 Java 虚拟机的实现中：解释执行，及时编译（just-in-time compiling），hot-spot compiling,native execution in silicon。

3、线程

Java 虚拟机规范定义了一种为了在更多平台上实现的线程模型。Java 线程模型的一个目标时可以利用本地线程。利用本地线程可以让 Java 程序中的线程能过在多处理器机器上真正的同时执行。

Java 线程模型的一个代价就是线程优先级，一个 Java 线程可以在 1-10 的优先级上运

行。1 最低，10 最高。如果设计者使用了本地线程，他们可能将这 10 个优先级映射到本地优先级上。Java 虚拟机规范只定义了，高一点优先级的线程可以却一些 cpu 时间，低优先级的线程在所有高优先级线程都堵塞时，也 可以获取一些 cpu 时间，但是这没有保证：低优先级的线程在高优先级线程没有堵塞时不可以获得一定的 cpu 时间。因此，如果需要在不同的线程间协作，你必 须使用的“同步（synchronizatoin）”。

同步意味着两个部分：对象锁（object locking）和线程等待、激活(thread wait and notify)。对象锁帮助线程可以不受其他线程的干扰。线程等待、激活可以让不同的线程进行协作。

在 Java 虚拟机的规范中，Java 线程被描述为变量、主内存、工作内存。每一个 Java 虚拟机的实例都有一个主内存，他包含了所有程序的变量：对象、数组合类变量。每一个线程都有自己的工作内存，他保存了哪些他可能用到的变量的拷贝。规则：

1)、从主内存拷贝变量的值到工作内存中

2)、将工作内存中的值写会主内存中

如果一个变量没有被同步化，线程可能以任何顺序更新主内存中的变量。为了保证多线程程序的正确的执行，必须使用同步机制。

十四、本地方法接口（Native Method Interface）

Java 虚拟机的实现并不是必须实现本地方法接口。一些实现可能根本不支持本地方法接口。Sun 的本地方法接口是 JNI(Java Native Interface)。

十五、现实中的机器（The Real Machine）

十六、数学方法：仿真(Eternal Math : A Simulation)

五、 分布式缓存面试题目

Redis 相比 memcached 有哪些优势?

(1) memcached 所有的值均是简单的字符串，redis 作为其替代者，支持更为丰富的数据类型

(2) redis 的速度比 memcached 快很多

(3) redis 可以持久化其数据

Redis 支持哪几种数据类型?

String、List、Set、Sorted Set、hashes

Redis 集群方案应该怎么做？都有哪些方案?

1.twemproxy，大概概念是，它类似于一个代理方式，使用方法和普通 redis 无任何区别，设置好它下属的多个 redis 实例后，使用时在本需要连接 redis 的地方改为连接 twemproxy，它会以一个代理的身份接收请求并使用一致性 hash 算法，将请求转接到

具体 redis，将结果再返回 twemproxy。使用方式简便(相对 redis 只需修改连接端口)，对旧项目扩展的首选。 问题：twemproxy 自身单端口实例的压力，使用一致性 hash 后，对 redis 节点数量改变时候的计算值的改变，数据无法自动移动到新的节点。

Redis 回收使用的是什么算法?

LRU 算法

为什么要做 Redis 分区?

分区可以让 Redis 管理更大的内存，Redis 将可以使用所有机器的内存。如果没有分区，你最多只能使用一台机器的内存。分区使 Redis 的计算能力通过简单地增加计算机得到成倍提升,Redis 的网络带宽也会随着计算机和网卡的增加而成倍增长。

Redis 的内存占用情况怎么样?

给你举个例子： 100 万个键值对（键是 0 到 999999 值是字符串“hello world”）在我的 32 位的 Mac 笔记本上 用了 100MB。同样的数据放到一个 key 里只需要 16MB，这是因为键值有一个很大的开销。 在 Memcached 上执行也是类似的结果，但是相对 Redis 的开销要小一点点，因为 Redis 会记录类型信息引用计数等等。

Memcached 服务特点及工作原理是什么?

a、完全基于内存缓存的

b、节点之间相互独立

c、C/S 模式架构，C 语言编写，总共 2000 行代码。

d、异步 I/O 模型，使用 libevent 作为事件通知机制。

e、被缓存的数据以 key/value 键值对形式存在的。

f、全部数据存放于内存中，无持久性存储的设计，重启服务器，内存里的数据会丢失。

g、当内存中缓存的数据容量达到启动时设定的内存值时，就自动使用 LRU 算法删除过期的缓存数据。

h、可以对存储的数据设置过期时间，这样过期后的数据自动被清除，服务本身不会监控过期，而是在访问的时候查看 key 的时间戳,判断是否过期。

j、memcache 会对设定的内存进行分块，再把块分组，然后再提供服务。

如何实现集群中的 session 共享存储?

Session 是运行在一台服务器上的，所有的访问都会到达我们的唯一服务器上，这样我们可以根据客户端传来的 sessionID，来获取 session，或在对应 Session 不存在的情况下（session 生命周期到了/用户第一次登录），创建一个新的 Session；但是，如果我们在集群环境下，假设我们有两台服务器 A，B，用户的请求会由 Nginx 服务器进行转发（别的方案也是同理），用户登录时，Nginx 将请求转发至服务器 A 上，A 创建了新的 session，并将 SessionID 返回给客户端，用户在浏览其他页面时，客户端验证登录状态，Nginx 将请求转发至服务器 B，由于 B 上并没有对应客户端发来 sessionId 的 session，所以会重新创建一个新的 session，并且再将这个新的 sessionID 返回给客户

端，这样，我们可以想象一下，用户每一次操作都有 1/2 的概率进行再次的登录，这样不仅对用户体验特别差，还会让服务器上的 session 激增，加大服务器的运行压力。

为了解决集群环境下的 seesion 共享问题，共有 4 种解决方案：

1.粘性 session

粘性 session 是指 Ngnix 每次都将同一用户的所有请求转发至同一台服务器上，即将用户与服务器绑定。

2.服务器 session 复制

即每次 session 发生变化时，创建或者修改，就广播给所有集群中的服务器，使所有的服务器上的 session 相同。

3.session 共享

缓存 session，使用 redis， memcached。

4.session 持久化

将 session 存储至数据库中，像操作数据一样才做 session。

memcached 与 redis 的区别?

1、Redis 不仅仅支持简单的 k/v 类型的数据，同时还提供 list，set，zset，hash 等数据结构的存储。而 memcache 只支持简单数据类型，需要客户端自己处理复杂对象

2、Redis 支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用（PS：持久化在 rdb、aof)。

3、由于 Memcache 没有持久化机制，因此宕机所有缓存数据失效。Redis 配置为持久化，宕机重启后，将自动加载宕机时刻的数据到缓存系统中。具有更好的灾备机制。

4、Memcache 可以使用 Magent 在客户端进行一致性 hash 做分布式。Redis 支持在服务器端做分布式（PS:Twemproxy/Codis/Redis-cluster 多种分布式实现方式）

5、Memcached 的简单限制就是键（key）和 Value 的限制。最大键长为 250 个字符。可以接受的储存数据不能超过 1MB（可修改配置文件变大），因为这是典型 slab 的最大值，不适合虚拟机使用。而 Redis 的 Key 长度支持到 512k。

6、Redis 使用的是单线程模型，保证了数据按顺序提交。Memcache 需要使用 cas 保证数据一致性。CAS（Check and Set）是一个确保并发一致性的机制，属于“乐观锁”范畴；原理很简单：拿版本号，操作，对比版本号，如果一致就操作，不一致就放弃任何操作

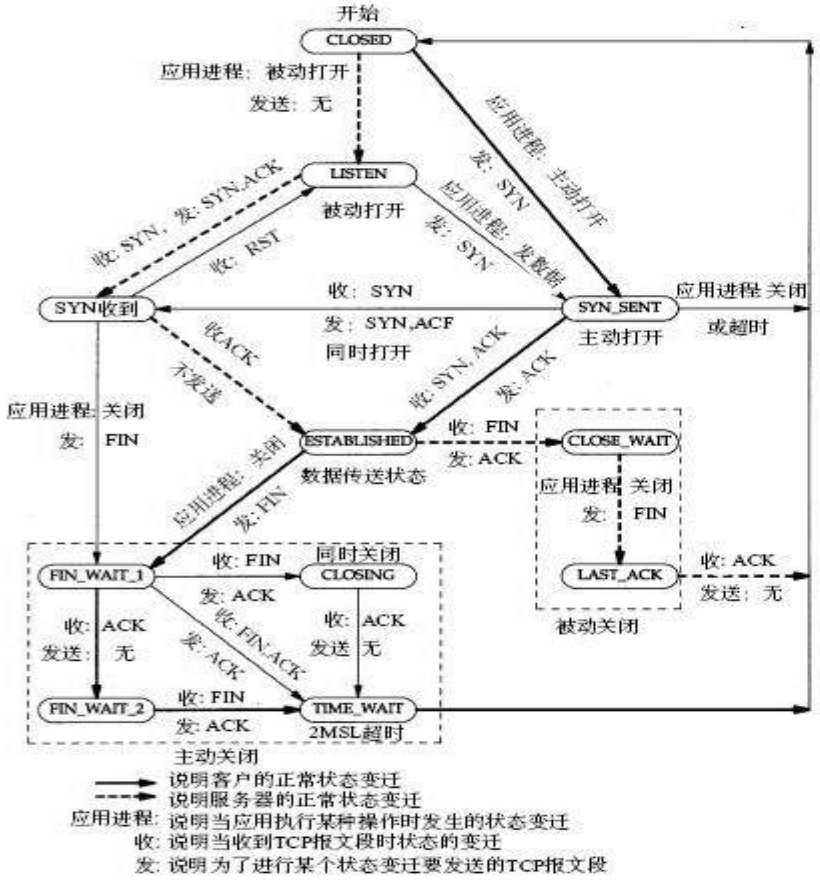
cpu 利用。由于 Redis 只使用单核，而 Memcached 可以使用多核，所以平均每一个核上 Redis 在存储小数据时比 Memcached 性能更 高。而在 100k 以上的数据中，Memcached 性能要高于 Redis 。

7、memcache 内存管理：使用 Slab Allocation。原理相当简单，预先分配一系列大小固定的组，然后根据数据大小选择最合适的块存储。避免了内存碎片。(缺点：不能变长,浪费了一定空间)memcached 默认情况下下一个 slab 的最大值为前一个的 1.25 倍。

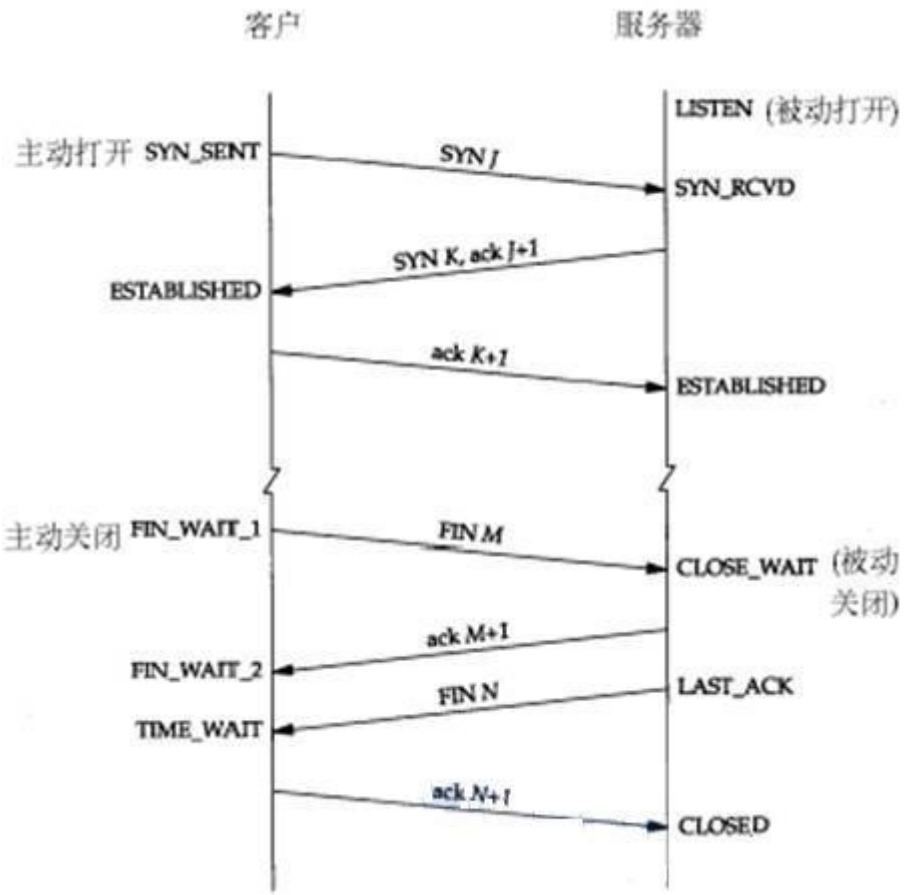
8、redis 内存管理： Redis 通过定义一个数组来记录所有的内存分配情况， Redis 采用的是包装的 malloc/free，相较于 Memcached 的内存 管理方法来说，要简单很多。由于 malloc 首先以链表的方式搜索已管理的内存中可用的空间分配，导致内存碎片比较多

六、 TCP 协议三次握手连接四次挥手断开和 DOS 攻击

TCP 连接的状态图



TCP 建立连接的三次握手过程，以及关闭连接的四次握手过程



贴一个 telnet 建立连接，断开连接的使用 wireshark 捕获的 packet 截图。

Source	Destination	Protocol	Info
172.29.132.60	172.29.21.25	TCP	proxy-gateway > telnet [SYN] Seq=0 Win=65535 Len=0 MSS=1460 SACK_PERM=1
172.29.21.25	172.29.132.60	TCP	telnet > proxy-gateway [SYN, ACK] Seq=0 Ack=1 Win=49640 Len=0 MSS=1460 SACK_PERM=1
172.29.132.60	172.29.21.25	TCP	proxy-gateway > telnet [ACK] Seq=1 Ack=1 Win=65535 Len=0
172.29.132.60	172.29.21.25	TCP	proxy-gateway > telnet [FIN, ACK] Seq=76 Ack=80 Win=65456 Len=0
172.29.21.25	172.29.132.60	TCP	telnet > proxy-gateway [ACK] Seq=80 Ack=77 Win=49640 Len=0
172.29.21.25	172.29.132.60	TCP	telnet > proxy-gateway [FIN, ACK] Seq=80 Ack=77 Win=49640 Len=0
172.29.132.60	172.29.21.25	TCP	proxy-gateway > telnet [ACK] Seq=77 Ack=81 Win=65456 Len=0

1、建立连接协议（三次握手）

（1）客户 端发送一个带 SYN 标志的 TCP 报文到服务器。这是三次握手过程中的报文 1。

（2） 服务器端回应客户端的，这是三次握手中的第 2 个报文，这个报文同时带 ACK 标志和 SYN 标志。因此它表示对刚才客户端 SYN 报文的回应；同时又标志 SYN 给客户端，询

问客户端是否准备好进行数据通 讯。

（3） 客户必须再次回应服务段一个 ACK 报文，这是报文段 3。

2、连接终止协议（四次握手）

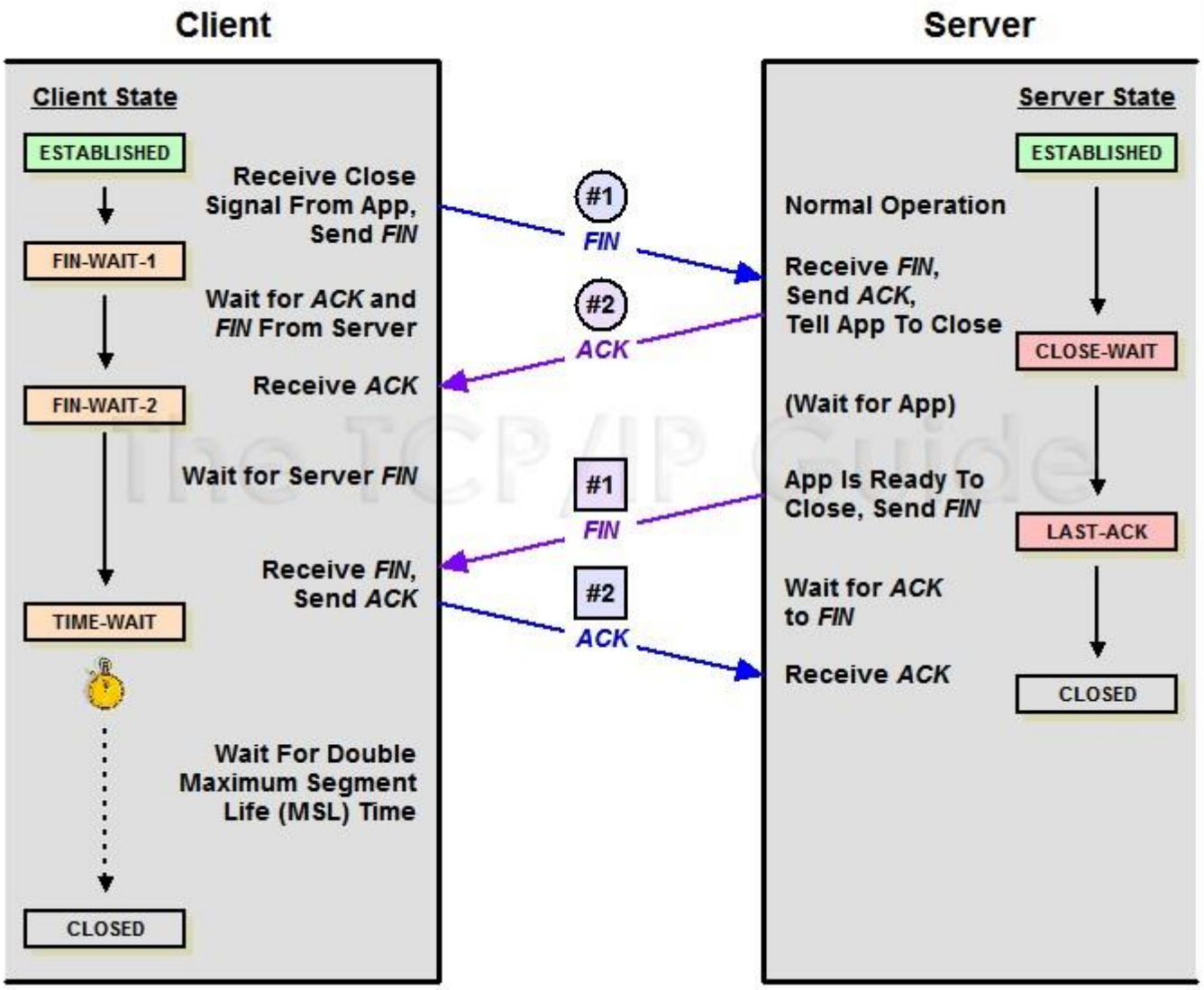
由于 TCP 连 接是全双工的，因此每个方向都必须单独进行关闭。这原则是当一方完成它的数据发送任务后就能发送一个 FIN 来终止这个方向的连接。收到一个 FIN 只意味着这一方向上没有数据流动，一个 TCP 连接在收到一个 FIN 后仍能发送数据。首先进行关闭的一方将执行主动关闭，而另一方执行被动关闭。

（1） TCP 客 户端发送一个 FIN，用来关闭客户到服务器的数据传送（报文段 4）。

（2） 服务器收到这个 FIN，它发回一个 ACK，确认序号为收到的序号加 1（报文段 5）。和 SYN 一样，一个 FIN 将占用一个序号。

（3） 服务器关闭客户端的连接，发送一个 FIN 给客户端（报文段 6）。

（4） 客户段发回 ACK 报文确认，并将确认序号设置为收到序号加 1（报文段 7）。



CLOSED：这个没什么好说的了，表示初始状态。

LISTEN：这个也是非常容易理解的一个状态，表示服务器端的某个 SOCKET 处 于监听状态，可以接受连接了。

SYN_RCVD：这个状态表示接受到了 SYN 报文，在正常情况下，这个状态是服务器端的 SOCKET 在建立 TCP 连接时的三次握手会话过程中的一个中间状态，很短暂，基本上用 netstat 你是很难看到这种状态的，除非你特意写了一个客户端测试程序，故意将三次 TCP 握手过程中最后一个 ACK 报文不予发送。因此这种状态时，当收到客户端的 ACK 报文 后，

它会进入到 ESTABLISHED 状态。

SYN_SENT: 这个状态与 SYN_RCVD 遥想呼应，当客户端 SOCKET 执行 CONNECT 连接时，它首先发送 SYN 报文，因此也随即它会进入到了 SYN_SENT 状态，并等待服务端的发送三次握手中的第 2 个报文。SYN_SENT 状态表示客户端已发送 SYN 报文。

ESTABLISHED：这个容易理解了，表示连接已经建立了。

FIN_WAIT_1: 这个状态要好好解释一下，其实 FIN_WAIT_1 和 FIN_WAIT_2 状态的真正含义都是表示等待对方的 FIN 报文。而这两种状态的区别是：FIN_WAIT_1 状态实际上是当 SOCKET 在 ESTABLISHED 状态时，它想主动关闭连接，向对方发送了 FIN 报文，此时该 SOCKET 即进入到 FIN_WAIT_1 状态。而当对方回应 ACK 报文后，则进入到 FIN_WAIT_2 状态，当然在实际的正常情况下，无论对方何种情况下，都应该马上回应 ACK 报文，所以 FIN_WAIT_1 状态一般是比较难见到的，而 FIN_WAIT_2 状态还有时常常可以用 netstat 看到。

FIN_WAIT_2：上面已经详细解释了这种状态，实际上 FIN_WAIT_2 状态下的 SOCKET，表示半连接，也即有一方要求 close 连接，但另外还告诉对方，我暂时还有点数据需要传送给你，稍后再关闭连接。

TIME_WAIT: 表示收到了对方的 FIN 报文，并发送出了 ACK 报文，就等 2MSL 后即可回到 CLOSED 可用状态了。如果 FIN_WAIT_1 状态下，收到了对方同时带 FIN 标志和 ACK 标志的报文时，可以直接进入到 TIME_WAIT 状态，而无须经过 FIN_WAIT_2 状态。

CLOSING: 这种状态比较特殊，实际情况中应该是很少见，属于一种比较罕见的例外状态。正常情况下，当你发送 FIN 报文后，按理来说是应该先收到（或同时收到）对方的 ACK 报文，再收到对方的 FIN 报文。但是 CLOSING 状态表示你发送 FIN 报文后，并没有收到对方的 ACK 报文，反而却也收到了对方的 FIN 报文。什么情况下会出现此种情况呢？其实细

想一下，也不难得出结论：那就是如果双方几乎在同时 close 一个 SOCKET 的话，那么就出现了双方同时发送 FIN 报文的情况，也即会出现 CLOSING 状态，表示双方都正在关闭 SOCKET 连接。

CLOSE_WAIT: 这种状态的含义其实是表示在等待关闭。怎么理解呢？当对方 close 一个 SOCKET 后发送 FIN 报文给自己，你系统毫无疑问地会回应一个 ACK 报文给对方，此时则进入到 CLOSE_WAIT 状态。接下来呢，实际上你真正需要考虑的事情是察看你是否还有数据发送给对方，如果没有的话，那么你也就可以 close 这个 SOCKET，发送 FIN 报文给对方，也即关闭连接。所以你在 CLOSE_WAIT 状态下，需要完成的事情是等待你去关闭连接。

LAST_ACK: 这个状态还是比较容易好理解的，它是被动关闭一方在发送 FIN 报文后，最后等待对方的 ACK 报文。当收到 ACK 报文后，也即可以进入到 CLOSED 可用状态了。

补充：

a. 默认情况下(不改变 socket 选项)，当你调用 close(or closesocket，以下说 close 不再重复)时，如果发送缓冲中还有数据，TCP 会继续把数据发送完。

b. 发送了 FIN 只是表示这端不能继续发送数据(应用层不能再调用 send 发送)，但是还可以接收数据。

c. 应用层如何知道对端关闭？通常，在最简单的阻塞模型中，当你调用 recv 时，如果返回 0，则表示对端关闭。在这个时候通常的做法就是也调用 close，那么 TCP 层就发送 FIN，继续完成四次握手。如果你不调用 close，那么对端就会处于 FIN_WAIT_2 状态，而本端则会处于 CLOSE_WAIT 状态。这个可以写代码试试。

d. 在很多时候，TCP 连接的断开都会由 TCP 层自动进行，例如你 CTRL+C 终止你的程序，TCP 连接依然会正常关闭，你可以写代码试试。

1、 为什么建立连接协议是三次握手，而关闭连接却是四次握手呢?

这是因为服务端的 LISTEN 状态下的 SOCKET 当收到 SYN 报文的建连请求后，它可以把 ACK 和 SYN（ACK 起应答作用，而 SYN 起同步作用）放在一个报文里来发送。但关闭连接时，当收到对方的 FIN 报文通知时，它仅仅表示对方没有数据发送给你了；但未必你所有的数据都全部发送给对方了，所以你未必会马上会关闭 SOCKET,也即你可能还需要发送一些数据给对方之后，再发送 FIN 报文给对方来表示你同意现在可以关闭连接了，所以它这里的 ACK 报文和 FIN 报文多数情况下都是分开发送的。

2、 为什么 TIME_WAIT 状态还需要等 2MSL 后才能返回到 CLOSED 状态?

什么是 2MSL？MSL 即 Maximum Segment Lifetime，也就是报文最大生存时间，引用《TCP/IP 详解》中的话： “它(MSL)是任何报文段被丢弃前在网络内的最长时间。”那么，2MSL 也就是这个时间的 2 倍，当 TCP 连接完成四个报文段的交换时，主动关闭的一方将继续等待一定时间(2-4 分钟)，即使两端的应用程序结束。例如在上面的 telnet 程序客户端关闭后，使用 netstat 查看的结果:

C:\>netstat -na | find "172.29.21.25"

TCP 172.29.132.60:2795 172.29.21.25:23 TIME_WAIT

为什么需要这个 2MSL 呢，

第一，虽然双方都同意关闭连接了，而且握手的 4 个报文也都协调和发送完毕，按理可以直接回到 CLOSED 状态（就好比从 SYN_SEND 状态到 ESTABLISH 状态那样）；但是因为我们必须要假想网络是不可靠的，你无法保证你最后发送的 ACK 报文会一定被对方收到，

因此对方处于 LAST_ACK 状态下的 SOCKET 可能会因为超时未收到 ACK 报文，而重发 FIN 报文，所以这个 TIME_WAIT 状态的作用就是用来重发可能丢失的 ACK 报文。

第二，报文可能会被混淆，意思是说，其他时候的连接可能会被当作本次的连接。直接引用《The TCP/IP Guide》的说法：The second is to provide a “buffering period” between the end of this connection and any subsequent ones. If not for this period, it is possible that packets from different connections could be mixed, creating confusion.

当某个连接的一端处于 TIME_WAIT 状态时，该连接将不能再被使用。事实上，对于我们比较有现实意义的是，这个端口将不能再被使用。某个端口处于 TIME_WAIT 状态(其实应该是这个连接)时，这意味着这个 TCP 连接并没有断开(完全断开)，那么，如果你 bind 这个端口，就会失败。对于服务器而言，如果服务器突然 crash 掉了，那么它将无法在 2MSL 内重新启动，因为 bind 会失败。解决这个问题的一个方法就是设置 socket 的 SO_REUSEADDR 选项。这个选项意味着你可以重用一个地址。

当建立一个 TCP 连接时，服务器端会继续用原有端口监听，同时用这个端口与客户端通信。而客户端默认情况下会使用一个随机端口与服务器端的监听端口通信。有时候，为了服务器端的安全性，我们需要对客户端进行验证，即限定某个 IP 某个特定端口的客户端。客户端可以使用 bind 来使用特定的端口。对于服务器端，当设置了 SO_REUSEADDR 选项时，它可以在 2MSL 内启动并 listen 成功。但是对于客户端，当使用 bind 并设置 SO_REUSEADDR 时，如果在 2MSL 内启动，虽然 bind 会成功，但是在 windows 平台上 connect 会失败。而在 linux 上则不存在这个问题。(我的实验平台：winxp, ubuntu7.10)

要解决 windows 平台的这个问题，可以设置 SO_LINGER 选项。SO_LINGER 选项决定调用 close 时 TCP 的行为。SO_LINGER 涉及到 linger 结构体，如果设置结构体中 l_onoff 为非 0，l_linger 为 0，那么调用 close 时 TCP 连接会立刻断开，TCP 不会将发送缓冲中未发送的数据发送，而是立即发送一个 RST 报文给对方，这个时候 TCP 连接(关闭时)就不会进入 TIME_WAIT 状态。如你所见，这样做虽然解决了问题，但是并不安全。通过以上方式设置 SO_LINGER 状态，等同于设置 SO_DONTLINGER 状态。

当 TCP 连接发生一些物理上的意外情况时，例如网线断开，linux 上的 TCP 实现会依然认为该连接有效，而 windows 则会在一定时间后返回错误信息。这似乎可以通过设置 SO_KEEPALIVE 选项来解决，不过不知道这个选项是否对于所有平台都有效。

3. 为什么不能用两次握手进行连接?

我们知道，3 次握手完成两个重要的功能，既要双方做好发送数据的准备工作(双方都知道彼此已准备好)，也要允许双方就初始序列号进行协商，这个序列号在握手过程中被发送和确认。

现在把三次握手改成仅需要两次握手，死锁是可能发生的。作为例子，考虑计算机 S 和 C 之间的通信，假定 C 给 S 发送一个连接请求分组，S 收到了这个分组，并发送了确认应答分组。按照两次握手的协定，S 认为连接已经成功地建立了，可以开始发送数据分组。可是，C 在 S 的应答分组在传输中被丢失的情况下，将不知道 S 是否已准备好，不知道 S 建立什么样的序列号，C 甚至怀疑 S 是否收到自己的连接请求分组。在这种情况下，C 认为连接还未建立成功，将忽略 S 发来的任何数据分组，只等待连接确认应答分组。而 S 在发出的分

组超时后，重复发送同样的分组。这样就形成了死锁。

DoS 攻击

DoS 攻击、DDoS 攻击和 DRDoS 攻击相信大家已经早有耳闻了吧!DoS 是 Denial of Service 的简写就是拒绝服务，而 DDoS 就是 Distributed Denial of Service 的简写就是分布式拒绝服务,而 DRDoS 就是 Distributed Reflection Denial of Service 的简写,这是分布反射式拒绝服务的意思。

不过这 3 中攻击方法最厉害的还是 DDoS，那个 DRDoS 攻击虽然是新近出的一种攻击方法，但它只是 DDoS 攻击的变形，它的唯一不同就是不用占领大量的“肉鸡”。这三种方法都是利用 TCP 三次握手的漏洞进行攻击的，所以对它们的防御办法都是差不多的。

DoS 攻击是最早出现的，它的攻击方法说白了就是单挑，是比谁的机器性能好、速度快。但是现在的科技飞速发展，一般的网站主机都有十几台主机，而且各个主机的处理能力、内存大小和网络速度都有飞速的发展，有的网络带宽甚至超过了千兆级别。这样我们的一对一单挑式攻击就没有什么作用了，搞不好自己的机子就会死掉。举个这样的攻击例子，假如你的机器每秒能够发送 10 个攻击用的数据包，而被你攻击的机器(性能、网络带宽都是顶尖的)每秒能够接受并处理 100 攻击数据包，那样的话，你的攻击就什么用处都没有了，而且非常有死机的可能。要知道，你若是发送这种 1Vs1 的攻击，你的机器的 CPU 占用率是 90%以上的，你的机器要是配置不够高的话，那你就死定了。

不过，科技在发展，黑客的技术也在发展。正所谓道高一尺，魔高一仗。经过无数次当机，黑客们终于又找到一种新的 DoS 攻击方法，这就是 DDoS 攻击。它的原理说白了就是群殴，用好多的机器对目标机器一起发动 DoS 攻击，但这不是很多黑客一起参与的，这种攻击只

是由一名黑客来操作的。这名黑客不是拥有很多机器，他是通过他的机器在网络上占领很多的“肉鸡”，并且控制这些“肉鸡”来发动 DDoS 攻击，要不然怎么叫做分布式呢。还是刚才的那个例子，你的机器每秒能发送 10 攻击数据包，而被攻击的机器每秒能够接受 100 的数据包，这样你的攻击肯定不会起作用，而你再用 10 台或更多的机器来对被攻击目标的机器进行攻击的话，嘿嘿!结果我就不说了。

DRDoS 分布反射式拒绝服务攻击这是 DDoS 攻击的变形，

解释:

SYN:(Synchronize sequence numbers)用来建立连接，在连接请求中，SYN=1，ACK=0，连接响应时，SYN=1，ACK=1。即，SYN 和 ACK 来区分 Connection Request 和 Connection Accepted。

RST:(Reset the connection)用于复位因某种原因引起出现的错误连接，也用来拒绝非法数据和请求。如果接收到 RST 位时候，通常发生了某些错误。

ACK:(Acknowledgment field significant)置 1 时表示确认号(Acknowledgment Number)为合法，为 0 的时候表示数据段不包含确认信息，确认号被忽略。

设我们要准备建立连接，服务器正处于正常的接听状态。

第一步:我们也就是客户端发送一个带 SYN 位的请求，向服务器表示需要连接，假设请求包的序列号为 10，那么则为:SYN=10，ACK=0，然后等待服务器的回应。

第二步:服务器接收到这样的请求包后，查看是否在接听的是指定的端口，如果不是就发送 RST=1 回应，拒绝建立连接。如果接收请求包，那么服务器发送确认回应，SYN 为服务器的一个内码，假设为 100，ACK 位则是客户端的请求序号加 1，本例中发送的数据是:SYN=100，ACK=11，用这样的数据回应给我们。向我们表示，服务器连接已经准备好

了，等待我们的确认。这时我们接收到回应后，分析得到的信息，准备发送确认连接信号到服务器。

第三步:我们发送确认建立连接的信息给服务器。确认信息的 SYN 位是服务器发送的 ACK 位，ACK 位是服务器发送的 SYN 位加 1。即:SYN=11，ACK=101。

这样我们的连接就建立起来了。

DDoS 究竟如何攻击?目前最流行也是最好用的攻击方法就是使用 SYN-Flood 进行攻击，SYN-Flood 也就是 SYN 洪水攻击。SYN-Flood 不会完成 TCP 三次握手的第三步，也就是不发送确认连接的信息给服务器。这样，服务器无法完成第三次握手，但服务器不会立即放弃，服务器会不停的重试并等待一定的时间后放弃这个未完成的连接，这段时间叫做 SYN timeout，这段时间大约 30 秒-2 分钟左右。若是一个用户在连接时出现问题导致服务器的一个线程等待 1 分钟并不是什么大不了的问题，但是若有人用特殊的软件大量模拟这种情况，那后果就可想而知了。一个服务器若是处理这些大量的半连接信息而消耗大量的系统资源和网络带宽，这样服务器就不会再有空余去处理普通用户的正常请求(因为客户的正常请求比率很小)。这样这个服务器就无法工作了，这种攻击就叫做:SYN-Flood 攻击。

到目前为止，进行 DDoS 攻击的防御还是比较困难的。首先，这种攻击的特点是它利用了 TCP/IP 协议的漏洞，除非你不用 TCP/IP，才有可能完全抵御住 DDoS 攻击。不过这不等于我们就没有办法阻挡 DDoS 攻击，我们可以尽力来减少 DDoS 的攻击。下面就是一些防御方法:

1.确保服务器的系统文件是最新的版本，并及时更新系统补丁。

2.关闭不必要的服务。

3.限制同时打开的 SYN 半连接数目。

4.缩短 SYN 半连接的 time out 时间。

5.正确设置防火墙

6.禁止对主机的非开放服务的访问

7.限制特定 IP 地址的访问

8.启用防火墙的防 DDoS 的属性

9.严格限制对外开放的服务器的向外访问

10.运行端口映射程序祸端口扫描程序，要认真检查特权端口和非特权端口。

11.认真检查网络设备和主机/服务器系统的日志。只要日志出现漏洞或是时间变更，那这台机器就可能遭到了攻击。

12.限制在防火墙外与网络文件共享。这样会给黑客截取系统文件的机会，主机的信息暴露给黑客，无疑是给了对方入侵的机会。

七、 JVM 调优

(一) -- 一些概念

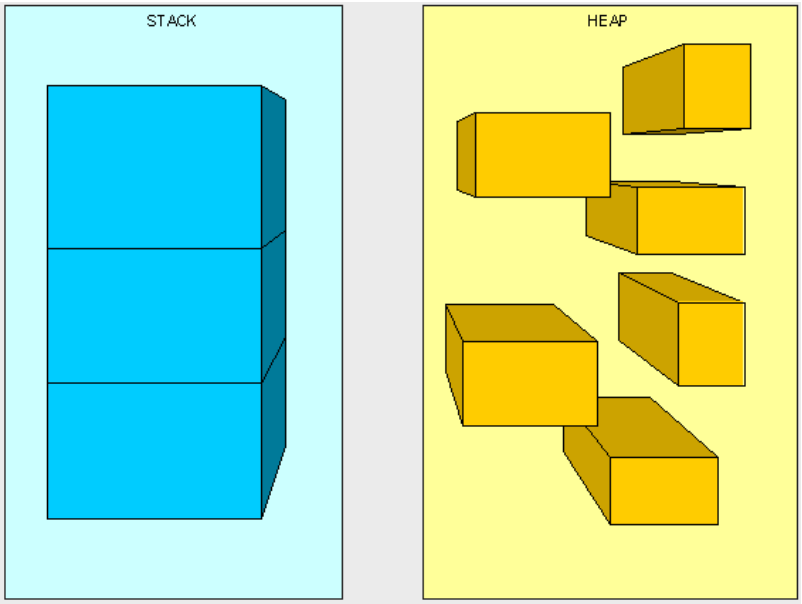
Java 虚拟机中，数据类型可以分为两类：**基本类型**和**引用类型**。基本类型的变量保存原始值，即：他代表的值就是数值本身；而引用类型的变量保存引用值。“引用值”代表了某个对象的引用，而不是对象本身，对象本身存放在这个引用值所表示的地址的位置。

基本类型包括：byte,short,int,long,char,float,double,Boolean,returnAddress

引用类型包括：**类类型**，**接口类型**和**数组**。(String 也是引用类型)

堆与栈

堆和栈是程序运行的关键，很有必要把他们的关系说清楚。



栈是运行时的单位，而堆是存储的单位。

栈解决程序的运行问题，即程序如何执行，或者说如何处理数据；堆解决的是数据存储的问题，即数据怎么放、放在哪儿。

在 Java 中一个线程就会相应有一个线程栈与之对应，这点很容易理解，因为不同的线程执行逻辑有所不同，因此需要一个独立的线程栈。而堆则是所有线程共享的。栈因为是运行单位，因此里面存储的信息都是跟当前线程（或程序）相关信息的。包括局部变量、程序运行状态、方法返回值等等；而堆只负责存储对象信息。

为什么要把堆和栈区分出来呢？栈中不是也可以存储数据吗?

第一，从软件设计的角度看，栈代表了处理逻辑，而堆代表了数据。这样分开，使得处理逻辑更为清晰。分而治之的思想。这种隔离、模块化的思想在软件设计的方方面面都有体现。

第二，堆与栈的分离，使得堆中的内容可以被多个栈共享（也可以理解为多个线程访问同一个对象）。这种共享的收益是很多的。一方面这种共享提供了一种有效的数据交互方式(如：共享内存)，另一方面，堆中的共享常量和缓存可以被所有栈访问，节省了空间。

第三，栈因为运行时的需要，比如保存系统运行的上下文，需要进行地址段的划分。由于栈只能向上增长，因此就会限制住栈存储内容的能力。而堆不同，堆中的对象是可以根据需要动态增长的，因此栈和堆的拆分，使得动态增长成为可能，相应栈中只需记录堆中的一个地址即可。

第四，面向对象就是堆和栈的完美结合。其实，面向对象方式的程序与以前结构化的程序在执行上没有任何区别。但是，面向对象的引入，使得对待问题的思考方式发生了改变，而更接近于自然方式的思考。当我们把对象拆开，你会发现，对象的属性其实就是数据，存放在堆中；而对象的行为（方法），就是运行逻辑，放在栈中。我们在编写对象的时候，其实即编写了数据结构，也编写的处理数据的逻辑。不得不承认，面向对象的设计，确实很美。

在 Java 中，Main 函数就是栈的起始点，也是程序的起始点。

程序要运行总是有一个起点的。同 C 语言一样，java 中的 Main 就是那个起点。无论什么 java 程序，找到 main 就找到了程序执行的入口：）

堆中存什么？栈中存什么?

堆中存的是对象。栈中存的是基本数据类型和堆中对象的引用。一个对象的大小是不可估计的，或者说是可以动态变化的，但是在栈中，一个对象只对应了一个 4btye 的引用（堆栈分离的好处：））。

为什么不把基本类型放堆中呢？因为其占用的空间一般是 1~8 个字节——需要空间比较少，而且因为是基本类型，所以不会出现动态增长的情况——长度固定，因此栈中存储就够了，如果把他存在堆中是没有什么意义的（还会浪费空间，后面说明）。可以这么说，基本类型和对象的引用都是存放在栈中，而且都是几个字节的一个数，因此在程序运行时，他们的处理方式是统一的。但是基本类型、对象引用和对象本身就有所区别了，因为一个是栈中的数据一个是堆中的数据。最常见的一个问题就是，Java 中参数传递时的问题。

Java 中的参数传递时传值呢？还是传引用？

要说明这个问题，先要明确两点：

1. **不要试图与 C 进行类比，Java 中没有指针的概念**

2. **程序运行永远都是在栈中进行的，因而参数传递时，只存在传递基本类型和对象引用的问题。**不会直接传对象本身。

明确以上两点后。Java 在方法调用传递参数时，因为没有指针，所以**它都是进行传值调用**（这点可以参考 C 的传值调用）。因此，很多书里面都说 Java 是进行传值调用，这点没有问题，而且也简化的 C 中复杂性。

*但是传引用的错觉是如何造成的呢？*在运行栈中，基本类型和引用的处理是一样的，都是传值，所以，如果是传引用的方法调用，也同时可以理解为“传引用值”的传值调用，即引用的处理跟基本类型是完全一样的。但是当进入被调用方法时，被传递的这个引用的值，被程序解释（或者查找）到堆中的对象，这个时候才对应到真正的对象。如果此时进行修改，修改的是引用对应的对象，而不是引用本身，即：修改的是堆中的数据。所以这个修改是可以保持的了。

对象，从某种意义上说，是由基本类型组成的。可以把一个对象看作为一棵树，对象的属性如果还是对象，则还是一颗树（即非叶子节点），基本类型则为树的叶子节点。

程序参数传递时，被传递的值本身都是不能进行修改的，但是，如果这个值是一个非叶子节点（即一个对象引用），则可以修改这个节点下面的所有内容。

堆和栈中，栈是程序运行最根本的东西。程序运行可以没有堆，但是不能没有栈。而堆是为栈进行数据存储服务，说白了堆就是一块共享的内存。不过，正是因为堆和栈的分离的思想，才使得 Java 的垃圾回收成为可能。

Java 中，栈的大小通过-Xss 来设置，当栈中存储数据比较多时，需要适当调大这个值，否则会出现 java.lang.StackOverflowError 异常。常见的出现这个异常的是无法返回的递归，因为此时栈中保存的信息都是方法返回的记录点。

1. （二）一些概念

Java 对象的大小

基本数据的类型的大小是固定的，这里就不多说了。对于非基本类型的 Java 对象，其大小就值得商榷。

在 Java 中，**一个空 Object 对象的大小是 8byte**，这个大小只是保存堆中一个没有任何属性的对象的大小。看下面语句：

```
Object ob = new Object();
```

这样在程序中完成了一个 Java 对象的生命，但是它所占的空间为：**4byte+8byte**。4byte 是上面部分所说的 Java 栈中保存引用的所需要的空间。而那 8byte 则是 Java 堆中对象的

信息。因为所有的 Java 非基本类型的对象都需要默认继承 Object 对象，因此不论什么样的 Java 对象，其大小都必须是大于 8byte。

有了 Object 对象的大小，我们就可以计算其他对象的大小了。

```
Class NewObject {
    int count;
    boolean flag;
    Object ob;
}
```

其大小为：空对象大小(8byte)+int 大小(4byte)+Boolean 大小(1byte)+空 Object 引用的大小(4byte)=17byte。但是因为 Java 在对对象内存分配时都是以 8 的整数倍来分，因此大于 17byte 的最接近 8 的整数倍的是 24，因此此对象的大小为 24byte。

这里需要注意一下**基本类型的包装类型的大小**。因为这种包装类型已经成为对象了，因此需要把他们作为对象来看待。包装类型的大小至少是 12byte（声明一个空 Object 至少需要的空间），而且 12byte 没有包含任何有效信息，同时，因为 Java 对象大小是 8 的整数倍，因此**一个基本类型包装类的大小至少是 16byte**。这个内存占用是很恐怖的，它是使用基本类型的 N 倍（N>2），有些类型的内存占用更是夸张（随便想下就知道了）。因此，可能的话应尽量少使用包装类。在 JDK5.0 以后，因为加入了自动类型装换，因此，Java 虚拟机会在存储方面进行相应的优化。

引用类型

对象引用类型分为**强引用、软引用、弱引用和虚引用。**

强引用:就是我们一般声明对象是时虚拟机生成的引用，强引用环境下，垃圾回收时需要严格判断当前对象是否被强引用，如果被强引用，则不会被垃圾回收

软引用:软引用一般被做为缓存来使用。与强引用的区别是，软引用在垃圾回收时，虚拟机会根据当前系统的剩余内存来决定是否对软引用进行回收。如果剩余内存比较紧张，则虚拟机会回收软引用所引用的空间；如果剩余内存相对富裕，则不会进行回收。换句话说，虚拟机在发生 OutOfMemory 时，肯定是没有软引用存在的。

弱引用:弱引用与软引用类似，都是作为缓存来使用。但与软引用不同，弱引用在进行垃圾回收时，是一定会被回收掉的，因此其生命周期只存在于一个垃圾回收周期内。

强引用不用说，我们系统一般在使用时都是用的强引用。而“软引用”和“弱引用”比较少见。他们一般被作为缓存使用，而且一般是在内存大小比较受限的情况下做为缓存。因为如果内存足够大的话，可以直接使用强引用作为缓存即可，同时可控性更高。因而，他们常见的是被使用在桌面应用系统的缓存。

2. （三）-基本垃圾回收算法

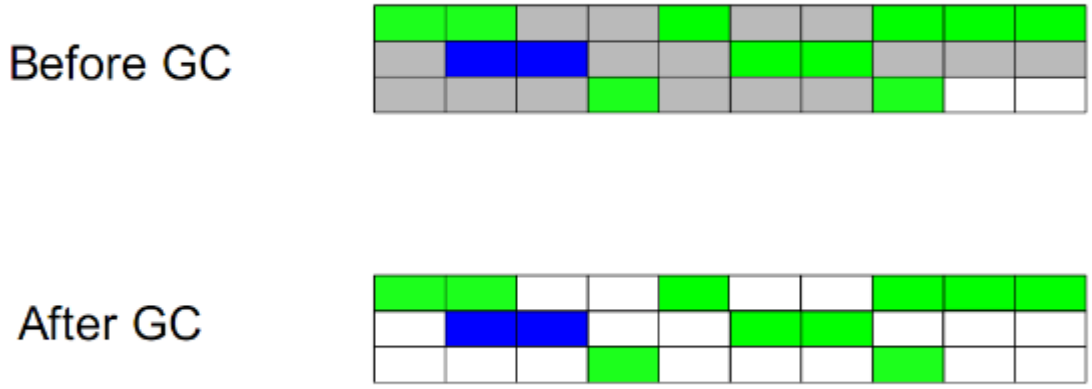
可以从不同的的角度去划分垃圾回收算法：

按照基本回收策略分

引用计数（Reference Counting）：

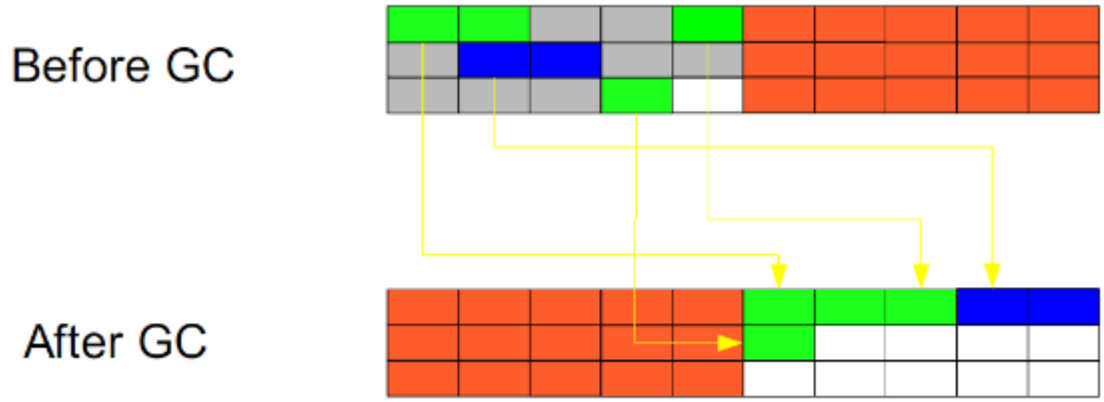
比较古老的回收算法。原理是此对象有一个引用，即增加一个计数，删除一个引用则减少一个计数。垃圾回收时，只用收集计数为 0 的对象。此算法最致命的是无法处理循环引用的问题。

标记-清除（Mark-Sweep）：



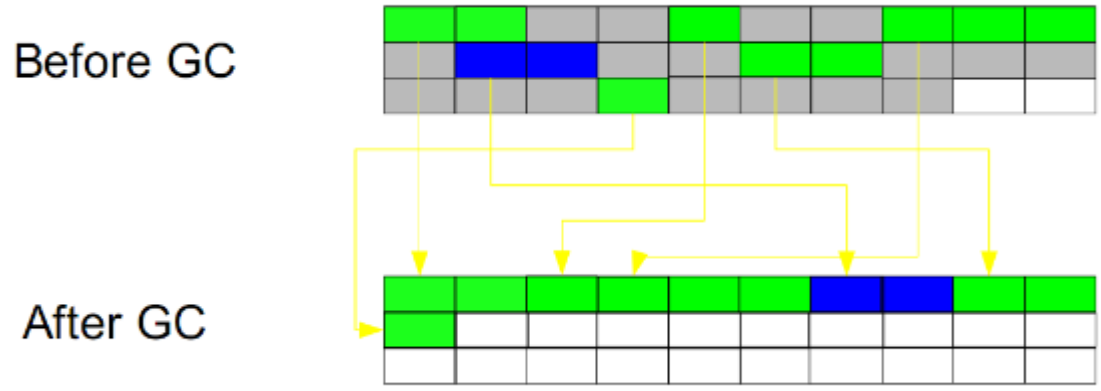
此算法执行分两阶段。第一阶段从引用根节点开始标记所有被引用的对象，第二阶段遍历整个堆，把未标记的对象清除。此算法需要暂停整个应用，同时，会产生内存碎片。

复制（Copying）：



此算法把内存空间划为两个相等的区域，每次只使用其中一个区域。垃圾回收时，遍历当前使用区域，把正在使用中的对象复制到另外一个区域中。此算法每次只处理正在使用中的对象，因此复制成本比较小，同时复制过去以后还能进行相应的内存整理，不会出现“碎片”问题。当然，此算法的缺点也是很明显的，就是需要两倍内存空间。

标记-整理（Mark-Compact）：



此算法结合了“标记-清除”和“复制”两个算法的优点。也是分两阶段，第一阶段从根节点开始标记所有被引用对象，第二阶段遍历整个堆，把清除未标记对象并且把存活对象“压缩”到堆的其中一块，按顺序排放。此算法避免了“标记-清除”的碎片问题，同时也避免了“复制”算法的空间问题。

按分区对待的方

增量收集（Incremental Collecting）:实时垃圾回收算法，即：在应用进行的同时进行垃圾回收。不知道什么原因 JDK5.0 中的收集器没有使用这种算法的。

分代收集（Generational Collecting）:基于对对象生命周期分析后得出的垃圾回收算法。把对象分为年青代、年老代、持久代，对不同生命周期的对象使用不同的算法（上述方式中的一个）进行回收。现在的垃圾回收器（从 J2SE1.2 开始）都是使用此算法的。

按系统线程分

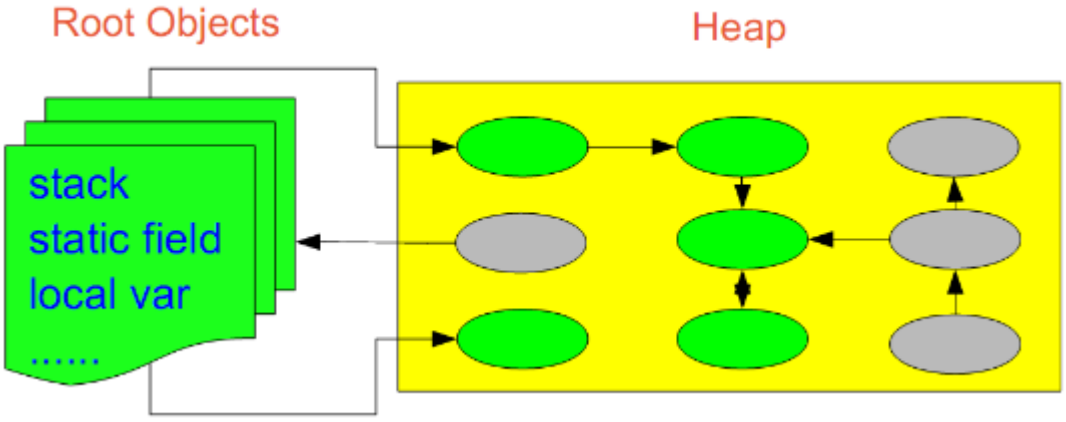
串行收集:串行收集使用单线程处理所有垃圾回收工作，因为无需多线程交互，实现容易，而且效率比较高。但是，其局限性也比较明显，即无法使用多处理器的优势，所以此收集适合单处理器机器。当然，此收集器也可以用在小数据量（100M 左右）情况下的多处理器机器上。

并行收集:并行收集使用多线程处理垃圾回收工作，因而速度快，效率高。而且理论上 CPU 数目越多，越能体现出并行收集器的优势。

并发收集:相对于串行收集和并行收集而言，前面两个在进行垃圾回收工作时，需要暂停整个运行环境，而只有垃圾回收程序在运行，因此，系统在垃圾回收时会有明显的暂停，而且暂停时间会因为堆越大而越长。

3. （四）-垃圾回收面临的问题

上面说到的“引用计数”法，通过统计控制生成对象和删除对象时的引用数来判断。垃圾回收程序收集计数为 0 的对象即可。但是这种方法无法解决循环引用。所以，后来实现的垃圾判断算法中，都是从程序运行的根节点出发，遍历整个对象引用，查找存活的对象。那么在这种方式的实现中，**垃圾回收从哪儿开始的呢**？即，从哪儿开始查找哪些对象是正在被当前系统使用的。上面分析的堆和栈的区别，其中栈是真正进行程序执行地方，所以要获取哪些对象正在被使用，则需要从 Java 栈开始。同时，一个栈是与一个线程对应的，因此，如果有多个线程的话，则必须对这些线程对应的所有的栈进行检查。



同时，除了栈外，还有系统运行时的寄存器等，也是存储程序运行数据的。这样，以栈或寄存器中的引用为起点，我们可以找到堆中的对象，又从这些对象找到对堆中其他对象的引用，这种引用逐步扩展，最终以 null 引用或者基本类型结束，这样就形成了一颗以 Java

栈中引用所对应的对象为根节点的一颗对象树，如果栈中有多个引用，则最终会形成多颗对象树。在这些对象树上的对象，都是当前系统运行所需要的对象，不能被垃圾回收。而其他剩余对象，则可以视为无法被引用到的对象，可以被当做垃圾进行回收。

因此，**垃圾回收的起点是一些根对象（java 栈，静态变量，寄存器...）**。而最简单的 Java 栈就是 Java 程序执行的 main 函数。这种回收方式，也是上面提到的“标记-清除”的回收方式

如何处理碎片

由于不同 Java 对象存活时间是不一定的，因此，在程序运行一段时间以后，如果不进行内存整理，就会出现零散的内存碎片。碎片最直接的问题就是会导致无法分配大块的内存空间，以及程序运行效率降低。所以，在上面提到的基本垃圾回收算法中，“复制”方式和“标记-整理”方式，都可以解决碎片的问题。

如何解决同时存在的对象创建和对象回收问题

垃圾回收线程是回收内存的，而程序运行线程则是消耗（或分配）内存的，**一个回收内存，一个分配内存**，从这点看，两者是矛盾的。因此，在现有的垃圾回收方式中，要进行垃圾回收前，一般都需要暂停整个应用（即：暂停内存的分配），然后进行垃圾回收，回收完成后再继续应用。这种实现方式是最直接，而且最有效的解决二者矛盾的方式。

但是**这种方式有一个很明显的弊端，就是当堆空间持续增大时，垃圾回收的时间也将会相应的持续增大，对应应用暂停的时间也会相应的增大。**一些对相应时间要求很高的应用，比如

最大暂停时间要求是几百毫秒，那么当堆空间大于几个 G 时，就很有可能超过这个限制，在这种情况下，垃圾回收将会成为系统运行的一个瓶颈。为解决这种矛盾，有了**并发垃圾回收算法**，使用这种算法，垃圾回收线程与程序运行线程同时运行。在这种方式下，解决了暂停的问题，但是因为需要在新生成对象的同时又要回收对象，算法复杂性会大大增加，系统的处理能力也会相应降低，同时，“碎片”问题将会比较难解决。

4. （五）-分代垃圾回收详述 1

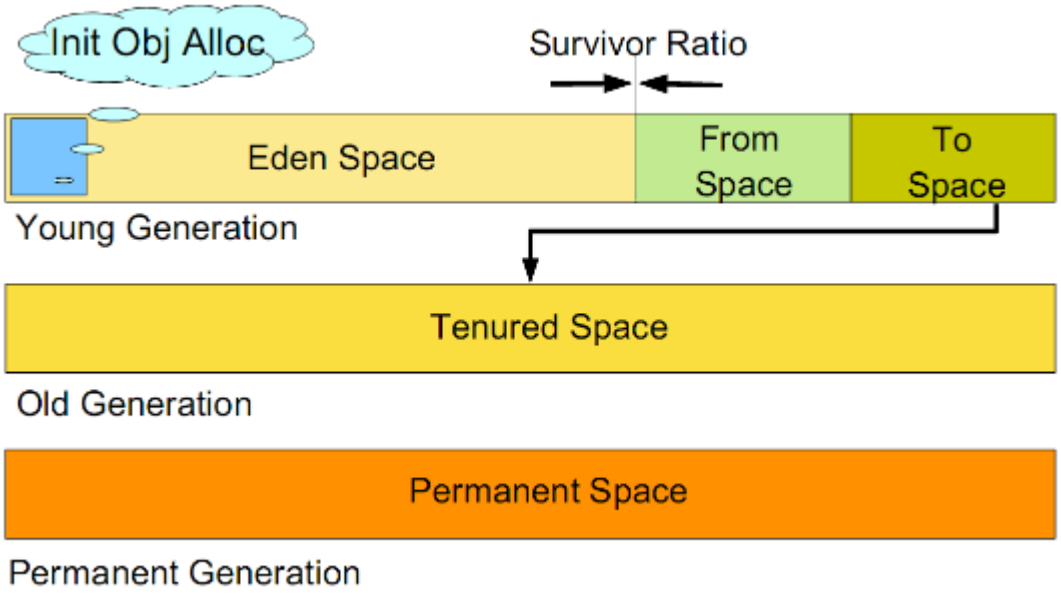
为什么要分代

分代的垃圾回收策略，是基于这样一个事实：**不同的对象的生命周期是不一样的。**因此，不同生命周期的对象可以采取不同的收集方式，以便提高回收效率。

在 Java 程序运行的过程中，会产生大量的对象，其中有些对象是与业务信息相关，比如 Http 请求中的 Session 对象、线程、Socket 连接，这类对象跟业务直接挂钩，因此生命周期比较长。但是还有一些对象，主要是程序运行过程中生成的临时变量，这些对象生命周期会比较短，比如：String 对象，由于其不变类的特性，系统会产生大量的这些对象，有些对象甚至只用一次即可回收。

试想，在不进行对象存活时间区分的情况下，每次垃圾回收都是对整个堆空间进行回收，花费时间相对会长，同时，因为每次回收都需要遍历所有存活对象，但实际上，对于生命周期长的对象而言，这种遍历是没有效果的，因为可能进行了很多次遍历，但是他们依旧存在。因此，分代垃圾回收采用分治的思想，进行代的划分，把不同生命周期的对象放在不同代上，不同代上采用最适合它的垃圾回收方式进行回收。

如何分代



如图所示：

虚拟机中的共划分为三个代：**年轻代（Young Generation）、年老点（Old Generation）和持久代（Permanent Generation）**。其中持久代主要存放的是 Java 类的类信息，与垃圾收集要收集的 Java 对象关系不大。年轻代和年老代的划分是对垃圾收集影响比较大的。

年轻代:

所有新生成的对象首先都是放在年轻代的。年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象。年轻代分三个区。一个 Eden 区，两个 Survivor 区(一般而言)。大部分对象在 Eden 区中生成。当 Eden 区满时，还存活的对象将被复制到 Survivor 区（两个中的一个），当这个 Survivor 区满时，此区的存活对象将被复制到另外一个 Survivor 区，当这个 Survivor 去也满了的时候，从第一个 Survivor 区复制过来的并且此时还存活的对象，将被复制“年老区(Tenured)”。需要注意，Survivor 的两个区是对称的，没先后关系，所以同一个区中可能同时存在从 Eden 复制过来 对象，和从前一个 Survivor 复制过来的对象，而复制到年老区的只有从第一个 Survivor 去过来的对象。而且，Survivor 区总有一个是空的。同时，根据程序需要，Survivor 区是可以配置为多个的（多于两个），这样可以增加对象在年轻代中的存在时间，减少被放到年老代的可能。

年老代:

在年轻代中经历了 N 次垃圾回收后仍然存活的对象，就会被放到年老代中。因此，可以认为年老代中存放的都是一些生命周期较长的对象。

持久代:

用于存放静态文件，如今 Java 类、方法等。持久代对垃圾回收没有显著影响，但是有些应用可能动态生成或者调用一些 class，例如 Hibernate 等，在这种时候需要设置一个比较大的持久代空间来存放这些运行过程中新增的类。持久代大小通过

-XX:MaxPermSize=<N>进行设置。

什么情况下触发垃圾回收

由于对象进行了分代处理，因此垃圾回收区域、时间也不一样。GC 有两种类型：**Scavenge GC** 和 **Full GC**。

Scavenge GC

一般情况下，当新对象生成，并且在 Eden 申请空间失败时，就会触发 Scavenge GC，对 Eden 区域进行 GC，清除非存活对象，并且把尚且存活的对象移动到 Survivor 区。然后整理 Survivor 的两个区。这种方式的 GC 是对年轻代的 Eden 区进行，不会影响到年老代。因为大部分对象都是从 Eden 区开始的，同时 Eden 区不会分配的很大，所以 Eden 区的 GC 会频繁进行。因而，一般在这里需要使用速度快、效率高的算法，使 Eden 去能尽快空闲出来。

Full GC

对整个堆进行整理，包括 Young、Tenured 和 Perm。Full GC 因为需要对整个对进行回收，所以比 Scavenge GC 要慢，因此应该尽可能减少 Full GC 的次数。在对 JVM 调优的过程中，很大一部分工作就是对于 FullGC 的调节。有如下原因可能导致 Full GC：

· 年老代（Tenured）被写满

· 持久代（Perm）被写满

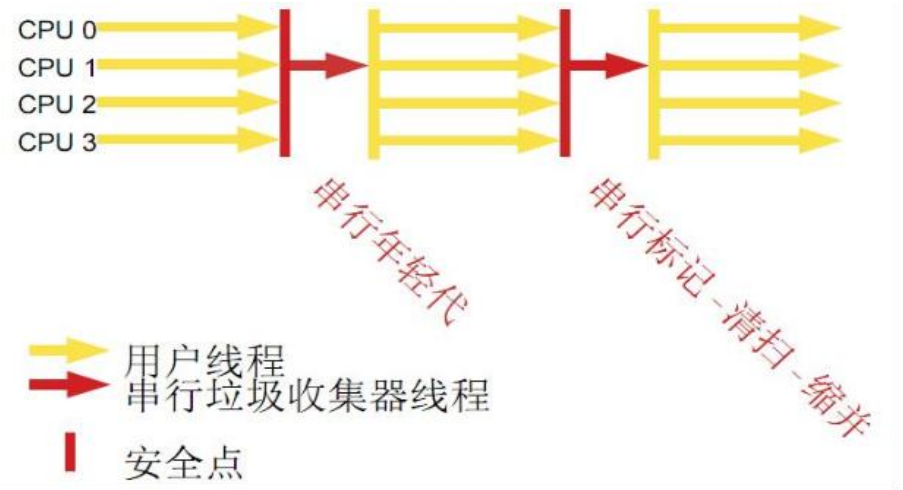
· System.gc()被显示调用

·上一次 GC 之后 Heap 的各域分配策略动态变化

5. （六）-分代垃圾回收详述 2

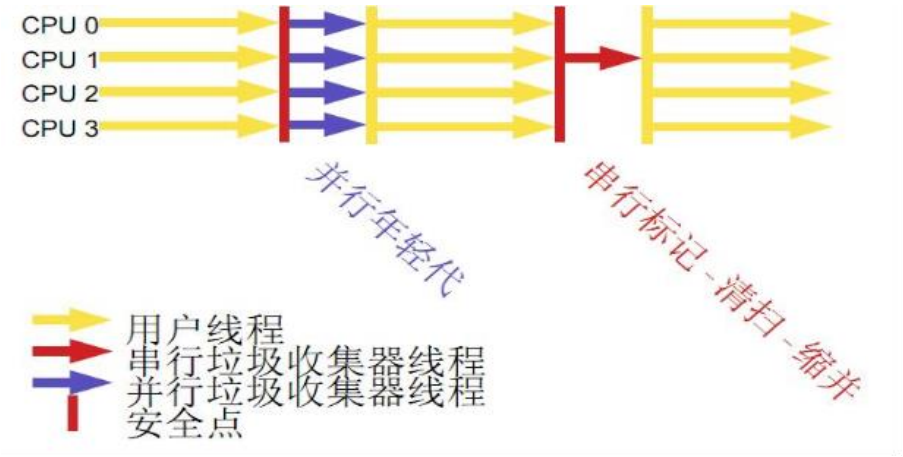
选择合适的垃圾收集算法

串行收集器



用单线程处理所有垃圾回收工作，因为无需多线程交互，所以效率比较高。但是，也无法使用多处理器的优势，所以此收集器适合单处理器机器。当然，此收集器也可以用在小数据量（100M 左右）情况下的多处理器机器上。可以使用-XX:+UseSerialGC 打开。

并行收集器



对年轻代进行并行垃圾回收，因此可以减少垃圾回收时间。一般在多线程多处理器机器上使用。使用-XX:+UseParallelGC.打开。并行收集器在 J2SE5.0 第六 6 更新上引入，在 Java SE6.0 中进行了增强--可以对年老代进行并行收集。如果年老代不使用并发收集的话，默认是使用单线程进行垃圾回收，因此会制约扩展能力。使用-XX:+UseParallelOldGC 打开。使用-XX:ParallelGCThreads=<N>设置并行垃圾回收的线程数。此值可以设置与机器处理器数量相等。

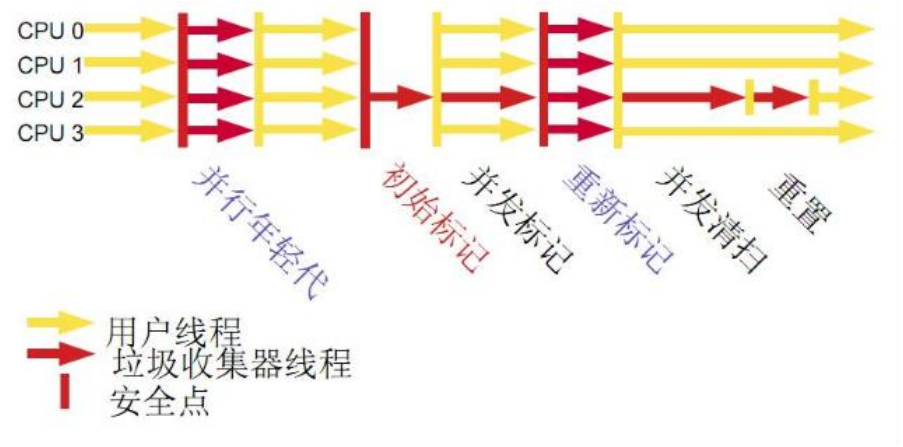
此收集器可以进行如下配置：

最大垃圾回收暂停:指定垃圾回收时的最长暂停时间，通过-XX:MaxGCPauseMillis=<N>指定。<N>为毫秒.如果指定了此值的话，堆大小和垃圾回收相关参数会进行调整以达到指定值。设定此值可能会减少应用的吞吐量。

吞吐量:吞吐量为垃圾回收时间与非垃圾回收时间的比值，通过-XX:GCTimeRatio=<N>来设定，公式为 1/（1+N）。例如，-XX:GCTimeRatio=19 时，表示 5%的时间用于垃圾回收。默认情况为 99，即 1%的时间用于垃圾回收。

并发收集器

可以保证大部分工作都并发进行（应用不停止），垃圾回收只暂停很少的时间，此收集器适合对响应时间要求比较高的中、大规模应用。使用-XX:+UseConcMarkSweepGC 打开。



并发收集器主要减少年老代的暂停时间，他在应用不停止的情况下使用独立的垃圾回收线程，跟踪可达对象。在每个年老代垃圾回收周期中，在收集初期并发收集器 会对整个应用进行简短的暂停，在收集中还会再暂停一次。第二次暂停会比第一次稍长，在此过程中多个线程同时进行垃圾回收工作。

并发收集器使用处理器换来短暂的停顿时间。在一个 N 个处理器的系统上，并发收集部分使用 K/N 个可用处理器进行回收，一般情况下 1<=K<=N/4。

在只有一个处理器的主机上使用并发收集器，设置为 incremental mode 模式也可获得较短的停顿时间。

浮动垃圾：由于在应用运行的同时进行垃圾回收，所以有些垃圾可能在垃圾回收进行完成时产生，这样就造成了“Floating Garbage”，这些垃圾需要在下次垃圾回收周期时才能回收掉。所以，并发收集器一般需要 20%的预留空间用于这些浮动垃圾。

Concurrent Mode Failure：并发收集器在应用运行时进行收集，所以需要保证堆在垃圾回收的这段时间有足够的空间供程序使用，否则，垃圾回收还未完成，堆空间先满了。这种情况下将会发生“并发模式失败”，此时整个应用将会暂停，进行垃圾回收。

启动并发收集器：因为并发收集在应用运行时进行收集，所以必须保证收集完成之前有足够的内存空间供程序使用，否则会出现“Concurrent Mode Failure”。通过设置 -XX:CMSInitiatingOccupancyFraction=<N>指定还有多少剩余堆时开始执行并发收集

小结

串行处理器：

--适用情况：数据量比较小（100M 左右）；单处理器下并且对响应时间无要求的应用。

--缺点：只能用于小型应用

并行处理器：

--适用情况：“对吞吐量有高要求”，多 CPU、对应用响应时间无要求的中、大型应用。举例：后台处理、科学计算。

--缺点：垃圾收集过程中应用响应时间可能加长

并发处理器：

--适用情况：“对响应时间有高要求”，多 CPU、对应用响应时间有较高要求的中、大型应用。举例：Web 服务器/应用服务器、电信交换、集成开发环境。

6. （七）-典型配置举例 1

以下配置主要针对分代垃圾回收算法而言。

堆大小设置

年轻代的设置很关键

JVM 中最大堆大小有三方面限制：相关操作系统的数据模型（32-bt 还是 64-bit）限制；系统的可用虚拟内存限制；系统的可用物理内存限制。32 位系统下，一般限制在 1.5G~2G；64 为操作系统对内存无限制。在 Windows Server 2003 系统，3.5G 物理内存，JDK5.0 下测试，最大可设置为 1478m。

典型设置：

java **-Xmx3550m -Xms3550m -Xmn2g** –Xss128k

-Xmx3550m：设置 JVM 最大可用内存为 3550M。

-Xms3550m：设置 JVM 初始内存为 3550m。此值可以设置与-Xmx 相同，以避免每次垃圾回收完成后 JVM 重新分配内存。

-Xmn2g：设置年轻代大小为 2G。整个堆大小=年轻代大小 + 年老代大小 + 持久代大小。持久代一般固定大小为 64m，所以增大年轻代后，将会减小年老代大小。此值对系统性能影响较大，Sun 官方推荐配置为整个堆的 3/8。

-Xss128k：设置每个线程的堆栈大小。JDK5.0 以后每个线程堆栈大小为 1M，以前每个线程堆栈大小为 256K。更具应用的线程所需内存大小进行调整。在相同物理内存下，减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在 3000~5000 左右。

java -Xmx3550m -Xms3550m -Xss128k **-XX:NewRatio=4 -XX:SurvivorRatio=4 -XX:MaxPermSize=16m -XX:MaxTenuringThreshold=0**

-XX:NewRatio=4:设置年轻代（包括 Eden 和两个 Survivor 区）与年老代的比值（除去持久代）。设置为 4，则年轻代与年老代所占比值为 1：4，年轻代占整个堆栈的 1/5

-XX:SurvivorRatio=4：设置年轻代中 Eden 区与 Survivor 区的大小比值。设置为 4，则两个 Survivor 区与一个 Eden 区的比值为 2:4，一个 Survivor 区占整个年轻代的 1/6

-XX:MaxPermSize=16m:设置持久代大小为 16m。

-XX:MaxTenuringThreshold=0：设置垃圾最大年龄。如果设置为 0 的话，则年轻代对象不经过 Survivor 区，直接进入年老代。对于年老代比较多的应用，可以提高效率。如果将此值设置为一个较大值，则年轻代对象会在 Survivor 区进行多次复制，这样可以增加对象再年轻代的存活时间，增加在年轻代即被回收的概论。

回收器选择

JVM 给了三种选择：**串行收集器、并行收集器、并发收集器**，但是串行收集器只适用于小数据量的情况，所以这里的选择主要针对并行收集器和并发收集器。默认情况下，JDK5.0 以前都是使用串行收集器，如果想使用其他收集器需要在启动时加入相应参数。JDK5.0 以后，JVM 会根据当前系统配置进行判断。

吞吐量优先的并行收集器

如上文所述，并行收集器主要以到达一定的吞吐量为目标，适用于科学技术和后台处理等。

典型配置：

java -Xmx3800m -Xms3800m -Xmn2g -Xss128k **-XX:+UseParallelGC**

-XX:ParallelGCThreads=20

-XX:+UseParallelGC：选择垃圾收集器为并行收集器。**此配置仅对年轻代有效。即上述配置下，年轻代使用并发收集，而年老代仍旧使用串行收集。**

-XX:ParallelGCThreads=20：配置并行收集器的线程数，即：同时多少个线程一起进行垃圾回收。此值最好配置与处理器数目相等。

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC

-XX:ParallelGCThreads=20 **-XX:+UseParallelOldGC**

-XX:+UseParallelOldGC：配置年老代垃圾收集方式为并行收集。JDK6.0 支持对年老代并行收集。

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k

-XX:+UseParallelGC **-XX:MaxGCPauseMillis=100**

-XX:MaxGCPauseMillis=100:设置每次年轻代垃圾回收的最长时间，如果无法满足此时间，JVM 会自动调整年轻代大小，以满足此值。

n java -Xmx3550m -Xms3550m -Xmn2g -Xss128k

-XX:+UseParallelGC -XX:MaxGCPauseMillis=100 **-XX:+UseAdaptiveSizePolicy**

-XX:+UseAdaptiveSizePolicy：设置此选项后，并行收集器会自动选择年轻代区大小和相应的 Survivor 区比例，以达到目标系统规定的最低相应时间或者收集频率等，此值建议使用并行收集器时，一直打开。

响应时间优先的并发收集器

如上文所述，并发收集器主要是保证系统的响应时间，减少垃圾收集时的停顿时间。适用于应用服务器、电信领域等。

典型配置：

```
java -Xmx3550m -Xms3550m -Xmn2g -Xss128k
-XX:ParallelGCThreads=20 -XX:+UseConcMarkSweepGC -XX:+UseParNewGC
```

-XX:+UseConcMarkSweepGC：设置年老代为并发收集。测试中配置这个以后，-XX:NewRatio=4 的配置失效了，原因不明。所以，此时年轻代大小最好用-Xmn 设置。

-XX:+UseParNewGC: 设置年轻代为并行收集。可与 CMS 收集同时使用。JDK5.0 以上，JVM 会根据系统配置自行设置，所以无需再设置此值。

```
java -Xmx3550m -Xms3550m -Xmn2g -Xss128k
-XX:+UseConcMarkSweepGC -XX:CMSFullGCsBeforeCompaction=5
-XX:+UseCMSCompactAtFullCollection
```

-XX:CMSFullGCsBeforeCompaction：由于并发收集器不对内存空间进行压缩、整理，所以运行一段时间以后会产生“碎片”，使得运行效率降低。此值设置运行多少次 GC 以后对内存空间进行压缩、整理。

-XX:+UseCMSCompactAtFullCollection：打开对年老代的压缩。可能会影响性能，但是可以消除碎片

辅助信息

JVM 提供了大量命令行参数，打印信息，供调试使用。主要有以下一些：

-XX:+PrintGC：输出形式：[GC 118250K->113543K(130112K), 0.0094143 secs] [Full GC 121376K->10414K(130112K), 0.0650971 secs]

-XX:+PrintGCDetails：输出形式：[GC [DefNew: 8614K->781K(9088K), 0.0123035 secs] 118250K->113543K(130112K), 0.0124633 secs] [GC [DefNew: 8614K->8614K(9088K), 0.0000665 secs][Tenured: 112761K->10414K(121024K), 0.0433488 secs] 121376K->10414K(130112K), 0.0436268 secs]

-XX:+PrintGCTimeStamps -XX:+PrintGC：PrintGCTimeStamps 可与上面两个混合使用

输出形式：11.851: [GC 98328K->93620K(130112K), 0.0082960 secs]

-XX:+PrintGCApplicationConcurrentTime：打印每次垃圾回收前，程序未中断的执行时间。可与上面混合使用。输出形式：Application time: 0.5291524 seconds

-XX:+PrintGCApplicationStoppedTime：打印垃圾回收期间程序暂停的时间。可与上面混合使用。输出形式：Total time for which application threads were stopped: 0.0468229 seconds

-XX:PrintHeapAtGC: 打印 GC 前后的详细堆栈信息。输出形式：

34.702: [GC {Heap before gc invocations=7:

def new generation　total 55296K, used 52568K [0x1ebd0000, 0x227d0000, 0x227d0000)

eden space 49152K,　99% used [0x1ebd0000, 0x21bce430, 0x21bd0000)

from space 6144K,　55% used [0x221d0000, 0x22527e10, 0x227d0000)

to　space 6144K,　0% used [0x21bd0000, 0x21bd0000, 0x221d0000)

tenured generation total 69632K, used 2696K [0x227d0000, 0x26bd0000,

0x26bd0000)

the space 69632K, 3% used [0x227d0000, 0x22a720f8, 0x22a72200, 0x26bd0000)

compacting perm gen total 8192K, used 2898K [0x26bd0000, 0x273d0000,

0x2abd0000)

 the space 8192K, 35% used [0x26bd0000, 0x26ea4ba8, 0x26ea4c00,

0x273d0000)

ro space 8192K, 66% used [0x2abd0000, 0x2b12bcc0, 0x2b12be00, 0x2b3d0000)

rw space 12288K, 46% used [0x2b3d0000, 0x2b972060, 0x2b972200, 0x2bfd0000)

34.735: [DefNew: 52568K->3433K(55296K), 0.0072126 secs]

55264K->6615K(124928K)Heap after gc invocations=8:

def new generation total 55296K, used 3433K [0x1ebd0000, 0x227d0000,

0x227d0000)

eden space 49152K, 0% used [0x1ebd0000, 0x1ebd0000, 0x21bd0000)

 from space 6144K, 55% used [0x21bd0000, 0x21f2a5e8, 0x221d0000)

 to space 6144K, 0% used [0x221d0000, 0x221d0000, 0x227d0000)

tenured generation total 69632K, used 3182K [0x227d0000, 0x26bd0000,

0x26bd0000)

the space 69632K, 4% used [0x227d0000, 0x22aeb958, 0x22aeba00, 0x26bd0000)

compacting perm gen total 8192K, used 2898K [0x26bd0000, 0x273d0000,

0x2abd0000)

the space 8192K, 35% used [0x26bd0000, 0x26ea4ba8, 0x26ea4c00, 0x273d0000)

ro space 8192K, 66% used [0x2abd0000, 0x2b12bcc0, 0x2b12be00, 0x2b3d0000)

rw space 12288K, 46% used [0x2b3d0000, 0x2b972060, 0x2b972200, 0x2bfd0000)

}

, 0.0757599 secs]

-Xloggc:filename:与上面几个配合使用，把相关日志信息记录到文件以便分析。

7. （八）-典型配置举例 2

常见配置汇总

堆设置

-Xms:初始堆大小

-Xmx:最大堆大小

-XX:NewSize=n:设置年轻代大小

-XX:NewRatio=n:设置年轻代和年老代的比值。如:为 3，表示年轻代与年老代比值为 1：3，年轻代占整个年轻代年老代和的 1/4

-XX:SurvivorRatio=n:年轻代中 Eden 区与两个 Survivor 区的比值。注意 Survivor 区有两个。如：3，表示 Eden：Survivor=3：2，一个 Survivor 区占整个年轻代的 1/5

-XX:MaxPermSize=n:设置持久代大小

收集器设置

-XX:+UseSerialGC:设置串行收集器

-XX:+UseParallelGC:设置并行收集器

-XX:+UseParalledlOldGC:设置并行年老代收集器

-XX:+UseConcMarkSweepGC:设置并发收集器

垃圾回收统计信息

-XX:+PrintGC

-XX:+PrintGCDetails

-XX:+PrintGCTimeStamps

-Xloggc:filename

并行收集器设置

-XX:ParallelGCThreads=n:设置并行收集器收集时使用的 CPU 数。并行收集线程数。

-XX:MaxGCPauseMillis=n:设置并行收集最大暂停时间

-XX:GCTimeRatio=n:设置垃圾回收时间占程序运行时间的百分比。公式为 1/(1+n)

并发收集器设置

-XX:+CMSIncrementalMode:设置为增量模式。适用于单 CPU 情况。

-XX:ParallelGCThreads=n:设置并发收集器年轻代收集方式为并行收集时，使用的 CPU 数。并行收集线程数。

调优总结

年轻代大小选择

响应时间优先的应用：尽可能设大，直到接近系统的最低响应时间限制（根据实际情况选择）。

在此种情况下，年轻代收集发生的频率也是最小的。同时，减少到达年老代的对象。

吞吐量优先的应用：尽可能的设置大，可能到达 Gbit 的程度。因为对响应时间没有要求，垃圾收集可以并行进行，一般适合 8CPU 以上的应用。

年老代大小选择

响应时间优先的应用：年老代使用并发收集器，所以其大小需要小心设置，一般要考虑**并发会话率**和**会话持续时间**等一些参数。如果堆设置小了，可以会造成内存碎片、高回收频率以及应用暂停而使用传统的标记清除方式；如果堆大了，则需要较长的收集时间。最优化的方案，一般需要参考以下数据获得：

1. 并发垃圾收集信息
2. 持久代并发收集次数
3. 传统 GC 信息
4. 花在年轻代和年老代回收上的时间比例

减少年轻代和年老代花费的时间，一般会提高应用的效率

吞吐量优先的应用

一般吞吐量优先的应用都有一个很大的年轻代和一个较小的年老代。原因是，这样可以尽可能回收掉大部分短期对象，减少中期的对象，而年老代尽存放长期存活对象。

较小堆引起的碎片问题

因为年老代的并发收集器使用标记、清除算法，所以不会对堆进行压缩。当收集器回收时，他会把相邻的空间进行合并，这样可以分配给较大的对象。但是，当堆空间较小时，运行一段时间以后，就会出现“碎片”，如果并发收集器找不到足够的空间，那么并发收集器将会停止，然后使用传统的标记、清除方式进行回收。如果出现“碎片”，可能需要进行如下配置：

1. **-XX:+UseCMSCompactAtFullCollection**：使用并发收集器时，开启对年老代的压缩。

1. **-XX:CMSFullGCsBeforeCompaction=0**：上面配置开启的情况下，这里设置多少次 Full GC 后，对年老代进行压缩

8. （九）-新一代的垃圾回收算法

垃圾回收的瓶颈

传统分代垃圾回收方式，已经在一定程度上把垃圾回收给应用带来的负担降到了最小，把应用的吞吐量推到了一个极限。但是他无法解决的一个问题，就是 Full GC 所带来的应用

暂停。在一些对实时性要求很高的应用场景下，GC 暂停所带来的请求堆积和请求失败是无法接受的。这类应用可能要求请求的返回时间在几百甚至几十毫秒以内，如果分代垃圾回收方式要达到这个指标，只能把最大堆的设置限制在一个相对较小范围内，但是这样有限制了应用本身的处理能力，同样也是不可接收的。

分代垃圾回收方式确实也考虑了实时性要求而提供了并发回收器，支持最大暂停时间的设置，但是受限于分代垃圾回收的内存划分模型，其效果也不是很理想。

为了达到实时性的要求（其实 Java 语言最初的设计也是在嵌入式系统上的），一种新垃圾回收方式呼之欲出，它既支持短的暂停时间，又支持大的内存空间分配。可以很好的解决传统分代方式带来的问题。

增量收集的演进

增量收集的方式在理论上可以解决传统分代方式带来的问题。增量收集把对堆空间划分成一系列内存块，使用时，先使用其中一部分（不会全部用完），垃圾收集时把之前用掉的部分中的存活对象再放到后面没有用的空间中，这样可以实现一直边使用边收集的效果，避免了传统分代方式整个使用完了再暂停的回收的情况。

当然，传统分代收集方式也提供了并发收集，但是他有一个很致命的地方，就是把整个堆做为一个内存块，这样一方面会造成碎片（无法压缩），另一方面他的每次收集都是对整个堆的收集，无法进行选择，在暂停时间的控制上还是很弱。而增量方式，通过内存空间的分块，恰恰可以解决上面问题。

Garbage Firest（G1）

这部分的内容主要参考这里，这篇文章算是对 G1 算法论文的解读。我也没加什么东西了。

目标

从设计目标看 G1 完全是为了大型应用而准备的。

支持很大的堆

高吞吐量

--支持多 CPU 和垃圾回收线程

--在主线程暂停的情况下，使用并行收集

--在主线程运行的情况下，使用并发收集

实时目标：可配置在 N 毫秒内最多只占用 M 毫秒的时间进行垃圾回收

当然 G1 要达到实时性的要求，相对传统的分代回收算法，在性能上会有一些损失。

算法详解

G1 可谓博采众家之长，力求到达一种完美。他吸取了增量收集优点，把整个堆划分为一个一个等大小的区域（region）。内存的回收和划分都以 region 为单位；同时，他也吸取

了 CMS 的特点，把这个垃圾回收过程分为几个阶段，分散一个垃圾回收过程；而且，G1 也认同分代垃圾回收的思想，认为不同对象的生命周期不同，可以采取不同收集方式，因此，它也支持分代的垃圾回收。为了达到对回收时间的可预计性，G1 在扫描了 region 以后，对其中的活跃对象的大小进行排序，首先会收集那些活跃对象小的 region，以便快速回收空间（要复制的活跃对象少了），因为活跃对象小，里面可以认为多数都是垃圾，所以这种方式被称为 Garbage First（G1）的垃圾回收算法，即：垃圾优先的回收。

回收步骤：

初始标记（Initial Marking）

G1 对于每个 region 都保存了两个标识用的 bitmap，一个为 previous marking bitmap，一个为 next marking bitmap，bitmap 中包含了一个 bit 的地址信息来指向对象的起始点。

开始 Initial Marking 之前，首先并发的清空 next marking bitmap，然后停止所有应用线程，并扫描标识出每个 region 中 root 可直接访问到的对象，将 region 中 top 的值放入 next top at mark start（TAMS）中，之后恢复所有应用线程。

触发这个步骤执行的条件为：

G1 定义了一个 JVM Heap 大小的百分比的阀值，称为 h，另外还有一个 H，H 的值为 (1-h)*Heap Size，目前这个 h 的值是固定的，后续 G1 也许会将其改为动态的，根据 jvm 的运行情况来动态的调整，在分代方式下，G1 还定义了一个 u 以及 soft limit，soft limit

的值为 H-u*Heap Size，当 Heap 中使用的内存超过了 soft limit 值时，就会在一次 clean up 执行完毕后在应用允许的 GC 暂停时间范围内尽快的执行此步骤；

在 pure 方式下，G1 将 marking 与 clean up 组成一个环，以便 clean up 能充分的使用 marking 的信息，当 clean up 开始回收时，首先回收能够带来最多内存空间的 regions，当经过多次的 clean up，回收到没多少空间的 regions 时，G1 重新初始化一个新的 marking 与 clean up 构成的环。

并发标记（Concurrent Marking）

按照之前 Initial Marking 扫描到的对象进行遍历，以识别这些对象的下层对象的活跃状态，对于在此期间应用线程并发修改的对象的以来关系则记录到 remembered set logs 中，新创建的对象则放入比 top 值更高的地址区间中，这些新创建的对象默认状态即为活跃的，同时修改 top 值。

最终标记暂停（Final Marking Pause）

当应用线程的 remembered set logs 未满时，是不会放入 filled RS buffers 中的，在这样的情况下，这些 remebered set logs 中记录的 card 的修改就会被更新了，因此需要这一步，这一步要做的就是把应用线程中存在的 remembered set logs 的内容进行处理，并相应的修改 remembered sets，这一步需要暂停应用，并行的运行。

存活对象计算及清除（Live Data Counting and Cleanup）

值得注意的是，在 G1 中，并不是说 Final Marking Pause 执行完了，就肯定执行 Cleanup 这步的，由于这步需要暂停应用，G1 为了能够达到准实时的要求，需要根据用户指定的最大的 GC 造成的暂停时间来合理的规划什么时候执行 Cleanup，另外还有几种情况也是会触发这个步骤的执行的：

G1 采用的是复制方法来进行收集，必须保证每次的”to space”的空间都是够的，因此 G1 采取的策略是当已经使用的内存空间达到了 H 时，就执行 Cleanup 这个步骤；

对于 full-young 和 partially-young 的分代模式的 G1 而言，则还有情况会触发 Cleanup 的执行，full-young 模式下，G1 根据应用可接受的暂停时间、回收 young regions 需要消耗的时间来估算出一个 yound regions 的数量值，当 JVM 中分配对象的 young regions 的数量达到此值时，Cleanup 就会执行；partially-young 模式下，则会尽量频繁的在应用可接受的暂停时间范围内执行 Cleanup，并最大限度的去执行 non-young regions 的 Cleanup。

展望

以后 JVM 的调优或许跟多需要针对 G1 算法进行调优了。

9. （十）-调优方法

Jconsole，jProfile，VisualVM

Jconsole : jdk 自带，功能简单，但是可以在系统有一定负荷的情况下使用。对垃圾回收算法有很详细的跟踪。详细说明参考这里

JProfiler：商业软件，需要付费。功能强大。详细说明参考这里

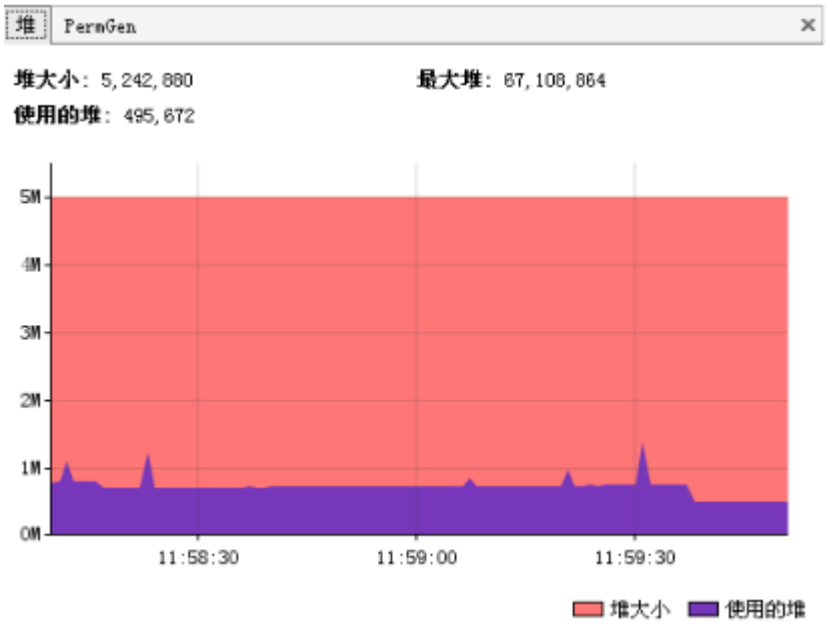
VisualVM：JDK 自带，功能强大，与 JProfiler 类似。推荐。

如何调优

观察内存释放情况、集合类检查、对象树

上面这些调优工具都提供了强大的功能，但是总的来说一般分为以下几类功能

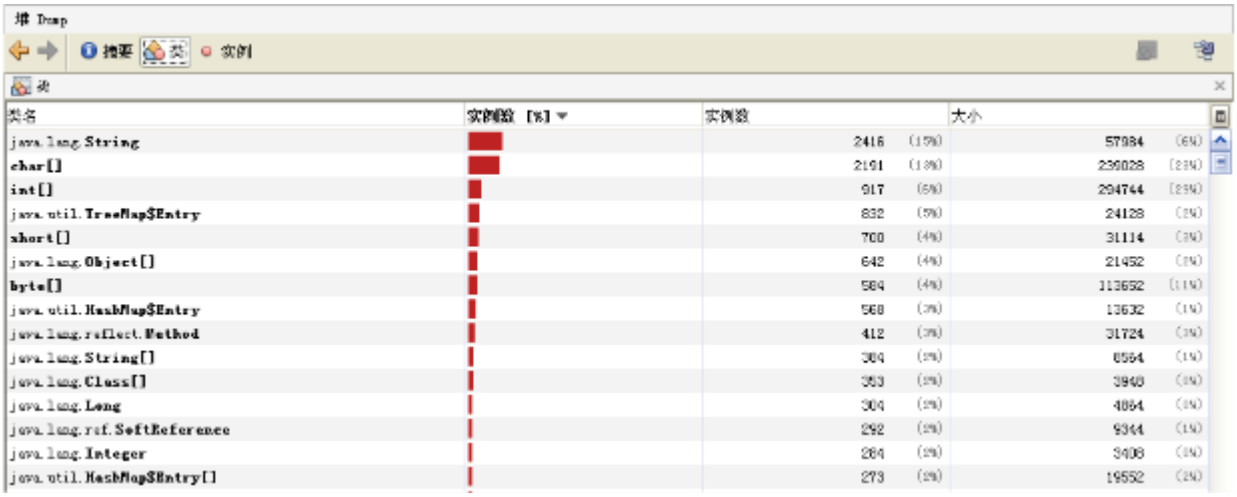
堆信息查看



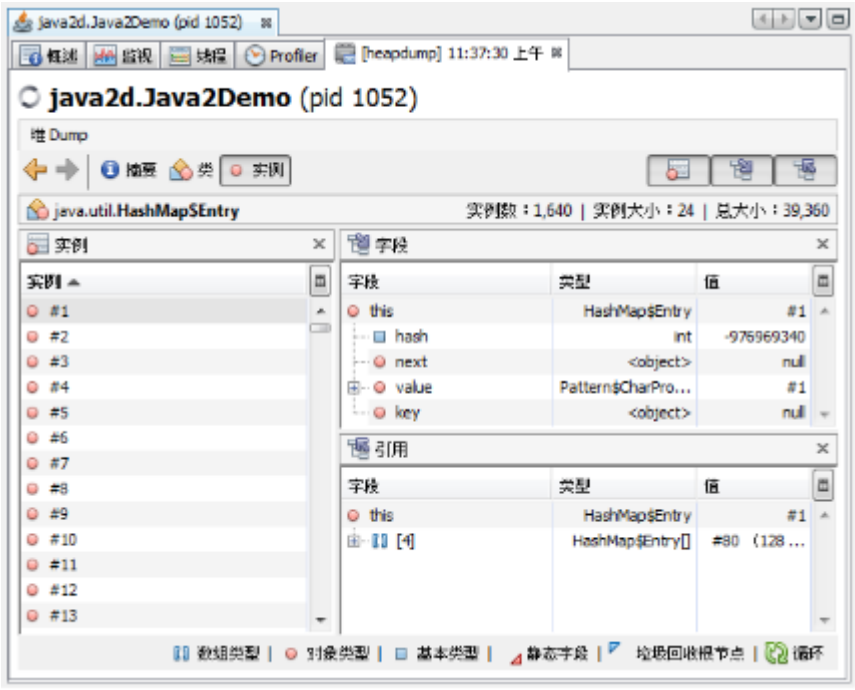
可查看堆空间大小分配（年轻代、年老代、持久代分配）

提供即时的垃圾回收功能

垃圾监控（长时间监控回收情况）



查看堆内类、对象信息查看：数量、类型等



对象引用情况查看

有了堆信息查看方面的功能，我们一般可以顺利解决以下问题：

--年老代年轻代大小划分是否合理

--内存泄漏

--垃圾回收算法设置是否合理

线程监控

线程信息监控：系统线程数量。

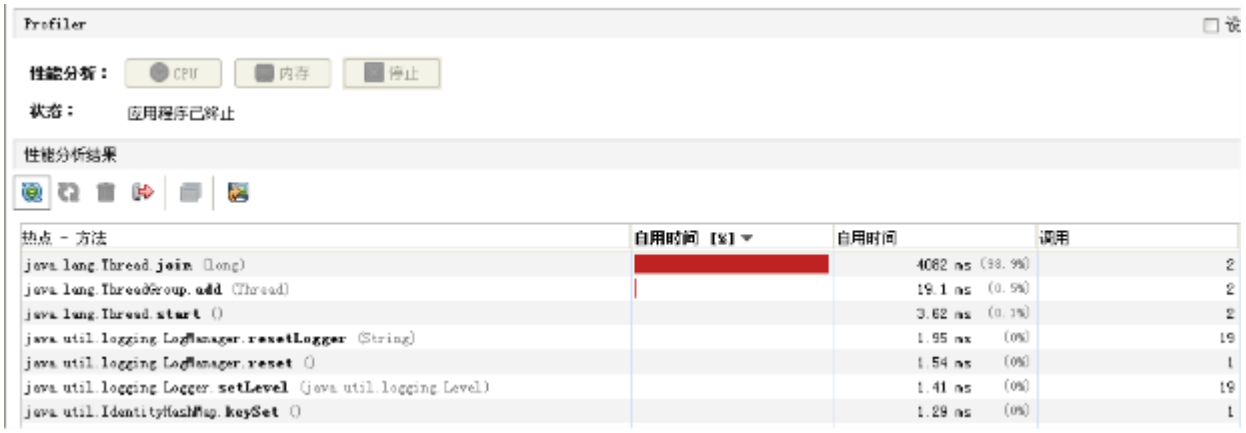
线程状态监控：各个线程都处在什么样的状态下

```
"Finalizer" daemon prio=8 tid=0x02ad0400 nid=0xd18 in Object.wait() [0x02c9f000..0x02c9fc94]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0x22ed2b78> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(Unknown Source)
        - locked <0x22ed2b78> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(Unknown Source)
        at java.lang.ref.Finalizer$FinalizerThread.run(Unknown Source)

   Locked ownable synchronizers:
        - None

"Reference Handler" daemon prio=10 tid=0x02acf000 nid=0xb08 in Object.wait() [0x02c4f000..0x02c4fd14]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0x22ed2c00> (a java.lang.ref.Reference$Lock)
        at java.lang.Object.wait(Object.java:485)
        at java.lang.ref.Reference$ReferenceHandler.run(Unknown Source)
        - locked <0x22ed2c00> (a java.lang.ref.Reference$Lock)

   Locked ownable synchronizers:
        - None

"main" prio=6 tid=0x00306800 nid=0xc00 runnable [0x0093f000..0x0093fa54]
   java.lang.Thread.State: RUNNABLE
        at CPUUserTest.longUseCpu(CPUUserTest.java:9)
        at CPUUserTest.main(CPUUserTest.java:31)

   Locked ownable synchronizers:
        - None
```

Dump 线程详细信息：查看线程内部运行情况

死锁检查

热点分析



CPU 热点：检查系统哪些方法占用的大量 CPU 时间

内存热点：检查哪些对象在系统中数量最大（一定时间内存活对象和销毁对象一起统计）

这两个东西对于系统优化很有帮助。我们可以根据找到的热点，有针对性的进行系统的瓶颈查找和进行系统优化，而不是漫无目的的进行所有代码的优化。

快照

快照是系统运行到某一时刻的一个定格。在我们进行调优的时候，不可能用眼睛去跟踪所有系统变化，依赖快照功能，我们就可以进行系统两个不同运行时刻，对象（或类、线程等）的不同，以便快速找到问题

举例说，我要检查系统进行垃圾回收以后，是否还有该收回的对象被遗漏下来的了。那么，我可以在进行垃圾回收前后，分别进行一次堆情况的快照，然后对比两次快照的对象情况。

内存泄漏检查

内存泄漏是比较常见的问题，而且解决方法也比较通用，这里可以重点说一下，而线程、热点方面的问题则是具体问题具体分析了。

内存泄漏一般可以理解为系统资源（各方面的资源，堆、栈、线程等）在错误使用的情况下，导致使用完毕的资源无法回收（或没有回收），从而导致新的资源分配请求无法完成，引起系统错误。

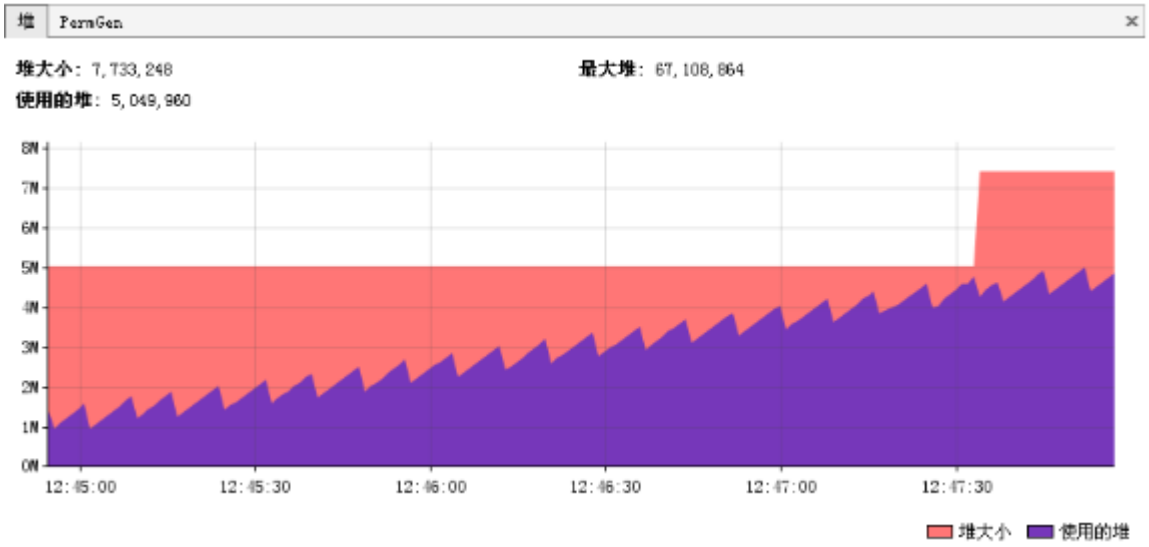
内存泄漏对系统危害比较大，因为他可以直接导致系统的崩溃。

需要区别一下，内存泄漏和系统超负荷两者是有区别的，虽然可能导致的最终结果是一样的。内存泄漏是用完的资源没有回收引起错误，而系统超负荷则是系统确实没有那么多资源可以分配了（其他的资源都在使用）。

年老代堆空间被占满

异常： java.lang.OutOfMemoryError: Java heap space

说明：



这是最典型的内存泄漏方式，简单说就是所有堆空间都被无法回收的垃圾对象占满，虚拟机无法再在分配新空间。

如上图所示，这是非常典型的内存泄漏的垃圾回收情况图。所有峰值部分都是一次垃圾回收点，所有谷底部分表示是一次垃圾回收后剩余的内存。连接所有谷底的点，可以发现一条由底到高的线，这说明，随时间的推移，系统的堆空间被不断占满，最终会占满整个堆空

间。因此可以初步认为系统内部可能有内存泄漏。（上面的图仅供示例，在实际情况下收集数据的时间需要更长，比如几个小时或者几天）

解决：

这种方式解决起来也比较容易，一般就是根据垃圾回收前后情况对比，同时根据对象引用情况（常见的集合对象引用）分析，基本都可以找到泄漏点。

持久代被占满

异常：java.lang.OutOfMemoryError: PermGen space

说明：

Perm 空间被占满。无法为新的 class 分配存储空间而引发的异常。这个异常以前是没有的，但是在 Java 反射大量使用的今天这个异常比较常见了。主要原因就是大量动态反射生成的类不断被加载，最终导致 Perm 区被占满。

更可怕的是，不同的 classLoader 即便使用了相同的类，但是都会对其进行加载，相当于同一个东西，如果有 N 个 classLoader 那么他将会被加载 N 次。因此，某些情况下，这个问题基本视为无解。当然，存在大量 classLoader 和大量反射类的情况其实也不多。

解决：

1. -XX:MaxPermSize=16m
2. 换用 JDK。比如 JRocket。

堆栈溢出

异常：java.lang.StackOverflowError

说明：这个就不多说了，一般就是递归没返回，或者循环调用造成

线程堆栈满

异常：Fatal: Stack size too small

说明：java 中一个线程的空间大小是有限制的。JDK5.0 以后这个值是 1M。与这个线程相关的数据将会保存在其中。但是当线程空间满了以后，将会出现上面异常。

解决：增加线程栈大小。-Xss2m。但这个配置无法解决根本问题，还要看代码部分是否有造成泄漏的部分。

系统内存被占满

异常：java.lang.OutOfMemoryError: unable to create new native thread

说明：

这个异常是由于操作系统没有足够的资源来产生这个线程造成的。系统创建线程时，除了要在 Java 堆中分配内存外，操作系统本身也需要分配资源来创建线程。因此，当线程数量大到一定程度以后，堆中或许还有空间，但是操作系统分配不出资源来了，就出现这个异常了。

分配给 Java 虚拟机的内存愈多，系统剩余的资源就越少，因此，当系统内存固定时，分配给 Java 虚拟机的内存越多，那么，系统总共能够产生的线程也就越少，两者成反比的关系。

同时，可以通过修改-Xss 来减少分配给单个线程的空间，也可以减少系统总共生产的线程数。

解决：

1. 重新设计系统减少线程数量。
2. 线程数量不能减少的情况下，通过-Xss 减小单个线程大小。以便能生产更多的线程。

10. （十一）-反思

垃圾回收的悖论

所谓“成也萧何败萧何”。Java 的垃圾回收确实带来了很多好处，为开发带来了便利。但是在一些高性能、高并发的情况下，垃圾回收确成为了制约 Java 应用的瓶颈。目前 JDK 的垃圾回收算法，始终无法解决垃圾回收时的暂停问题，因为这个暂停严重影响了程序的相应时间，造成拥塞或堆积。这也是后续 JDK 增加 G1 算法的一个重要原因。

当然，上面是从技术角度出发解决垃圾回收带来的问题，但是从系统设计方面我们就需要问一下了：

> **我们需要分配如此大的内存空间给应用吗?**
>
> **我们是否能够通过有效使用内存而不是通过扩大内存的方式来设计我们的系统呢?**

我们的内存中都放了什么

内存中需要放什么呢？个人认为，**内存中需要放的是你的应用需要在不久的将来再次用到到的东西。**想想看，如果你在将来不用这些东西，何必放内存呢？放文件、数据库不是更好？这些东西一般包括：

1. 系统运行时业务相关的数据。比如 web 应用中的 session、即时消息的 session 等。这些数据一般在一个用户访问周期或者一个使用过程中都需要存在。

2. 缓存。缓存就比较多了，你所要快速访问的都可以放这里面。其实上面的业务数据也可以理解为一种缓存。

3. 线程。

因此，我们是不是可以这么认为，如果我们不把业务数据和缓存放在 JVM 中，或者把他们独立出来，那么 Java 应用使用时所需的内存将会大大减少，同时垃圾回收时间也会相应减少。

我认为这是可能的。

解决之道

数据库、文件系统

把所有数据都放入数据库或者文件系统，这是一种最为简单的方式。在这种方式下，Java 应用的内存基本上等于处理一次峰值并发请求所需的内存。数据的获取都在每次请求时从数据库和文件系统中获取。也可以理解为，一次业务访问以后，所有对象都可以进行回收了。

这是一种内存使用最有效的方式，但是从应用角度来说，这种方式很低效。

内存-硬盘映射

上面的问题是因为我们使用了文件系统带来了低效。但是如果我们不是读写硬盘，而是写内存的话效率将会提高很多。

数据库和文件系统都是实实在在进行了持久化，但是当我们并不需要这样持久化的时候，我们可以做一些变通——把内存当硬盘使。

内存-硬盘映射很好很强大，既用了缓存又对 Java 应用的内存使用又没有影响。Java 应用还是 Java 应用，他只知道读写的还是文件，但是实际上是内存。

这种方式兼得的 Java 应用与缓存两方面的好处。memcached 的广泛使用也正是这一类的代表。

同一机器部署多个 JVM

这也是一种很好的方式，可以分为纵拆和横拆。纵拆可以理解为把 Java 应用划分为不同模块，各个模块使用一个独立的 Java 进程。而横拆则是同样功能的应用部署多个 JVM。

通过部署多个 JVM，可以把每个 JVM 的内存控制一个垃圾回收可以忍受的范围内即可。但是这相当于进行了分布式的处理，其额外带来的复杂性也是需要评估的。另外，也有支持分布式的这种 JVM 可以考虑，不要要钱哦：）

程序控制的对象生命周期

这种方式是理想当中的方式，目前的虚拟机还没有，纯属假设。即：考虑由编程方式配置哪些对象在垃圾收集过程中可以直接跳过，减少垃圾回收线程遍历标记的时间。

这种方式相当于在编程的时候告诉虚拟机某些对象你可以在*时间后在进行收集或者由代码标识可以收集了（类似 C、C++），在这之前你即便去遍历他也是没有效果的，他肯定是还在被引用的。

这种方式如果 JVM 可以实现，个人认为将是一个飞跃，Java 即有了垃圾回收的优势，又有了 C、C++对内存的可控性。

线程分配

Java 的阻塞式的线程模型基本上可以抛弃了，目前成熟的 NIO 框架也比较多了。阻塞式 IO 带来的问题是线程数量的线性增长，而 NIO 则可以转换成为常数线程。因此，对于服务端的应用而言，NIO 还是唯一选择。不过，JDK7 中为我们带来的 AIO 是否能让人眼前一亮呢？我们拭目以待。

其他的 JDK

本文说的都是 Sun 的 JDK，目前常见的 JDK 还有 JRocket 和 IBM 的 JDK。其中 JRocket 在 IO 方面比 Sun 的高很多，不过 Sun JDK6.0 以后提高也很大。而且 JRocket 在垃圾回收方面，也具有优势，其可设置垃圾回收的最大暂停时间也是很吸引人的。不过，系统 Sun 的 G1 实现以后，在这方面会有一个质的飞跃。

八、 分布式系统设计系列

【分布式系统中的概念】

三元组

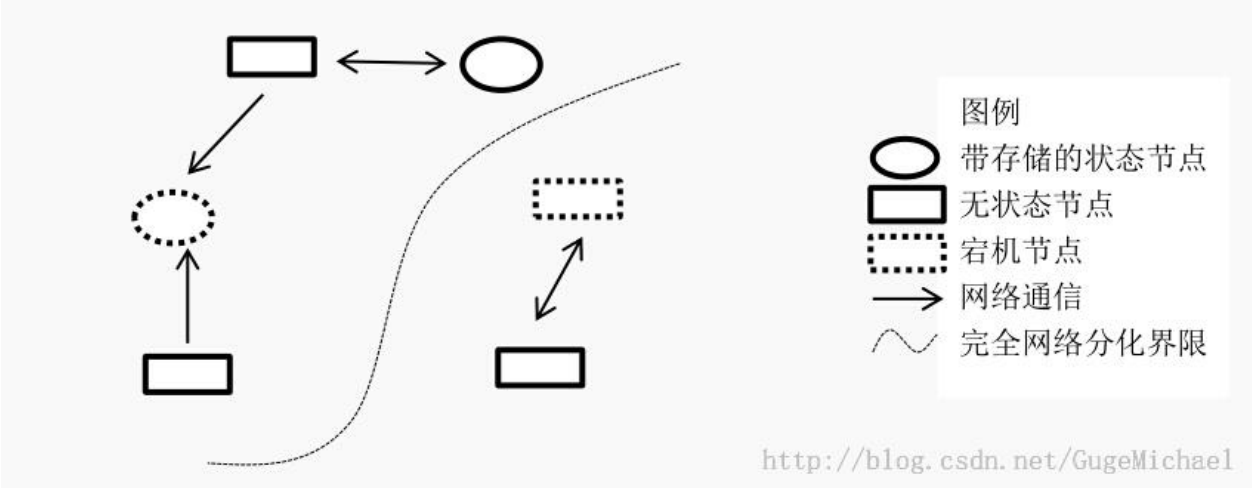
其实，分布式系统说白了，就是很多机器组成的集群，靠彼此之间的网络通信，担当的角色可能不同，共同完成同一个事情的系统。如果按”实体“来划分的话，就是如下这几种：

1、**节点** -- 系统中按照协议完成计算工作的一个逻辑实体，可能是执行某些工作的进程或机器

2、**网络** -- 系统的数据传输通道，用来彼此通信。通信是具有方向性的。

3、**存储** -- 系统中持久化数据的数据库或者文件存储。

如图



状态特性

各个节点的状态可以是“无状态”或者“有状态的”.

一般认为，节点是偏计算和通信的模块，一般是无状态的。这类应用一般不会存储自己的中间状态信息，比如 Nginx，一般情况下是转发请求而已，不会存储中间信息。另一种“有状态”的，如 mysql 等数据库，状态和数据全部持久化到磁盘等介质。

“无状态”的节点一般我们认为是可随意重启的，因为重启后只需要立刻工作就好。“有状态”的则不同，需要先读取持久化的数据，才能开始服务。所以，“无状态”的节点一般是可以随意扩展的，“有状态”的节点需要一些控制协议来保证扩展。

系统异常

异常，可认为是节点因为某种原因不能工作，此为节点异常。还有因为网络原因，临时、永久不能被其他节点所访问，此为网络异常。在分布式系统中，要有对异常的处

理，保证集群的正常工作。

【分布式系统与单节点的不同】

1、从 linux write()系统调用说起

众所周知，在 unix/linux/mac(类 Unix)环境下，两个机器通信，最常用的就是通过 socket 连接对方。传输数据的话，无非就是调用 write()这个系统调用，把一段内存缓冲区发出去。但是可以进一步想一下，write()之后能确认对方收到了这些数据吗?

答案肯定是不能，原因就是发送数据需要走内核->网卡->链路->对端网卡->内核，这一路径太长了，所以只能是异步操作。write()把数据写入内核缓冲区之后就返回到应用层了，具体后面何时发送、怎么发送、TCP 怎么做滑动窗口、流控都是 tcp/ip 协议栈内核的事情了。

所以在应用层，能确认对方受到了消息只能是对方应用返回数据，逻辑确认了这次发送才认为是成功的。这就区别与单系统编程，大部分系统调用、库调用只要返回了就说明已经确认完成了。

2、TCP/IP 协议是“不可靠”的

教科书上明确写明了互联网是不可靠的，TCP 实现了可靠传输。何来“不可靠”呢？先来看一下网络交互的例子，有 A、B 两个节点，之间通过 TCP 连接，现在 A、B 都想确认自己发出的任何一条消息都能被对方接收并反馈，于是开始了如下操作：

A->B 发送数据，然后 A 需要等待 B 收到数据的确认，B 收到数据后发送确认消息给 A，然后 B 需要等待 A 收到数据的确认，A 收到 B 的数据确认消息后再次发送确认消息给 B，然后 A 又去需要等待 B 收到的确认。。。死循环了!!

其实，这就是著名的“拜占庭将军”问题：

http://baike.baidu.com/link?url=6iPrbRxHLOo9an1hT-s6DvM5kAoq7RxclIrzgrS34W1fRq1h507RDWJOxfhkDOcihVFRZ2c7ybCkUosWQeUoS_

所以，通信双方是“不可能”同时确认对方受到了自己的信息。而教科书上定义的其实是指“单向”通信是成立的，比如 A 向 B 发起 Http 调用，收到了 HttpCode 200 的响应包，这只能确认，A 确认 B 收到了自己的请求，并且 B 正常处理了，不能确认的是 B 确认 A 受到了它的成功的消息。

3、不可控的状态

在单系统编程中，我们对系统状态是非常可控的。比如函数调用、逻辑运算，要么成功，要么失败，因为这些操作被框在一个机器内部，cpu/总线/内存都是可以快速得到反馈的。开发者可以针对这两个状态很明确的做出程序上的判断和后续的操作。

而在分布式的网络环境下，这就变得微妙了。比如一次 rpc、http 调用，可能成功、失败，还有可能是“超时”，这就比前者的状态多了一个不可控因素，导致后面的代码不是很容易做出判断。试想一下，用 A 用支付宝向 B 转了一大笔钱，当他按下“确认”后，界面上有个圈在转啊转，然后显示请求超时了，然后 A 就抓狂了，不知道到底钱转没转过去，开始确认自己的账户、确认 B 的账户、打电话找客服等等。

所以分布式环境下，我们的其实要时时刻刻考虑面对这种不可控的“第三状态”设计开发，这也是挑战之一。

4、视”异常“为”正常“

单系统下，进程/机器的异常概率十分小。即使出现了问题，可以通过人工干预重启、迁移等手段恢复。但在分布式环境下，机器上千台，每几分钟都可能出现宕机、死机、网络断网等异常，出现的概率很大。所以，这种环境下，进程 core 掉、机器挂掉都是需要我们在编程中认为随时可能出现的，这样才能使我们整个系统健壮起来，所以”容错“是基本需求。

异常可以分为如下几类：

节点错误：

一般是由于应用导致，一些 coredump 和系统错误触发，一般重新服务后可恢复。

硬件错误：

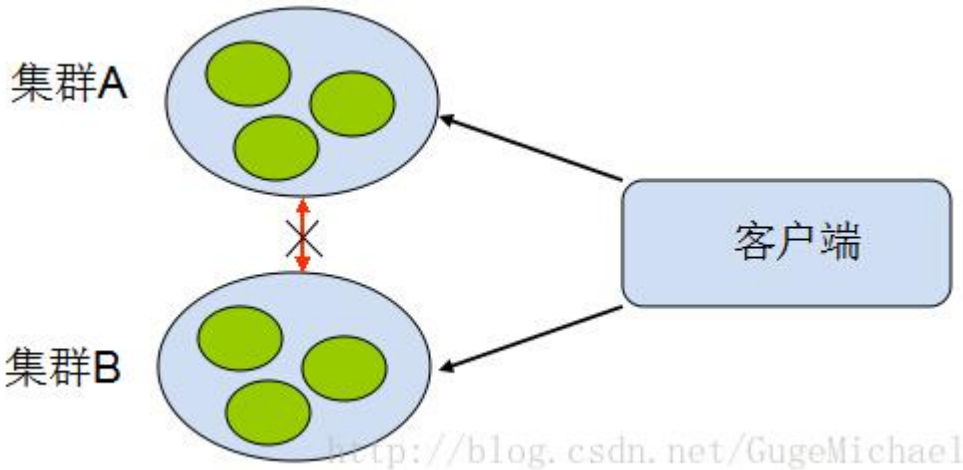
由于磁盘或者内存等硬件设备导致某节点不能服务，需要人工干预恢复。

网络错误：

由于点对点的网络抖动，暂时的访问错误，一般拓扑稳定后或流量减小可以恢复。

网络分化：

网络中路由器、交换机错误导致网络不可达，但是网络两边都正常，这类错误比较难恢复，并且需要在开发时特别处理。**【这种情况也会比较前面的问题较难处理】**



【分布式系统特性】

CAP 是分布式系统里最著名的理论，wiki 百科如下

Consistency(all nodes see the same data at the same time)

Availability (a guarantee that every request receives a response about

whether it was successful or failed)

Partition tolerance (the system continues to operate despite arbitrary message loss or failure of part of the system)

(摘自 ： http://en.wikipedia.org/wiki/CAP_theorem)

早些时候，国外的大牛已经证明了 CAP 三者是不能兼得，很多实践也证明了。

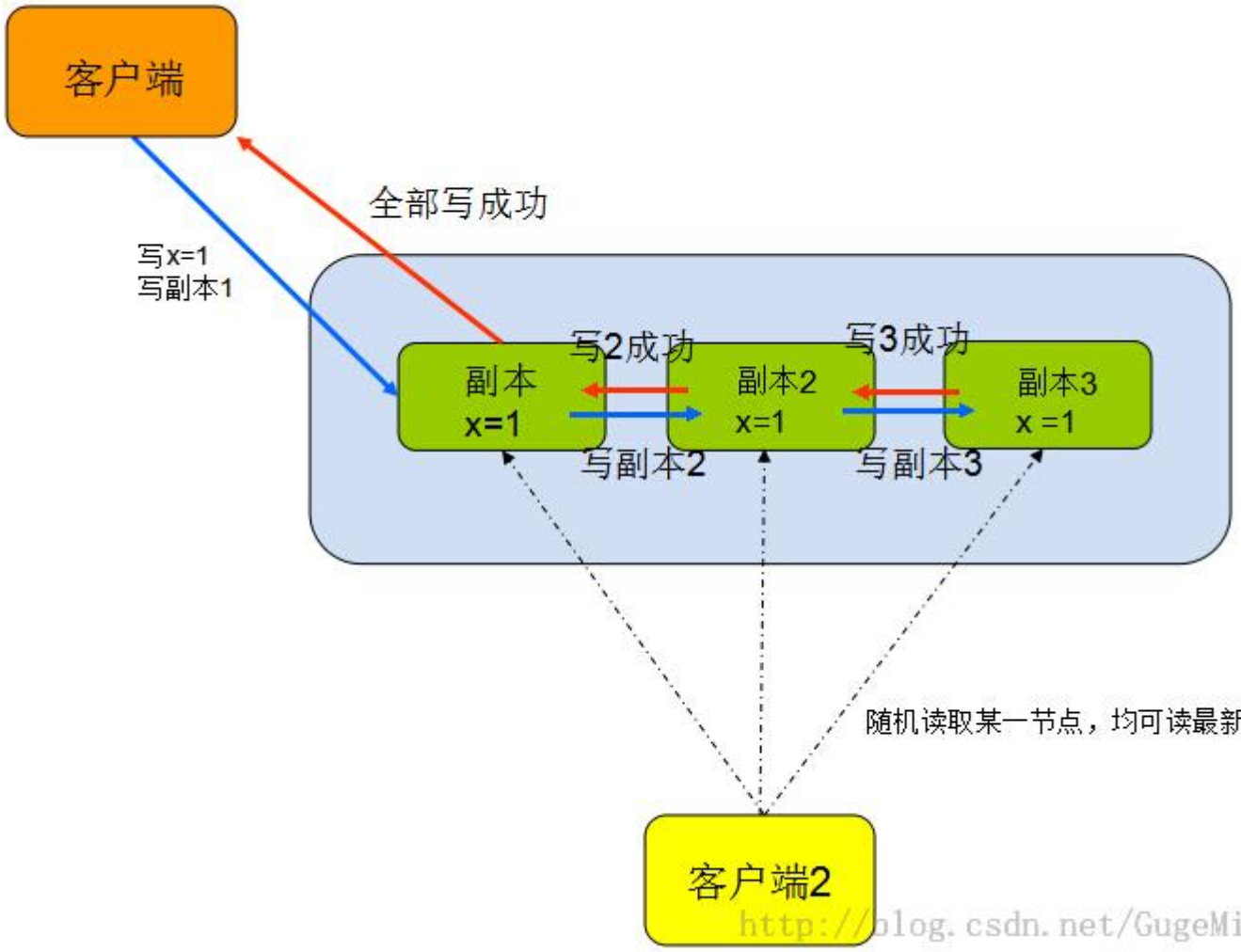
本人就不挑战权威了，感兴趣的同学可以自己 Google。本人以自己的观点总结了一下：

一致性

描述当前所有节点存储数据的统一模型，分为强一致性和弱一致性：

强一致性描述了所有节点的数据高度一致，无论从哪个节点读取，都是一样的。无需担心同一时刻会获得不同的数据。是级别最高的，实现的代价比较高

如图：



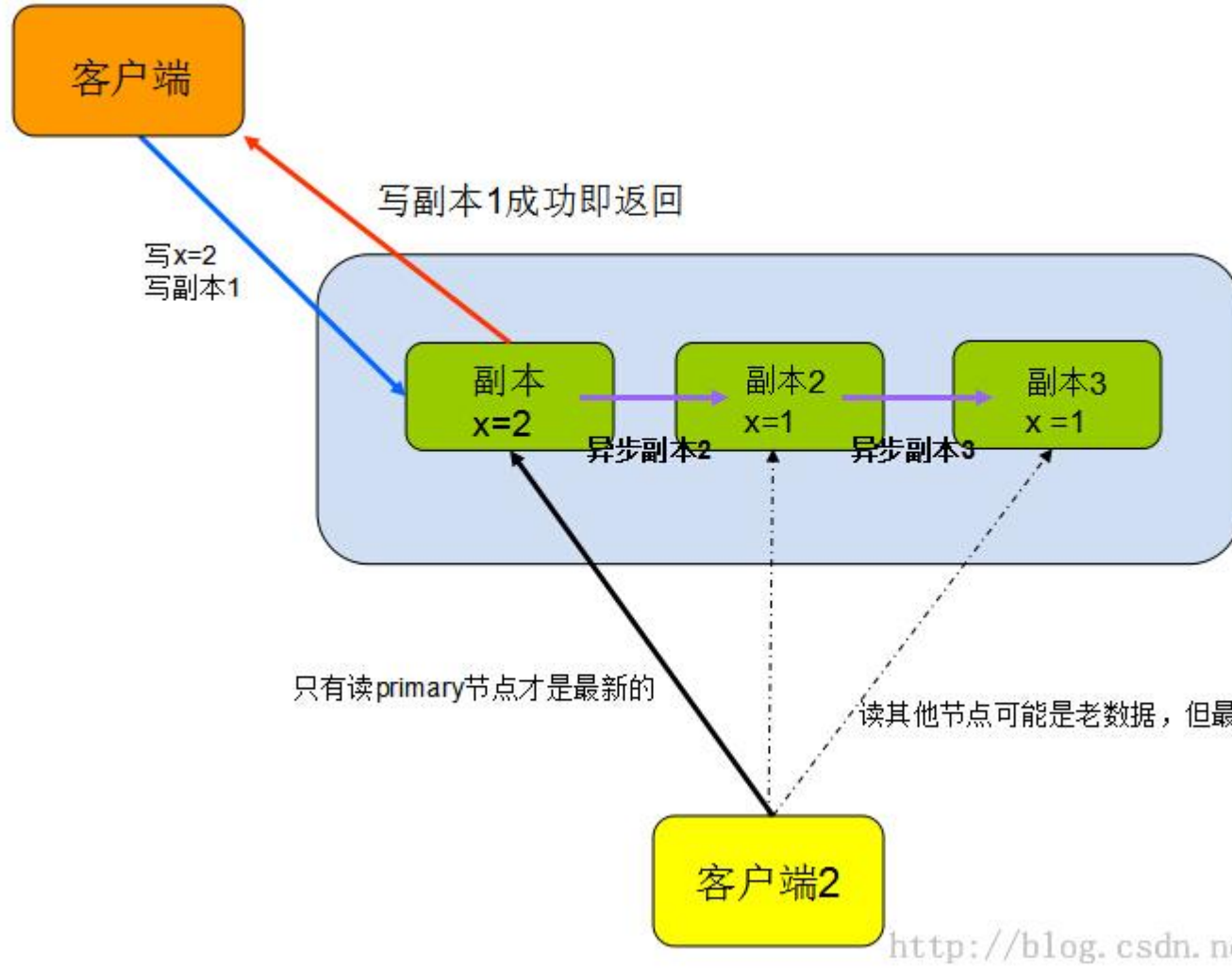
弱一致性又分为单调一致性和最终一致性：

1、单调一致性强调数据是按照时间的新旧，单调向最新的数据靠近，不会回退，如：

数据存在三个版本 v1->v2->v3，获取只能向 v3 靠近(如取到的是 v2，就不可能再次获得 v1)

2、最终一致性强调数据经过一个时间窗口之后，只要多尝试几次，最终的状态是一致的，是最新的数据

如图：



强一致性的场景，就好像交易系统，存取钱的+/-操作必须是马上一致的，否则会令很多人误解。

弱一致性的场景，大部分就像 web 互联网的模式，比如发了一条微博，改了某些配置，可能不会马上生效，但刷新几次后就可以看到了，其实弱一致性就是在系统上通过业务可接受的方式换取了一些系统的低复杂度和可用性。

可用性

保证系统的正常可运行性，在请求方看来，只要发送了一个请求，就可以得到恢复无论成功还是失败（不会超时）！

分区容忍性

在系统某些节点或网络有异常的情况下，系统依旧可以继续服务。

这通常是有负载均衡和副本来支撑的。例如计算模块异常可通过负载均衡引流到其他平行节点，存储模块通过其他几点上的副本来对外提供服务。

扩展性

扩展性是融合在 CAP 里面的特性，我觉得此处可以单独讲一下。扩展性直接影响了分布式系统的好坏，系统开发初期不可能把系统的容量、峰值都考虑到，后期肯定牵扯到扩容，而如何做到快而不太影响业务的扩容策略，也是需要考虑的。(后面在介绍数据分布时会着重讨论这个问题)

【分布式系统设计策略】

1、重试机制

一般情况下，写一段网络交互的代码，发起 rpc 或者 http，都会遇到请求超时而失败情况。可能是网络抖动(暂时的网络变更导致包不可达，比如拓扑变更)或者对端挂掉。这时一般处理逻辑是将请求包在一个重试循环块里，如下：

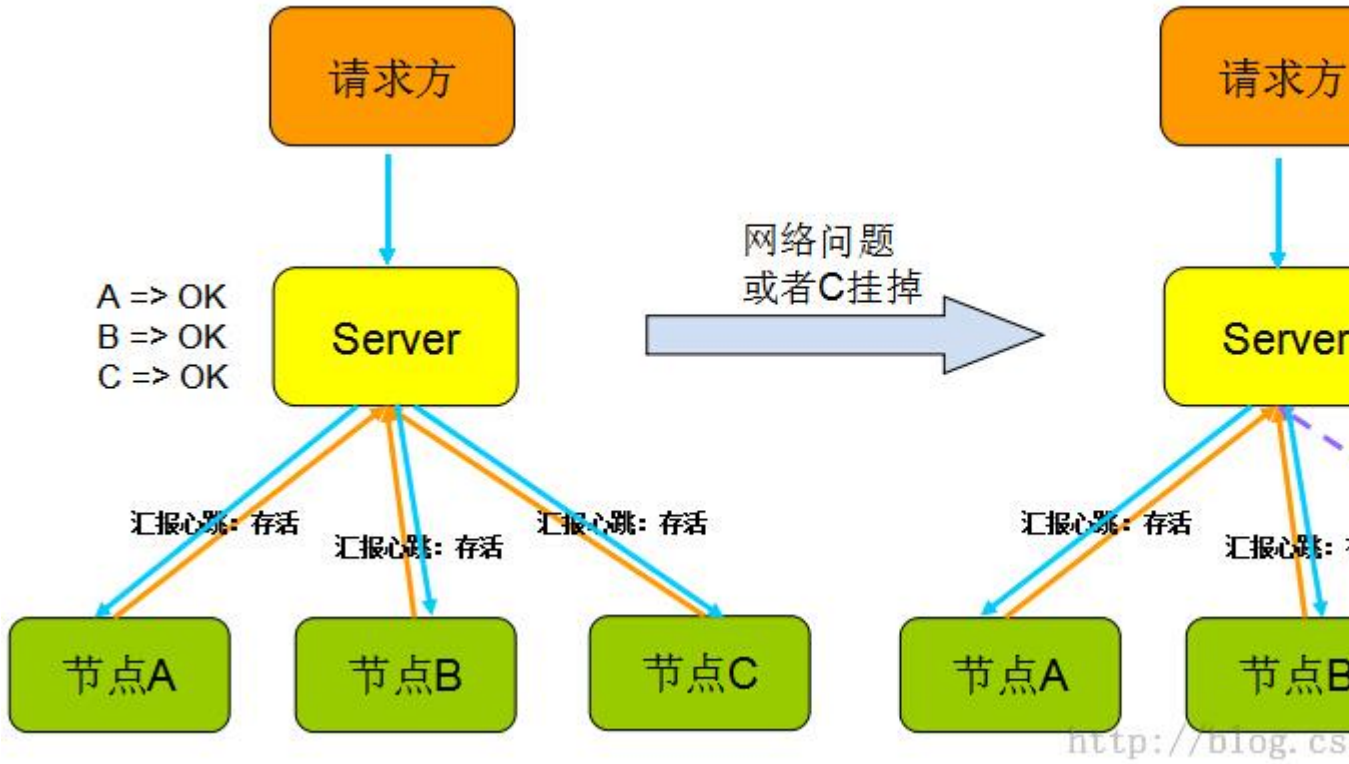
[cpp] view plain copy
print?

```cpp
int retry = 3;
while(!request() && retry--)
    sched_yield();   // or usleep(100)
```

此种模式可以防止网络暂时的抖动，一般停顿时间很短，并重试多次后，请求成功！但不能防止对端长时间不能连接(网络问题或进程问题)

2、心跳机制

心跳顾名思义，就是以固定的频率向其他节点汇报当前节点状态的方式。收到心跳，一般可以认为一个节点和现在的网络拓扑是良好的。当然，心跳汇报时，一般也会携带一些附加的状态、元数据信息，以便管理。如下图：



但心跳不是万能的，收到心跳可以确认 ok，但是收不到心跳却不能确认节点不存在或者挂掉了，因为可能是网络原因倒是链路不通但是节点依旧在工作。

所以切记，”心跳“只能告诉你正常的状态是 ok，它不能发现节点是否真的死亡，有可能还在继续服务。(后面会介绍一种可靠的方式 -- Lease 机制)

3、副本

副本指的是针对一份数据的多份冗余拷贝，在不同的节点上持久化同一份数据，当某一个节点的数据丢失时，可以从副本上获取数据。数据副本是分布式系统解决数据丢失异常的仅有的唯一途径。当然对多份副本的写入会带来一致性和可用性的问题，比如规定副本数为 3，同步写 3 份，会带来 3 次 IO 的性能问题。还是同步写 1 份，然后异步写 2 份，会带来一致性问题，比如后面 2 份未写成功其他模块就去读了(下个小结会详细讨论如果在副本一致性中间做取舍)。

4、中心化/无中心化

系统模型这方面，无非就是两种：

中心节点，例如 mysql 的 MSS 单主双从、MongDB Master、HDFS NameNode、MapReduce JobTracker 等，有 1 个或几个节点充当整个系统的核心元数据及节点管理工作，其他节点都和中心节点交互。这种方式的好处显而易见，数据和管理高度统一集中在一个地方，容易聚合，就像领导者一样，其他人都服从就好。简单可行。

但是缺点是模块高度集中，容易形成性能瓶颈，并且如果出现异常，就像群龙无首一样。

无中心化的设计，例如 cassandra、zookeeper，系统中不存在一个领导者，节点彼此通信并且彼此合作完成任务。好处在于如果出现异常，不会影响整体系统，局部不可用。缺点是比较协议复杂，而且需要各个节点间同步信息。

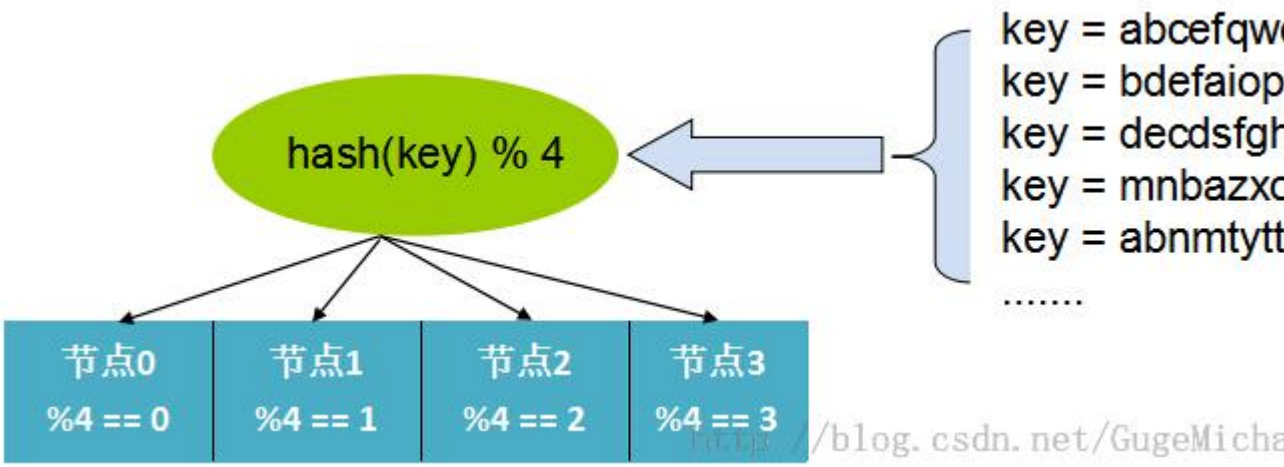
【分布式系统设计实践】

基本的理论和策略简单介绍这么多，后面本人会从工程的角度，细化说一下”数据分布“、"副本控制"和"高可用协议"

在分布式系统中，无论是计算还是存储，处理的对象都是数据，数据不存在于一台机器或进程中，这就牵扯到如何多机均匀分发数据的问题，此小结主要讨论"哈希取模"，”一致性哈希“，”范围表划分“，”数据块划分“

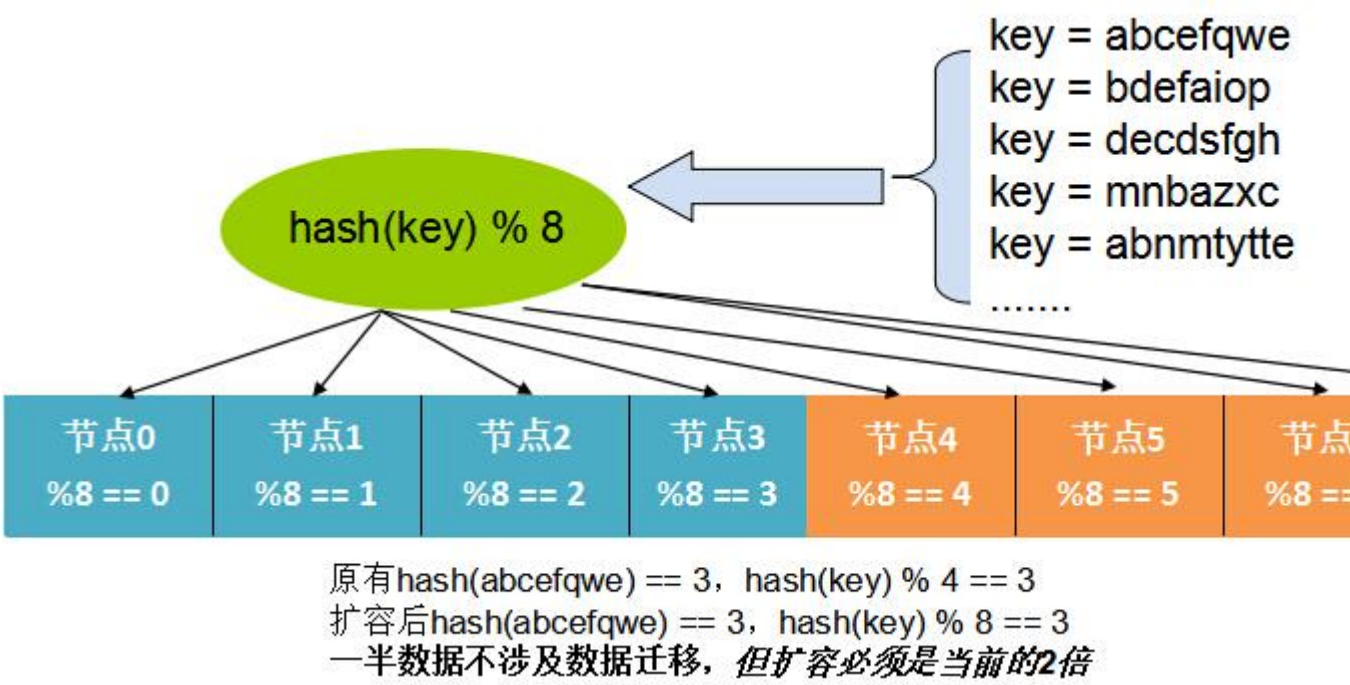
1、哈希取模：

哈希方式是最常见的数据分布方式，实现方式是通过可以描述记录的业务的 id 或 key(比如用户 id)，通过 Hash 函数的计算求余。余数作为处理该数据的服务器索引编号处理。如图：



这样的好处是只需要通过计算就可以映射出数据和处理节点的关系，不需要存储映射。难点就是如果 id 分布不均匀可能出现计算、存储倾斜的问题，在某个节点上分布过重。并且当处理节点宕机时，这种”硬哈希“的方式会直接导致部分数据异常，还有扩容非常困难，原来的映射关系全部发生变更。

此处，如果是”无状态“型的节点，影响比较小，但遇到”有状态“的存储节点时，会发生大量数据位置需要变更，发生大量数据迁移的问题。这个问题在实际生产中，可以通过按 2 的幂的机器数，成倍扩容的方式来缓解，如图：

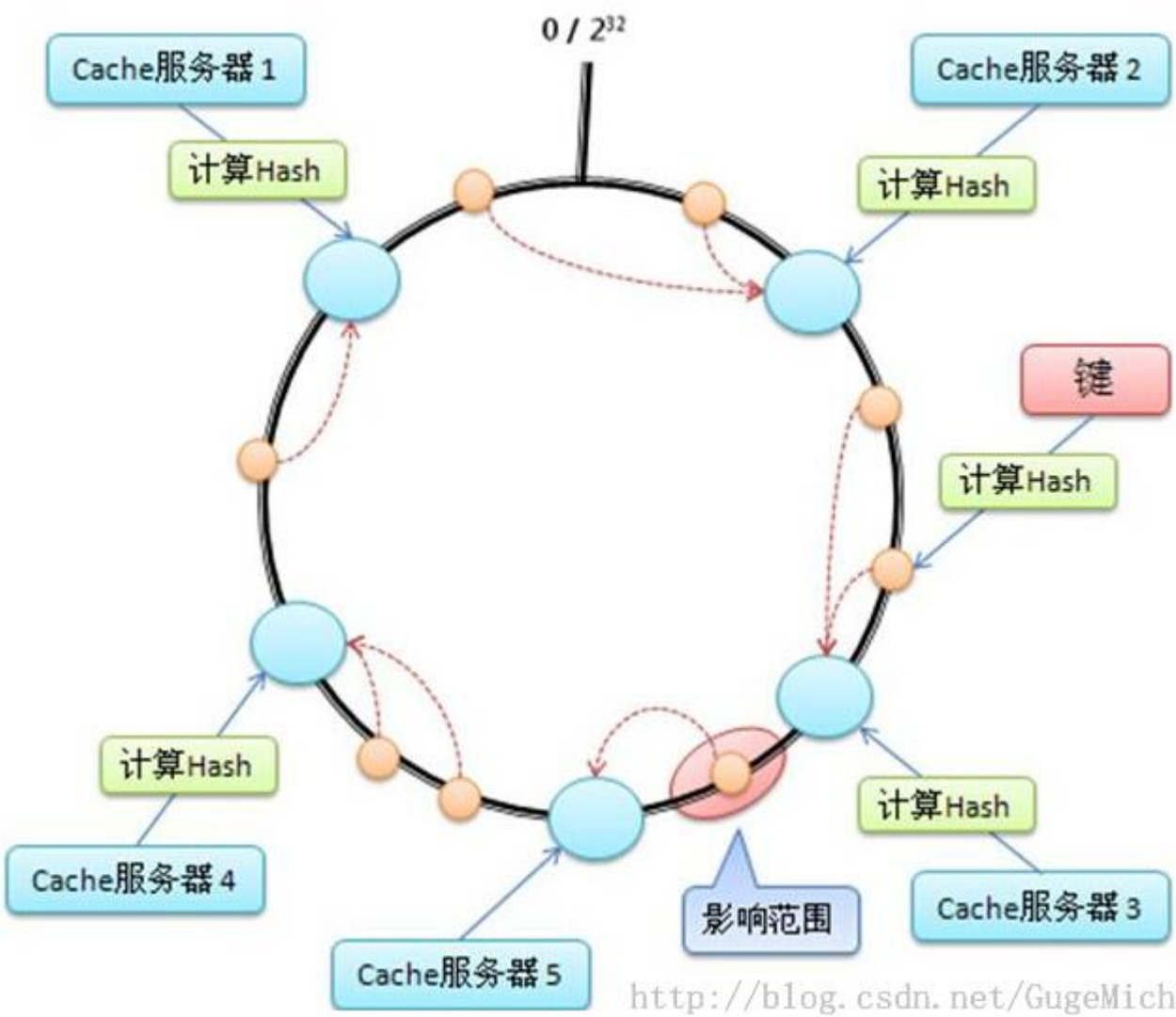


不过扩容的数量和方式后收到很大限制。下面介绍一种”自适应“的方式解决扩容和容灾的问题。

2、一致性哈希：

一致性哈希 -- Consistent Hash 是使用一个哈希函数计算数据或数据特征的哈希值，令该哈希函数的输出值域为一个封闭的环，最大值+1=最小值。将节点随机分布到这个环上，每个节点负责处理从自己开始顺

时针至下一个节点的全部哈希值域上的数据，如图：



##3

一致性哈希的优点在于可以任意动态添加、删除节点，每次添加、删除一个节点仅影响一致性哈希环上相邻的节点。 为了尽可能均匀的分布节点和数据，一种常见的改进算法是引入虚节点的概念，系统会创建许多虚拟节点，个数远大于当前节点的个数，均匀分布到一致性哈希值域环上。读写数据时，首先通过数据的哈希值在环上找到对应的虚节点，然

后查找到对应的 real 节点。这样在扩容和容错时，大量读写的压力会再次被其他部分节点分摊，主要解决了压力集中的问题。如图：

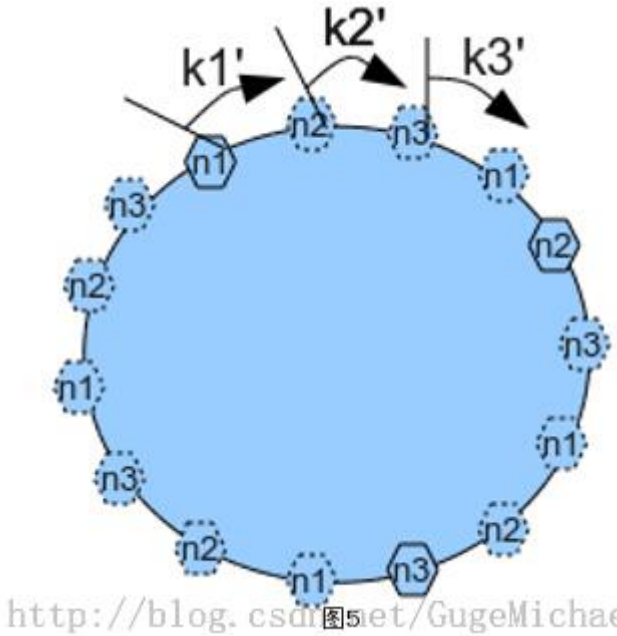


图5

3、数据范围划分：

有些时候业务的数据 id 或 key 分布不是很均匀，并且读写也会呈现聚集的方式。比如某些 id 的数据量特别大，这时候可以将数据按 Group 划分，从业务角度划分比如 id 为 0~10000，已知 8000 以上的 id 可能访问量特别大，那么分布可以划分为[[0~8000],[8000~9000],[9000~1000]]。将小访问量的聚集在一起。

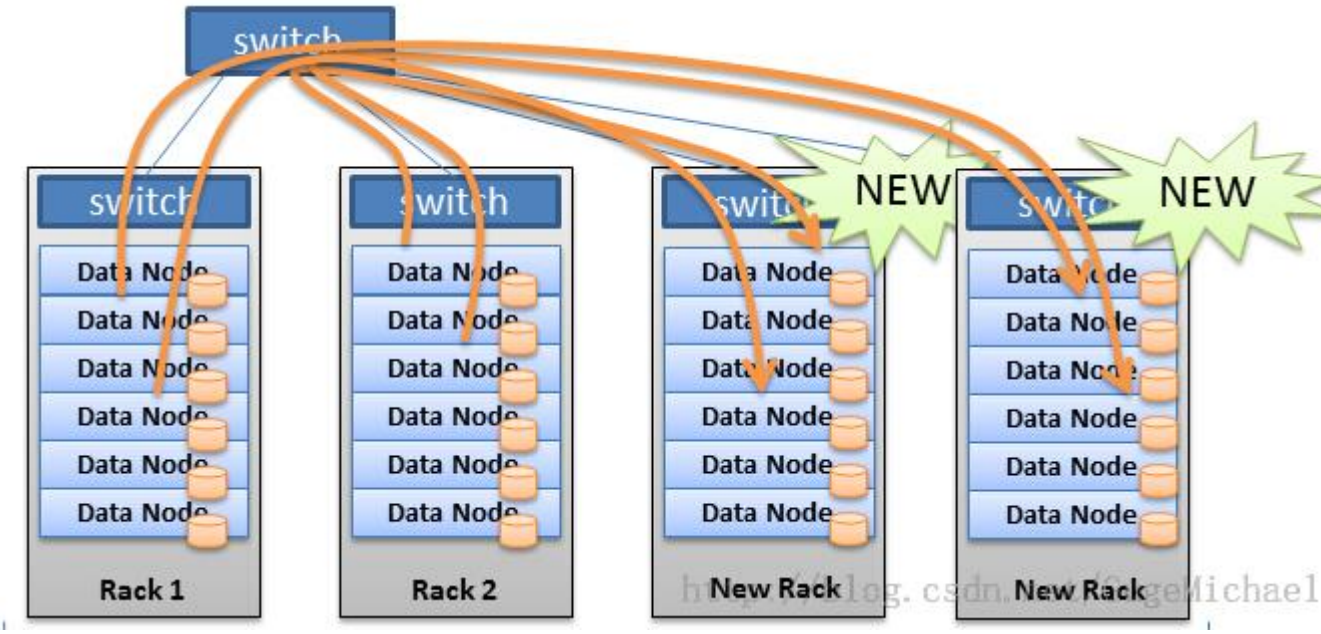
这样可以根据真实场景按需划分，缺点是由于这些信息不能通过计算获取，需要引入一个模块存储这些映射信息。这就增加了模块依赖，可能会有性能和可用性的额外代价。

4、数据块划分：

许多文件系统经常采用类似设计，将数据按固定块大小(比如 HDFS 的 64MB)，将数据分为一个个大小固定的块，然后这些块均匀的分布在各个节点，这种做法也需要外部节点来存储映射关系。

由于与具体的数据内容无关，按数据量分布数据的方式一般没有数据倾斜的问题，数据总是被均匀切分并分布到集群中。当集群需要重新负载均衡时，只需通过迁移数据块即可完成。

如图：



大概说了一下数据分布的具体实施，后面根据这些分布，看看工程中各个节点间如何相互配合、管理，一起对外服务。

1、paxos

paxos 很多人都听说过了，这是唯一一个被认可的在工程中证实的强一致性、高可用的去中心化分布式协议。

虽然论文里提到的概念比较复杂，但基本流程不难理解。本人能力有限，这里只简单的阐述一下基本原理：

Paxos 协议中，有三类角色：

Proposer：Proposer 可以有多个，Proposer 提出议案，此处定义为 value。不同的 Proposer 可以提出不同的甚至矛盾的 value，例如某个 Proposer 提议“将变量 a 设置为 x1”，另一个 Proposer 提议“将变量 a 设置为 x2”，但对同一轮 Paxos 过程，最多只有一个 value 被批准。

Acceptor：批准者。Acceptor 有 N 个，Proposer 提出的 value 必须获得超过半数(N/2+1)的 Acceptor 批准后才能通过。Acceptor 之间对等独立。

Learner：学习者。Learner 学习被批准的 value。所谓学习就是通过读取各个 Proposer 对 value 的选择结果，如果某个 value 被超过半数 Proposer 通过，则 Learner 学习到了这个 value。从而学习者需要至少读取 N/2+1 个 Accpetor，至多读取 N 个

Acceptor 的结果后，能学习到一个通过的 value。

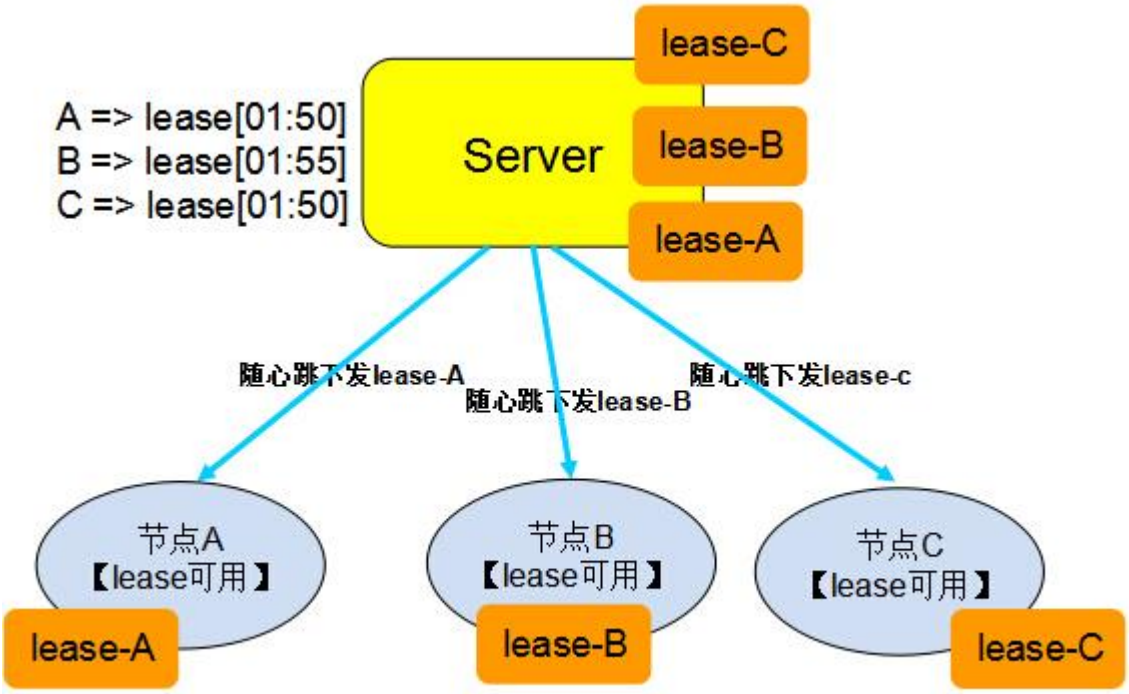
paxos 在开源界里比较好的实现就是 zookeeper(类似 Google chubby)，zookeeper 牺牲了分区容忍性，在一半节点宕机情况下，zookeeper 就不可用了。可以提供中心化配置管理下发、分布式锁、选主等消息队列等功能。其中前两者依靠了 Lease 机制来实现节点存活感知和网络异常检测。

2、Lease 机制

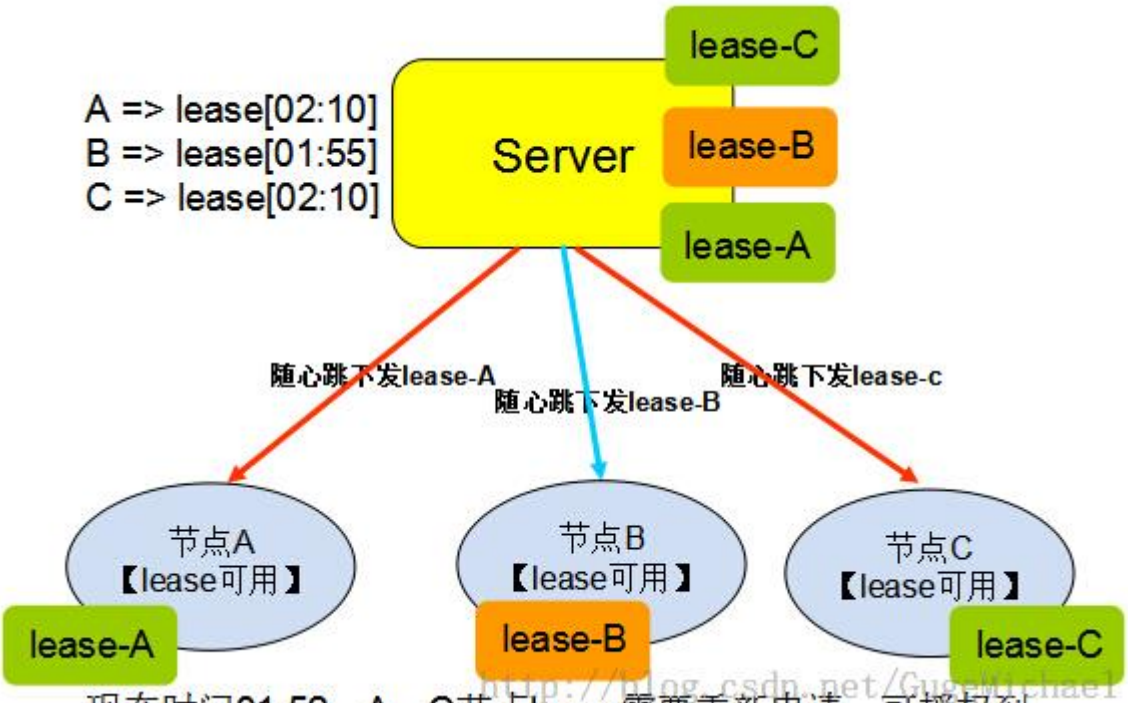
Lease 英文含义是”租期“、”承诺“。在分布式环境中，此机制描述为：

Lease 是由授权者授予的在一段时间内的承诺。授权者一旦发出 lease，则无论接受方是否收到，也无论后续接收方处于何种状态，只要 lease 不过期，授权者一定遵守承诺，按承诺的时间、内容执行。接收方在有效期内可以使用颁发者的承诺，只要 lease 过期，接收方放弃授权，不再继续执行，要重新申请 Lease。

如图：



现在时间01:45，三个节点lease均可用，最长到01:55



现在时间01:52，A、C节点lease需要重新申请，可授权到

Lease 用法举例 1：

现有一个类似 DNS 服务的系统，数据的规律是改动很少，大量的读操作。客户端从服务端获取数据，如果每次都去服务器查询，则量比较大。可以把数据缓存在本地，当数据有变动的时候重新拉取。现在服务器以 lease 的形式，把数据和 lease 一同推送给客户端，在 lease 中存放承诺该数据的不变的时间，然后客户端就可以一直放心的使用这些数据(因为这些数据在服务器不会发生变更)。如果有客户端修改了数据，则把这些数据推送给服务器，服务器会阻塞一直到已发布的所有 lease 都已经超时用完，然后后面发送数据和 lease 时，更新现在的数据。

这里有个优化可以做，当服务器收到数据更新需要等所有已经下发的 lease 超时的这段时间，可以直接发送让数据和 lease 失效的指令到客户端，减小服务器等待时间，如果不是所有的 lease 都失效成功，则退化为前面的等待方案(概率小)。

Lease 用法举例 2：

现有一个系统，有三个角色，选主模块 Manager，唯一的 Master，和其他 salver 节点。slaver 都向 Maganer 注册自己，并由 manager 选出唯一的 Master 节点并告知其他 slaver 节点。当网络出现异常时，可能是 Master 和 Manager 之间的链路断了，Master 认为 Master 已经死掉了，则会再选出一个 Master，但是原来的 Master 对其他网络链路可能都还是正常的，原来的 Master 认为自己还是主节点，继续服务。这时候系统中就出现了”双主“，俗称”脑裂“。

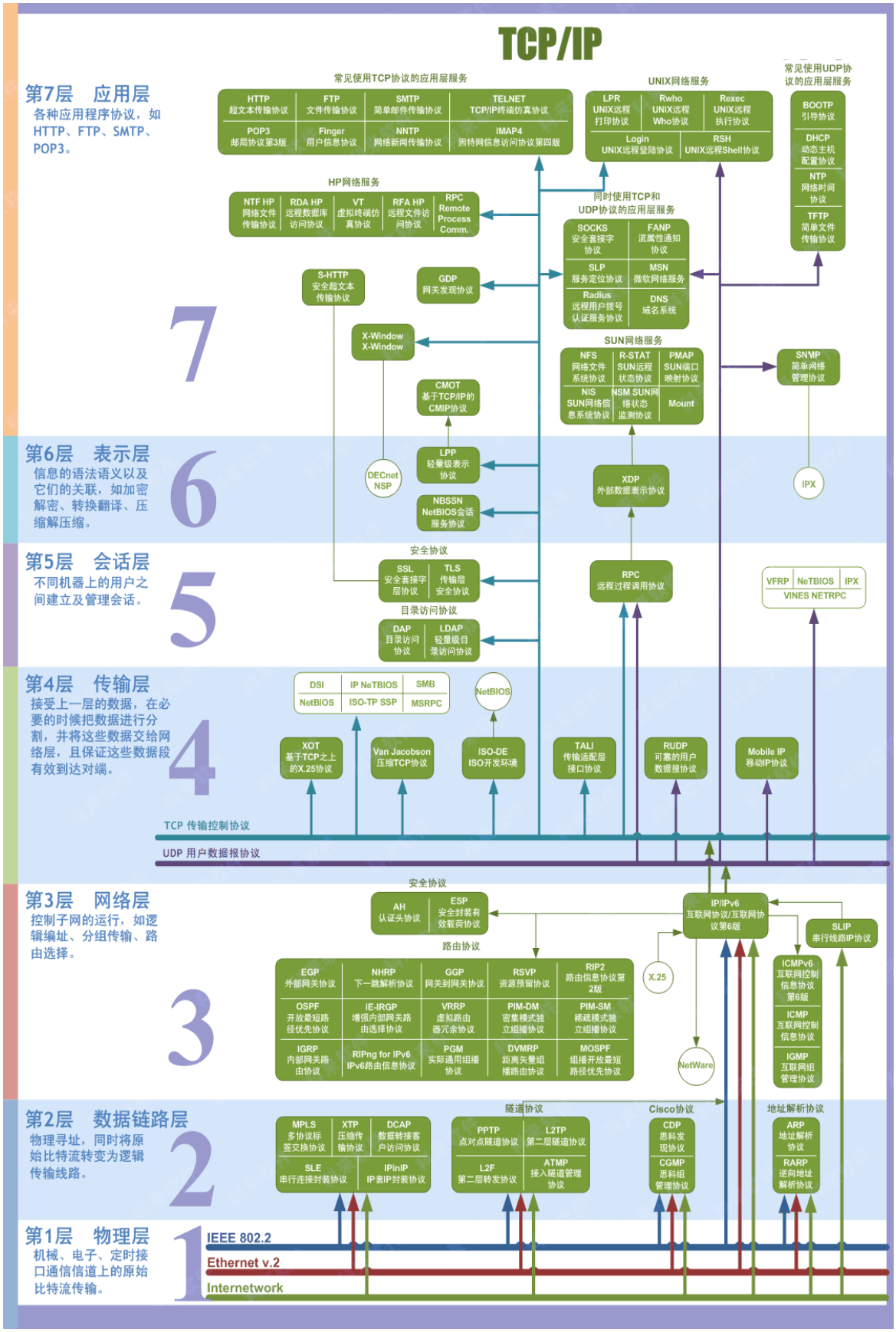
解决这个问题的方式可以通过 Lease，来规定节点可以当 Master 的时间，如果没有可用的 Lease，则自动退化为 Slaver。如果出现”双主“，原 Master 会因为 Lease 到期而放弃当 Master，退化为 Slaver，恢复了一个 Master 的情况。

3、选主算法

有些时候出于系统某些特性，可以在有取舍的情况下，实现一些类似 Lease 的选主的方案，可见本人另一篇文章：

http://blog.csdn.net/gugemichael/article/details/8964834

九、 OSI 七层模型详解


TCP/IP
第7层 应用层
各种应用程序协议，如HTTP、FTP、SMTP、POP3。
常见使用TCP协议的应用层服务
HTTP 超文本传输协议
FTP 文件传输协议
SMTP 简单邮件传输协议
TELNET TCP/IP终端仿真协议
POP3 邮局协议第3版
Finger 用户信息协议
NNTP 网络新闻传输协议
IMAP4 因特网信息访问协议第四版
UNIX网络服务
LPR UNIX远程打印协议
Rwho UNIX远程Who协议
Rexec UNIX远程执行协议
Login UNIX远程登陆协议
RSH UNIX远程Shell协议
常见使用UDP协议的应用层服务
BOOTP 引导协议
DHCP 动态主机配置协议
NTP 网络时间协议
TFTP 简单文件传输协议
HP网络服务
NTF HP 网络文件传输协议
RDA HP 远程数据库访问协议
VT 虚拟终端仿真协议
RFA HP 远程文件访问协议
RPC Remote Process Comm.
同时使用TCP和UDP协议的应用层服务
SOCKS 安全套接字协议
FANP 流属性通知协议
SLP 服务定位协议
MSN 微软网络服务
Radius 远程用户拨号认证服务协议
DNS 域名系统
S-HTTP 安全超文本传输协议
GDP 网关发现协议
X-Window X-Window
CMOT 基于TCP/IP的CMIP协议
SUN网络服务
NFS 网络文件系统协议
R-STAT SUN远程状态协议
PMAP SUN端口映射协议
NIS SUN网络信息系统协议
NSM SUN网络状态监测协议
Mount
SNMP 简单网络管理协议
7
第6层 表示层
信息的语法语义以及它们的关联，如加密解密、转换翻译、压缩解压缩。
6
DECnet NSP
LPP 轻量级表示协议
NBSSN NetBIOS会话服务协议
XDP 外部数据表示协议
IPX
第5层 会话层
不同机器上的用户之间建立及管理会话。
5
安全协议
SSL 安全套接字层协议
TLS 传输层安全协议
目录访问协议
DAP 目录访问协议
LDAP 轻量级目录访问协议
RPC 远程过程调用协议
VFRP NeTBIOS IPX VINES NETRPC
第4层 传输层
接受上一层的数据，在必要的时候把数据进行分割，并将这些数据交给网络层，且保证这些数据段有效到达对端。
4
DSI NetBIOS
IP NeTBIOS ISO-TP SSP
SMB MSRPC
NetBIOS
XOT 基于TCP之上的X.25协议
Van Jacobson 压缩TCP协议
ISO-DE ISO开发环境
TALI 传输适配层接口协议
RUDP 可靠的用户数据报协议
Mobile IP 移动IP协议
TCP 传输控制协议
UDP 用户数据报协议
第3层 网络层
控制子网的运行，如逻辑编址、分组传输、路由选择。
3
安全协议
AH 认证头协议
ESP 安全封装有效载荷协议
IP/IPv6 互联网协议/互联网协议第6版
SLIP 串行线路IP协议
路由协议
EGP 外部网关协议
NHRP 下一跳解析协议
GGP 网关到网关协议
RSVP 资源预留协议
RIP2 路由信息协议第2版
OSPF 开放最短路径优先协议
IE-IRGP 增强内部网关路由选择协议
VRRP 虚拟路由器冗余协议
PIM-DM 密集模式独立组播协议
PIM-SM 稀疏模式独立组播协议
IGRP 内部网关路由协议
RIPng for IPv6 IPv6路由信息协议
PGM 实际通用组播协议
DVMRP 距离矢量组播路由协议
MOSPF 组播开放最短路径优先协议
X.25
NetWare
ICMPv6 互联网控制信息协议第6版
ICMP 互联网控制信息协议
IGMP 互联网组管理协议
第2层 数据链路层
物理寻址，同时将原始比特流转变为逻辑传输线路。
2
隧道协议
MPLS 多协议标签交换协议
XTP 压缩传输协议
DCAP 数据转接客户访问协议
SLE 串行连接封装协议
IPinIP IP套IP封装协议
PPTP 点对点隧道协议
L2TP 第二层隧道协议
L2F 第二层转发协议
ATMP 接入隧道管理协议
Cisco协议
CDP 思科发现协议
CGMP 思科组管理协议
地址解析协议
ARP 地址解析协议
RARP 逆向地址解析协议
第1层 物理层
机械、电子、定时接口通信信道上的原始比特流传输。
1
IEEE 802.2
Ethernet v.2
Internetwork

OSI 七层模型通过七个层次化的结构模型使不同的系统不同的网络之间实现可靠的通讯，因此其最主要的功能就是帮助不同类型的主机实现数据传输 。

完成中继功能的节点通常称为中继系统。在 OSI 七层模型中，处于不同层的中继系统具有不同的名称。

一个设备工作在哪一层，关键看它工作时利用哪一层的数据头部信息。网桥工作时，是以 MAC 头部来决定转发端口的，因此显然它是数据链路层的设备。

具体说:

物理层：网卡，网线，集线器，中继器，调制解调器

数据链路层：网桥，交换机

网络层：路由器

网关工作在第四层传输层及其以上

集线器是物理层设备,采用广播的形式来传输信息。

交换机就是用来进行报文交换的机器。多为链路层设备(二层交换机)，能够进行地址学习，采用存储转发的形式来交换报文.。

路由器的一个作用是连通不同的网络，另一个作用是选择信息传送的线路。选择通畅快捷的近路，能大大提高通信速度，减轻网络系统通信负荷，节约网络系统资源，提高网络系统畅通率。

交换机和路由器的区别

交换机拥有一条很高带宽的背部总线和内部交换矩阵。交换机的所有的端口都挂接在这条总线上，控制电路收到数据包以后，处理端口会查找内存中的地址对照表以确定目的 MAC（网卡的硬件地址）的 NIC（网卡）挂接在哪个端口上，通过内部交换矩阵迅速将数据包传送到目的端口，目的 MAC 若不存在则广播到所有的端口，接收端口回应后交换机会“学习”新

的地址，并把它添加入内部 MAC 地址表中。

使用交换机也可以把网络“分段”，通过对照 MAC 地址表，交换机只允许必要的网络流量通过交换机。通过交换机的过滤和转发，可以有效的隔离广播风暴，减少误包和错包的出现，避免共享冲突。

交换机在同一时刻可进行多个端口对之间的数据传输。每一端口都可视为独立的网段，连接在其上的网络设备独自享有全部的带宽，无须同其他设备竞争使用。当节点 A 向节点 D 发送数据时，节点 B 可同时向节点 C 发送数据，而且这两个传输都享有网络的全部带宽，都有着自己的虚拟连接。假使这里使用的是 10Mbps 的以太网交换机，那么该交换机这时的总流通量就等于 2×10Mbps = 20Mbps，而使用 10Mbps 的共享式 HUB 时，一个 HUB 的总流通量也不会超出 10Mbps。

总之，交换机是一种基于 MAC 地址识别，能完成封装转发数据包功能的网络设备。交换机可以“学习”MAC 地址，并把其存放在内部地址表中，通过在数据帧的始发者和目标接收者之间建立临时的交换路径，使数据帧直接由源地址到达目的地址。

从过滤网络流量的角度来看，路由器的作用与交换机和网桥非常相似。但是与工作在网络物理层，从物理上划分网段的交换机不同，路由器使用专门的软件协议从逻辑上对整个网络进行划分。例如，一台支持 IP 协议的路由器可以把网络划分成多个子网段，只有指向特殊 IP 地址的网络流量才可以通过路由器。对于每一个接收到的数据包，路由器都会重新计算其校验值，并写入新的物理地址。因此，使用路由器转发和过滤数据的速度往往要比只查看数据包物理地址的交换机慢。但是，对于那些结构复杂的网络，使用路由器可以提高网络的整体效率。路由器的另外一个明显优势就是可以自动过滤网络广播。

集线器与路由器在功能上有什么不同?

首先说 HUB,也就是集线器。它的作用可以简单的理解为将一些机器连接起来组成一个局域网。而交换机（又名交换式集线器）作用与集线器大体相同。但是两者在性能上有区别：集线器采用的式共享带宽的工作方式，而交换机是独享带宽。这样在机器很多或数据量很大时，两者将会有比较明显的。而路由器与以上两者有明显区别，它的作用在于连接不同的网段并且找到网络中数据传输最合适的路径。路由器是产生于交换机之后，就像交换机产生于集线器之后，所以路由器与交换机也有一定联系，不是完全独立的两种设备。路由器主要克服了交换机不能路由转发数据包的不足。

总的来说，路由器与交换机的主要区别体现在以下几个方面：

（1）工作层次不同

最初的的交换机是工作在数据链路层，而路由器一开始就设计工作在网络层。由于交换机工作在数据链路层，所以它的工作原理比较简单，而路由器工作在网络层，可以得到更多的协议信息，路由器可以做出更加智能的转发决策。

（2）数据转发所依据的对象不同

交换机是利用物理地址或者说 MAC 地址来确定转发数据的目的地址。而路由器则是利用 IP 地址来确定数据转发的地址。IP 地址是在软件中实现的，描述的是设备所在的网络。MAC 地址通常是硬件自带的，由网卡生产商来分配的，而且已经固化到了网卡中去，一般来说是不可更改的。而 IP 地址则通常由网络管理员或系统自动分配。

（3）传统的交换机只能分割冲突域，不能分割广播域；而路由器可以分割广播域

由交换机连接的网段仍属于同一个广播域，广播数据包会在交换机连接的所有网段上传播，在某些情况下会导致通信拥挤和安全漏洞。连接到路由器上的网段会被分配成不同的广播域，广播数据不会穿过路由器。虽然第三层以上交换机具有 VLAN 功能，也可以分割广播域，但是各子广播域之间是不能通信交流的，它们之间的交流仍然需要路由器。

（4）路由器提供了防火墙的服务

路由器仅仅转发特定地址的数据包，不传送不支持路由协议的数据包传送和未知目标网络数据包的传送，从而可以防止广播风暴。

物理层

在 OSI 参考模型中，物理层（Physical Layer）是参考模型的最低层，也是 OSI 模型的第一层。

物理层的主要功能是：利用传输介质为数据链路层提供物理连接，实现比特流的透明传输。

物理层的作用是实现相邻计算机节点之间比特流的透明传送，尽可能屏蔽掉具体传输介质和物理设备的差异。使其上面的数据链路层不必考虑网络的具体传输介质是什么。“透明传送比特流”表示经实际电路传送后的比特流没有发生变化，对传送的比特流来说，这个电路好像是看不见的。

数据链路层

数据链路层（Data Link Layer）是 OSI 模型的第二层，负责建立和管理节点间的链路。该层的主要功能是：通过各种控制协议，将有差错的物理信道变为无差错的、能可靠传输数据帧的数据链路。

在计算机网络中由于各种干扰的存在，物理链路是不可靠的。因此，这一层的主要功能是在

物理层提供的比特流的基础上，通过差错控制、流量控制方法，使有差错的物理线路变为无差错的数据链路，即提供可靠的通过物理介质传输数据的方法。

该层通常又被分为介质访问控制（MAC）和逻辑链路控制（LLC）两个子层。

MAC 子层的主要任务是解决共享型网络中多用户对信道竞争的问题，完成网络介质的访问控制；

LLC 子层的主要任务是建立和维护网络连接，执行差错校验、流量控制和链路控制。

数据链路层的具体工作是接收来自物理层的位流形式的数据，并封装成帧，传送到上一层；同样，也将来自上层的数据帧，拆装为位流形式的数据转发到物理层；并且，还负责处理接收端发回的确认帧的信息，以便提供可靠的数据传输。

网络层

网络层（Network Layer）是 OSI 模型的第三层，它是 OSI 参考模型中最复杂的一层，也是通信子网的最高一层。它在下两层的基础上向资源子网提供服务。其主要任务是：通过路由选择算法，为报文或分组通过通信子网选择最适当的路径。该层控制数据链路层与传输层之间的信息转发，建立、维持和终止网络的连接。具体地说，数据链路层的数据在这一层被转换为数据包，然后通过路径选择、分段组合、顺序、进/出路由等控制，将信息从一个网络设备传送到另一个网络设备。

一般地，数据链路层是解决同一网络内节点之间的通信，而网络层主要解决不同子网间的通信。例如在广域网之间通信时，必然会遇到路由（即两节点间可能有多条路径）选择问题。

在实现网络层功能时，需要解决的主要问题如下：

寻址：数据链路层中使用的物理地址（如 MAC 地址）仅解决网络内部的寻址问题。在不同子网之间通信时，为了识别和找到网络中的设备，每一子网中的设备都会被分配一个唯一的地址。由于各子网使用的物理技术可能不同，因此这个地址应当是逻辑地址 (如 IP 地址) 。

交换：规定不同的信息交换方式。常见的交换技术有：线路交换技术和存储转发技术，后者又包括报文交换技术和分组交换技术。

路由算法：当源节点和目的节点之间存在多条路径时，本层可以根据路由算法，通过网络为数据分组选择最佳路径，并将信息从最合适的路径由发送端传送到接收端。

连接服务：与数据链路层流量控制不同的是，前者控制的是网络相邻节点间的流量，后者控制的是从源节点到目的节点间的流量。其目的在于防止阻塞，并进行差错检测。

传输层

OSI 下 3 层的主要任务是数据通信，上 3 层的任务是数据处理。而传输层(Transport Layer)是 OSI 模型的第 4 层。因此该层是通信子网和资源子网的接口和桥梁，起到承上启下的作用。

该层的主要任务是：向用户提供可靠的端到端的差错和流量控制，保证报文的正确传输。传输层的作用是向高层屏蔽下层数据通信的细节，即向用户透明地传送报文。该层常见的协议：TCP/IP 中的 TCP 协议、Novell 网络中的 SPX 协议和微软的 NetBIOS/NetBEUI 协议。

传输层提供会话层和网络层之间的传输服务，这种服务从会话层获得数据，并在必要时，对数据进行分割。然后，传输层将数据传递到网络层，并确保数据能正确无误地传送到网络层。因此，传输层负责提供两节点之间数据的可靠传送，当两节点的联系确定之后，传输层则负责监督工作。综上，传输层的主要功能如下：

传输连接管理：提供建立、维护和拆除传输连接的功能。传输层在网络层的基础上为高层提供“面向连接”和“面向无接连”的两种服务。

处理传输差错：提供可靠的“面向连接”和不太可靠的“面向无连接”的数据传输服务、差错控制和流量控制。在提供“面向连接”服务时，通过这一层传输的数据将由目标设备确认，

如果在指定的时间内未收到确认信息，数据将被重发。

监控服务质量。

会话层

会话层（Session Layer）是 OSI 模型的第 5 层，是用户应用程序和网络之间的接口，主要任务是：向两个实体的表示层提供建立和使用连接的方法。将不同实体之间的表示层的连接称为会话。因此会话层的任务就是组织和协调两个会话进程之间的通信，并对数据交换进行管理。

用户可以按照半双工、单工和全双工的方式建立会话。当建立会话时，用户必须提供他们想要连接的远程地址。而这些地址与 MAC（介质访问控制子层）地址或网络层的逻辑地址不同，它们是为用户专门设计的，更便于用户记忆。域名（DN）就是一种网络上使用的远程地址例如：www.3721.com 就是一个域名。会话层的具体功能如下：

会话管理：允许用户在两个实体设备之间建立、维持和终止会话，并支持它们之间的数据交换。例如提供单方向会话或双向同时会话，并管理会话中的发送顺序，以及会话所占用时间的长短。

会话流量控制：提供会话流量控制和交叉会话功能。

寻址：使用远程地址建立会话连接。l

出错控制：从逻辑上讲会话层主要负责数据交换的建立、保持和终止，但实际的工作却是接收来自传输层的数据，并负责纠正错误。会话控制和远程过程调用均属于这一层的功能。但应注意，此层检查的错误不是通信介质的错误，而是磁盘空间、打印机缺纸等类型的高级错误。

表示层

表示层（Presentation Layer）是 OSI 模型的第六层，它对来自应用层的命令和数据进行解

释，对各种语法赋予相应的含义，并按照一定的格式传送给会话层。其主要功能是“处理用户信息的表示问题，如编码、数据格式转换和加密解密”等。表示层的具体功能如下：

数据格式处理：协商和建立数据交换的格式，解决各应用程序之间在数据格式表示上的差异。

数据的编码：处理字符集和数字的转换。例如由于用户程序中的数据类型（整型或实型、有符号或无符号等）、用户标识等都可以有不同的表示方式，因此，在设备之间需要具有在不同字符集或格式之间转换的功能。

压缩和解压缩：为了减少数据的传输量，这一层还负责数据的压缩与恢复。

数据的加密和解密：可以提高网络的安全性。

应用层

应用层（Application Layer）是 OSI 参考模型的最高层，它是计算机用户，以及各种应用程序和网络之间的接口，其功能是直接向用户提供服务，完成用户希望在网络上完成的各种工作。它在其他 6 层工作的基础上，负责完成网络中应用程序与网络操作系统之间的联系，建立与结束使用者之间的联系，并完成网络用户提出的各种网络服务及应用所需的监督、管理和服务等各种协议。此外，该层还负责协调各个应用程序间的工作。

应用层为用户提供的服务和协议有：文件服务、目录服务、文件传输服务（FTP）、远程登录服务（Telnet）、电子邮件服务（E-mail）、打印服务、安全服务、网络管理服务、数据库服务等。上述的各种网络服务由该层的不同应用协议和程序完成，不同的网络操作系统之间在功能、界面、实现技术、对硬件的支持、安全可靠性以及具有的各种应用程序接口等各个方面的差异是很大的。应用层的主要功能如下：

用户接口：应用层是用户与网络，以及应用程序与网络间的直接接口，使得用户能够与网络进行交互式联系。

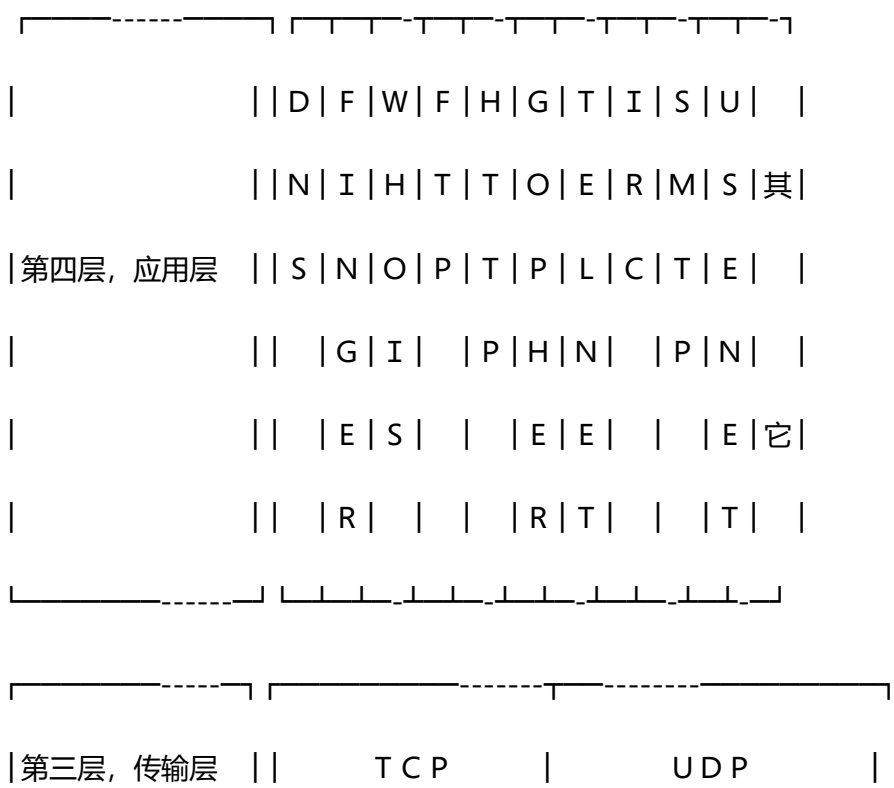
实现各种服务：该层具有的各种应用程序可以完成和实现用户请求的各种服务。

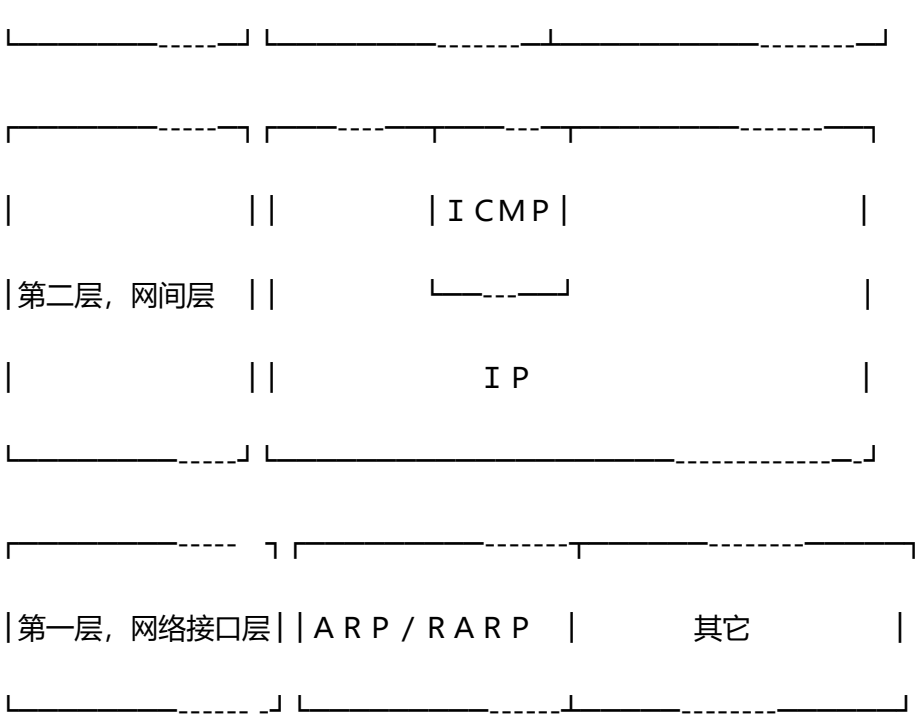
OSI7 层模型的小结

由于 OSI 是一个理想的模型，因此一般网络系统只涉及其中的几层，很少有系统能够具有所有的 7 层，并完全遵循它的规定。

在 7 层模型中，每一层都提供一个特殊的网络功能。从网络功能的角度观察：下面 4 层（物理层、数据链路层、网络层和传输层）主要提供数据传输和交换功能，即以节点到节点之间的通信为主；第 4 层作为上下两部分的桥梁，是整个网络体系结构中最关键的部分；而上 3 层（会话层、表示层和应用层）则以提供用户与应用程序之间的信息和数据处理功能为主。简言之，下 4 层主要完成通信子网的功能，上 3 层主要完成资源子网的功能。

TCP/IP 四层分层模型





TCP/IP 四层参考模型

TCP/IP 协议被组织成四个概念层，其中有三层对应于 ISO 参考模型中的相应层。TCP/IP 分层模型的四个协议层分别完成以下的功能：

第一层:网络接口层

包括用于协作 IP 数据在已有网络介质上传输的协议。实际上 TCP/IP 标准并不定义与 ISO 数据链路层和物理层相对应的功能。相反，它定义像地址解析协议(Address Resolution Protocol,ARP)这样的协议，提供 TCP/IP 协议的数据结构和实际物理硬件之间的接口。

第二层:网间层

对应于 OSI 七层参考模型的网络层。本层包含 IP 协议、RIP 协议(Routing Information Protocol，路由信息协议)，负责数据的包装、寻址和路由。同时还包含网间控制报文协议(Internet Control Message Protocol,ICMP)用来提供网络诊断信息。

第三层:传输层

对应于 OSI 七层参考模型的传输层，它提供两种端到端的通信服务。其中 TCP 协议(Transmission Control Protocol)提供可靠的数据流运输服务，UDP 协议(Use Datagram Protocol)提供不可靠的用户数据报服务。

第四层:应用层

对应于 OSI 七层参考模型的应用层和表达层。因特网的应用层协议包括 Finger、Whois、FTP(文件传输协议)、Gopher、HTTP(超文本传输协议)、Telent(远程终端协议)、SMTP(简单邮件传送协议)、IRC(因特网中继会话)、NNTP（网络新闻传输协议）等，这也是本书将要讨论的重点。