第九章 池化和负载均衡中间件

李会格 博士/讲师 京江学院
E-mail: 1034434100@qq.com

前言

- 在高并发环境下，程序涉及到大量系统调用，消耗大量的CPU资源，频繁申请释放小块内存的部分代码常常成为整个程序的性能瓶颈。池化技术能够减少资源对象的创建次数，提高程序的性能。
- 随着互联网的高速发展，服务器的请求数据量越来越大，出现了服务器负载均衡的解决方案，以消除单点故障，实现系统的高可用性。
- 本章将介绍数据库连接池、对象池和线程池等池化技术，并简要介绍负载均衡的解决方案。

大纲

- 资源池技术概述
- 对象池技术
- 数据库连接池技术
- 线程池技术

- 负载均衡技术概述
- 典型负载均衡技术

资源池技术概述

- 当程序创建线程或者在堆上申请内存时，涉及很多系统调用。特别是当程序有很多类似线程，或频繁申请释放小块内存，这部分代码会成为整个程序的性能瓶颈。

- 资源池（Resource Pool）是涉及资源共享方面的一个著名的设计模式。资源池提前保存大量的资源对象，以解决资源频繁分配和释放所造成的性能问题。

- 线程、内存、数据库连接对象等都可称为资源。

资源池技术概述

池化技术
- 对象池技术
- 数据库连接池技术
- 线程池技术

- 对象池技术的核心是缓存和共享，即对于那些被频繁使用的对象，在使用完后不立即将它们释放，而是缓存起来。这样后续的应用程序可以重复使用这些对象，从而减少创建对象和释放对象的次数，改善应用程序的性能。

资源池技术概述

- 数据库连接池为数据库连接建立一个"缓冲池"。预先在缓冲池中放入一定数量的连接，当需要建立数据库连接时，只需从"缓冲池"中取出一个，使用完毕之后再放回去。通过设定最大连接数，可以防止系统无尽地与数据库连接，并为系统开发、测试及性能调整提供依据。

- 线程池的原理和连接池基本相同，只不过线程池针对的是线程的创建，连接池针对的是数据库连接。

大纲

- 资源池技术概述
- 对象池技术
  - ✓ 对象池技术概述
  - ✓ Commons Pool简介及编程案例
  - ✓ Commons Pool实现原理
- 数据库连接池技术
- 线程池技术
- 负载均衡技术概述
- 典型负载均衡技术

对象池技术

- 对象池的概念
- 对象是面向对象编程中的基本概念。创建一个对象，需要内存资源或其它更多资源，提高服务效率的一个手段就是尽可能减少创建和销毁对象的次数。

- 对象池技术是一种常见的对象缓存手段，就是将具有生命周期的结构化对象缓存到带有一定管理功能的容器中，复用对象以提高对象的访问性能。
- "对象"意味着池中的内容是一种结构化实体，是一般意义上面向对象中的对象模型。

对象池技术

- 对象池的优点
  - 可以复用池中的对象，避免了分配内存和创建堆中对象的开销；
  - 避免了释放内存和销毁堆中对象的开销，进而减少垃圾收集器的负担；
  - 避免内存抖动，不必重复初始化对象状态。

大纲

- 资源池技术概述
- 对象池技术
  - ✓ 对象池技术概述
  - ✓ Commons Pool简介及编程案例
  - ✓ Commons Pool实现原理
- 数据库连接池技术
- 线程池技术
- 负载均衡技术概述
- 典型负载均衡技术

Commons Pool

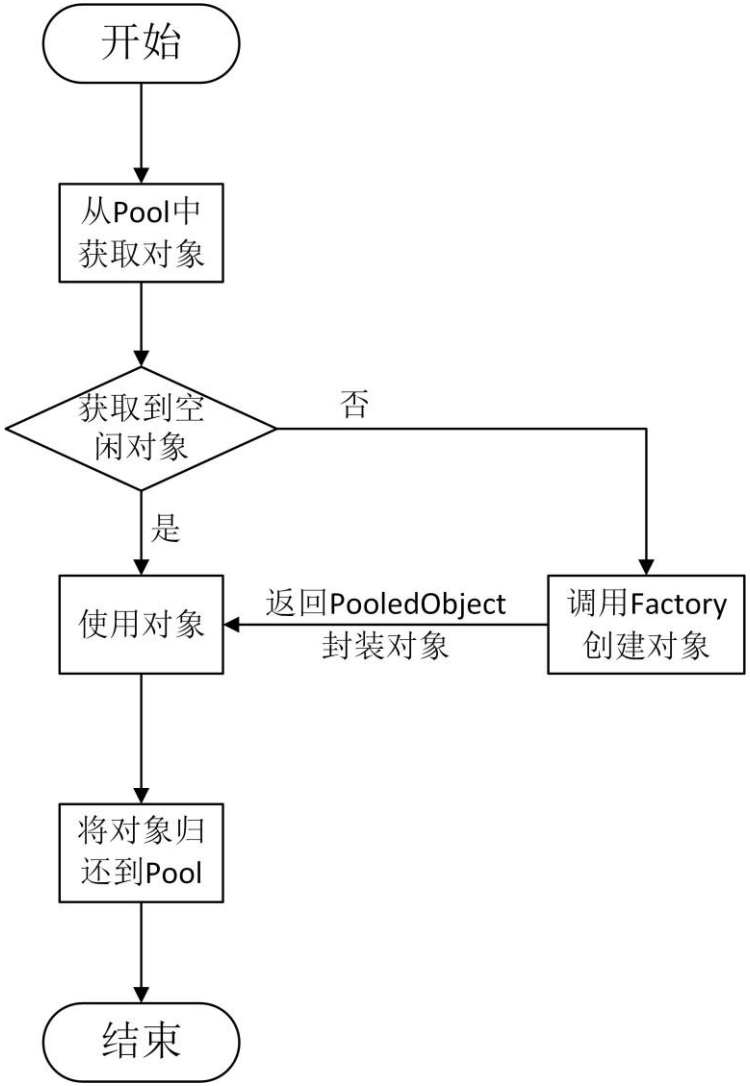
- Commons Pool
- Apache提供了一个通用的对象池技术的实现框架: Commons Pool2，提供了一整套用于实现对象池化的API，以及若干种各具特色的对象池实现，是很多连接池实现的基础。
- Commons Pool2是Apache Commons Pool的第二个大版本。

https://commons.apache.org/proper/commons-pool/

Commons Pool

- Commons Pool的结构
- Common Pool2的核心部分相对简单，围绕着三个基础接口和相关的实现类来实现：
  - 1) ObjectPool：对象池，持有对象并提供"取"和"还"等方法。
  - 2) PooledObjectFactory：对象工厂，提供对象的创建、初始化、销毁等操作，由 Pool 调用。一般需要使用者自己实现这些操作。
  - 3) PooledObject：池化对象，对池中对象的封装，封装对象的状态和一些其他信息。由对象工厂创建的对象就是池化对象。

Commons Pool



- Common Pool2具体的**调用流程**如左图所示，其提供的最基本的实现就是由 Factory 创建对象并使用PooledObject池化对象放入Pool中。在使用对象池时，一般需要基于BasePooledObjectFactory创建自己的对象工厂，并初始化一个对象池，将该工厂与对象池绑定。

Commons Pool

- ObjectPool定义对象池的应该实现的行为有：

接口	功能
addObject()	向池中添加对象
borrowObject()	从池中借走到一个对象
returnObject()	将对象归还给对象池
invalidateObject()	销毁一个对象
getNumIdle()	返回对象池中能够被借走的对象的数量
getNumActive()	返回对象池中正在被使用的对象的数量
Clear()	清理对象池。清理所有空闲对象，释放相关资源
Close()	关闭对象池。清空所有对象及相关资源

Commons Pool

- ObjectPool的核心实现类是GenericObjectPool。GenericObjectPool和GenericKeyedObjectPool是整个Apache Commons Pool的核心实现。

- 此外还实现了软引用对象池SoftReferenceObjectPool，软引用对象池中的对象又被SoftReference封装了一层；ProxiedObjectPool提供了池对象代理功能，防止客户端将池对象还回后还能继续使用。

Commons Pool

- 以下Commons Pool例子实现了由Reader到String的映射转换：

```
public String readToString(Reader in) throws IOException {
    StringBuilder buf = null;
    Closer closer = Closer.create();
    closer.register(in);
    try {
        buf = pool.borrowObject();   // 从对象池中借出对象
        for (int c = in.read(); c != -1; c = in.read())
            buf.append((char) c);
        return buf.toString();
    } catch (IOException e) {
        throw e;
    } finally {
        closer.close();
        try {
            pool.returnObject(buf);  // 归还对象
        } catch (Exception e) {}
    }
}
```

Commons Pool

```
// 继承实现BasePooledObjectFactory对象工厂，进行池对象的生命周期管理
private static class StringBuilderFactory extends BasePooledObjectFactory<StringBuilder> {
    @Override
    public StringBuilder create() throws Exception {  // 创建新对象
        return new StringBuilder();
    }
    @Override
    public PooledObject<StringBuilder> wrap(StringBuilder obj) {  // 将对象包装成池对象
        return new DefaultPooledObject<>(obj);
    }
    @Override
    public void passivateObject(PooledObject<StringBuilder> pooledObject) {
        // 反初始化，归还对象时将被调用
        pooledObject.getObject().setLength(0);
    }

}
```

Commons Pool

```
public class ReaderUtil {
    private ObjectPool<StringBuilder> pool;
    ReaderUtil(ObjectPool<StringBuilder> pool) { this.pool = pool; }
}
public static void main(String[] args) {
    // GenericObjectPool是通用的范型对象池，并将对象工厂与对象池绑定
    ReaderUtil readerUtil = new ReaderUtil(new GenericObjectPool<>(new
        StringBuilderFactory()));
}
```

Commons Pool

- 在以上代码，使用对象池的主要方法pool.borrowObject()和pool.returnObject(buf)进行对象的申请和释放。

- BasePooledObjectFactory是池对象工厂，用于管理池对象的生命周期，只需继承它，并覆写父类相关方法即可控制池对象的生成、初始化、反初始化、校验等。

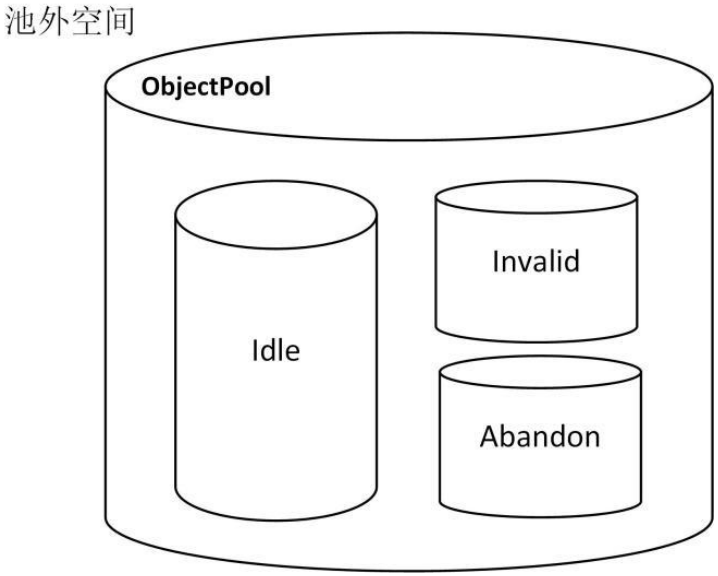
- GenericObjectPool是Apache Commons Pool实现的一个通用泛型对象池，是一个对象池的完整实现，直接构建即可使用。

大纲

- 资源池技术概述
- 对象池技术
  - ✓ 对象池技术概述
  - ✓ Commons Pool简介及编程案例
  - ✓ Commons Pool实现原理
- 数据库连接池技术
- 线程池技术
- 负载均衡技术概述
- 典型负载均衡技术

Commons Pool的实现原理

- 对象池的空间划分
- 对象池空间分为池外空间和池内空间。池外空间是指被"出借"的对象所在的空间（逻辑空间）。池内空间进一步可以划分为idle空间，abandon空间和invalid空间。



Commons Pool的实现原理

- 对象池的空间划分
- idle空间就是空闲对象所在的空间，空闲对象之间是有一定的组织结构的。
- abandon空间又被称作放逐空间，用于放逐被出借的对象。
- invalid空间中的对象将不会再被使用，而是等待被处理掉。

Commons Pool的实现原理

- 池对象的状态
- 对状态的管理是池对象管理最重要的方面。池对象有一套自己的状态机，Commons Pool所定义的池对象状态如下所示：

  - IDLE：空闲状态
  - ALLOCATED：已出借状态
  - EVICTION：正在进行驱逐测试
  - EVICTION_RETURN_TO_HEAD：驱逐测试通过 对象放回到头部

Commons Pool的实现原理

- VALIDATION：空闲校验中
- VALIDATION_PREALLOCATED：出借前校验中
- VALIDATION_RETURN_TO_HEAD：校验通过后放回头部
- INVALID：无效对象
- ABANDONED：放逐中
- RETURNING：换回对象池中

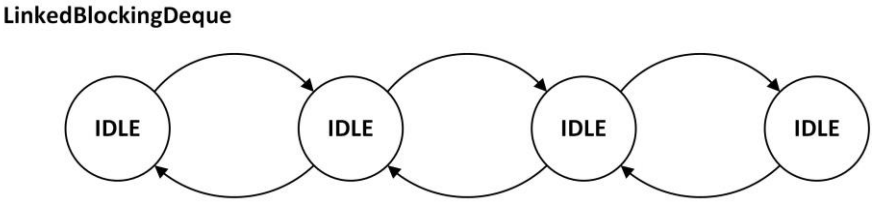
Commons Pool的实现原理

- 池对象的生命周期控制
- Commons Pool通过PooledObjectFactory<T>接口对对象生命周期进行控制。该接口有如下方法：
  - makeObject：创建对象
  - destroyObject：销毁对象
  - validateObject：校验对象
  - activateObject：重新初始化对象
  - passivateObject：反初始化对象

- 需要注意，池对象必须经过创建（makeObject）和初始化过程（activateObject）后才能够被使用。

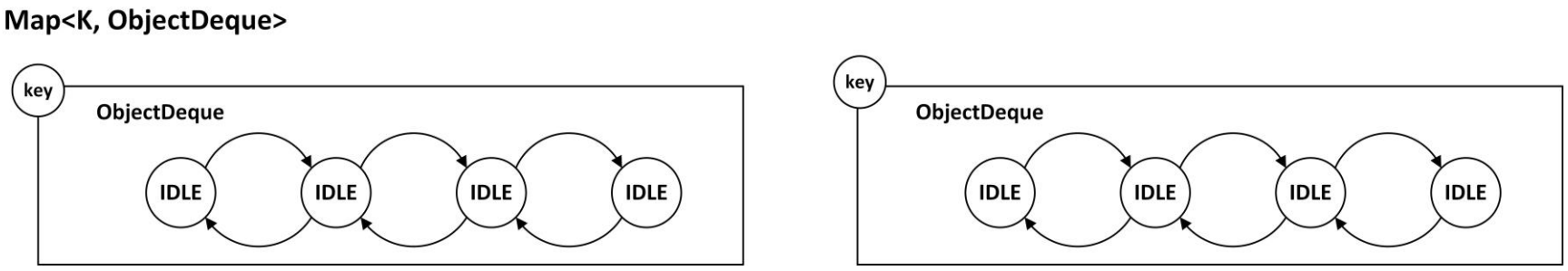
Commons Pool的实现原理

- 池对象组织结构
- 池中的对象具备一定的组织结构。Commons Pool提供了两种组织结构：有界阻塞双端队列(LinkedBlockingDeque)和key桶。
- LinkedBlockingDeque是阻塞队列。在插入或者获取队列元素时，如果队列状态不允许该操作，可能会阻塞住该线程直到队列状态变更为允许操作。阻塞一般有两种情况，一种是插入元素时队列已满，另一种是读取元素时队列为空。



Commons Pool的实现原理

- Commons Pool的另一种组织结构是key桶。每一个key对应一个双端阻塞队列ObjectDeque。ObjectDeque实际上就是包装了的LinkedBlockingDeque，采用这种结构能够对池对象进行一定的划分，从而更加灵活地使用对象池。
- Commons Pool采用KeyedObjectPool<K,V>表示采用这种数据结构的对象池。当对象取出和返还时，都需要指定对应的key空间。



Commons Pool的实现原理

- 对象池的放逐与驱逐
- 驱逐(eviction)和放逐(abandon)这两个概念是对象池设计的核心。EVICTION（驱逐）指的是空闲对象超时销毁，ABANDONED（放逐）指的是不在对象池中的对象超时流放。
- 对象池的一个重要的特性是伸缩性，即对象池能够根据空闲对象的数量（通过maxIdle和minIdle配置）自动进行调整，进而避免内存的浪费。自动伸缩是通过驱逐达到所需要达到的目标。在对象池内部，可以维护一个驱逐定时器(EvictionTimer)，每次达到驱逐时间后就选定一批对象进行驱逐测试。

Commons Pool的实现原理

- 驱逐测试可以采用策略模式，比如Commons Pool的DefaultEvictionPolicy，代码如下：

```
@Override
public boolean evict(EvictionConfig config, PooledObject<T> underTest, int idleCount) {
   if ((config.getIdleSoftEvictTime() < underTest.getIdleTimeMillis() &&
         config.getMinIdle() < idleCount) ||
         config.getIdleEvictTime() < underTest.getIdleTimeMillis()) {
      return true;
   }//根据驱逐定时器检测符合驱逐条件的对象
   return false;
}
```

- 符合驱逐条件的对象将会被对象池驱逐出空闲空间，并丢弃到invalid空间。之后对象池还需要保证内部空闲对象数量需要至少达到minIdle的控制要求。

Commons Pool的实现原理

- 出借时间太长（由removeAbandonedTimeout控制）的对象被称作流浪对象，被放逐的对象被搁置到abandon空间，不允许再次回归到对象池中，进而进入invalid空间被清理。放逐由removeAbandoned()方法实现，分为标记过程和放逐过程。

Commons Pool的实现原理

- 对象池的有效性探测
- 对象池提供了testOnBorrow ，testOnCreate，testOnReturn，testWhileIdle等有效性探测。可以在对象池的功能配置中进行配置。

  - testWhileIdle是当对象处于空闲状态的时候所进行的测试，当测试通过则继续留在对象池中；如果失效，则弃置到invalid空间。

  - testOnBorrow就是当对象出借前进行测试。在测试之前需要调用factory.activateObject()以激活对象，再调用factory.validateObject(p) 对准备出借的对象做有效性检查。

Commons Pool的实现原理

- testOnCreate表示当对象创建之后，进行有效性测试。这并不适用于频繁创建和销毁对象的对象池，与testOnBorrow的行为类似。

- testOnReturn是在对象还回到对象池之前进行的测试。

对象池的应用场景

- 在一些CPU性能不够强，内存较紧张，垃圾收集，内存抖动会造成比较大的影响的应用中，通过对象池可提高内存管理效率，提高系统的响应性。

- 对象池使用的使用场景包含两个方面：
  - 1）处理网络连接，如一些RPC框架的缓存何数据库连接的缓存池等；
  - 2）创建成本高昂的对象，如比较常见的线程池，字节数组池等。从这个角度看，数据库连接池和线程池是对象池的特例。

大纲

- 资源池技术概述
- 对象池技术
- 数据库连接池技术
- 线程池技术

- 负载均衡技术概述
- 典型负载均衡技术

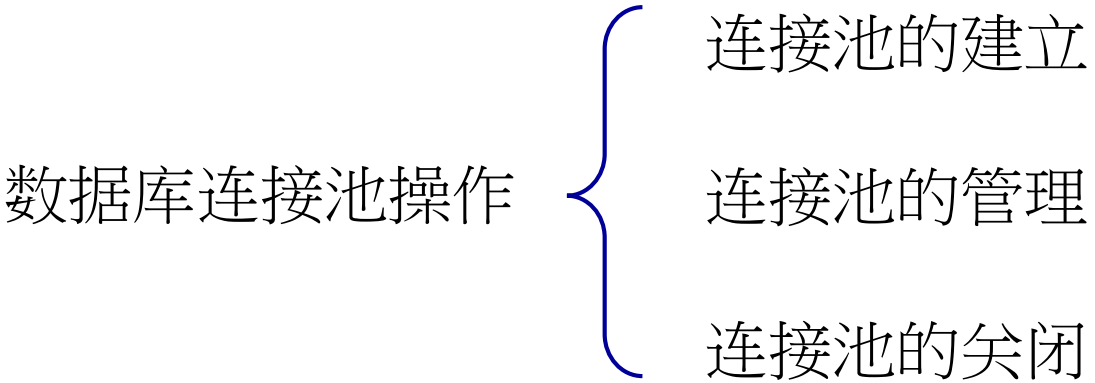
数据库连接池的概念

- 数据库连接是一种关键的、有限的、昂贵的资源，特别在多用户的网页应用程序中尤为突出。一方面，反复打开关闭物理数据库连接会降低系统性能；另一方面，ODBC、JDBC等数据访问中间件对原始连接的封装使数据库连接的复用成为可能。

- 数据库连接池技术（Database connection pool）的核心思想是数据库连接的复用。通过建立一个数据库连接池以及一套连接使用、分配、管理策略，连接池中的连接可以得到高效、安全的复用，避免了数据库连接频繁建立、关闭的开销。

数据库连接池的操作

- 数据库连接池的操作主要分为以下三个方面：



数据库连接池的操作

- 连接池的建立
- 一般地，在系统初始化时，会根据相应的配置创建连接并放置到连接池中，以便需要使用时能从连接池中获取。

- 因此，连接池其实是其静态的。应用程序建立的连接池中的连接在系统初始化时就已分配好，不能随意关闭连接。Java中提供了很多容器类可以方便地构建连接池，如：Vector、Stack、Servlet、Bean等，通过读取连接属性文件Connections.properties与数据库实例建立连接。

数据库连接池的操作

- 连接池的管理
- 连接池管理策略是连接池机制的核心。当连接池建立后，对连接池中的连接进行管理，解决好连接池内连接的分配和释放，对系统的性能有很大的影响。

- 连接的合理分配、释放可提高连接的复用，降低了系统建立新连接的开销，同时也加速了用户的访问速度。一般采用引用记数(Reference Counting)来实现连接池中连接的分配和释放策略。

数据库连接池的操作

- 连接池的关闭
- 当应用程序退出时，应关闭连接池。此时应把在连接池建立时向数据库申请的连接对象统一归还给数据库（即关闭所有数据库连接），这与连接池的建立正好是一个相反的过程。

配置数据库连接池

- 数据库连接池的连接数是影响系统性能的关键参数，一般用minConn和maxConn来限制。
- minConn限定最小数据库连接数。无论这些数据库连接是否被使用，连接池都将一直保证至少拥有这么多的连接数量。
- maxConn限定连接池的最大数据库连接数。当应用程序向连接池请求的连接数超过最大连接数量时，这些请求将被加入到等待队列中，有可能会影响之后的数据库操作。

配置数据库连接池

- 大部分的WEB服务器(Weblogic，WebSphere，Tomcat)都提供了数据源DataSoruce的实现。数据源中都包含了数据库连接池的实现。典型的Java数据库连接池实现有C3P0、BoneCP、DBCP和Proxool等。

- 其中，C3P0是一个开放源代码的JDBC连接池。它在lib目录中与Hibernate一起发布，包括了实现jdbc3和jdbc2扩展规范说明的Connection 和Statement 池的DataSources 对象。

配置数据库连接池

- 只需在hibernate. cfg. xml中加入以下代码，即可完成连接池的配置：

```
<property name="hibernate.c3p0.max_size">20</property>
<property name="hibernate.c3p0.min_size">5</property>
<property name="hibernate.c3p0.timeout">120</property>
<property name="hibernate.c3p0.max_statements">100</property>
<property name="hibernate.c3p0.idle_test_period">120</property>
<property name="hibernate.c3p0.acquire_increment">2</property>
<property name="hibernate.c3p0.validate">true</property>
```

其中，max_size和min_size表示最大和最小连接数。timeout是获得连接的超时时间。如果超过这个时间则会抛出异常。max_statements是最大的PreparedStatement的数量。idle_test_period是检查连接池空闲连接的时间间隔。acquire_increment表示当连接池里面的连接用完的时候，C3P0一次性获取新连接的数量。validate表示是否需要每次都验证连接是否可用。

典型的Java连接池

- 在Java中数据库连接池有以下几种：
- C3P0是一个开放源代码的JDBC连接池。它实现了数据源和JNDI绑定，支持JDBC3规范和JDBC2的标准扩展。目前使用它的开源项目有Hibernate，Spring等。

- DBCP（Database Connection Pool）是Apache 软件基金组织下的开源连接池实现，是一个依赖Jakarta commons-pool对象池机制的数据库连接池。Tomcat 的连接池正是采用该连接池来实现的。该数据库连接池既可以与应用服务器整合使用，也可由应用程序独立使用。但DBCP没有自动回收空闲连接的功能。

典型的Java连接池

- Proxool是一个Java SQL Driver驱动程序，提供了对其它类型的驱动程序的连接池的封装。可以非常简单地移植到现存的代码中。完全可配置、快速、成熟，健壮且可以透明地为现存的JDBC驱动程序增加连接池功能。

- Druid是阿里巴巴开源的数据库连接池项目。Druid连接池为监控而生，内置强大的监控功能，监控特性不影响性能。支持所有JDBC兼容的数据库，包括Oracle、MySql、Derby、Postgresql、SQL Server、H2等。

大纲

- 资源池技术概述
- 对象池技术
- 数据库连接池技术
- 线程池技术

- 负载均衡技术概述
- 典型负载均衡技术

线程池的概念

- 与数据库连接池、对象连接池的原理相似，通过池化线程资源，可以使得更多的CPU时间和内存用来处理应用，而不是频繁的进行线程创建与销毁。

- 为了简化对这些线程的管理，主流开发平台都提供了相应的线程池接口和框架。比如，.NET框架为每个进程提供了一个线程池。一个线程池有若干个等待操作状态，当一个等待操作完成时，线程池中的辅助线程会执行回调函数。线程池中的线程由系统管理，程序员不需要费力于线程管理。

线程池的组成

- 线程池的原理类似于操作系统中缓冲区的概念。先启动若干数量的线程，并处于睡眠状态，当客户端有新请求时，就会唤醒线程池中某个睡眠线程，来处理客户端请求。处理完请求后，线程又处于睡眠状态。睡眠的线程仅定期被唤醒以轮循更改或更新状态信息，然后再次进入休眠状态。

线程池的组成

- 线程池一般包括以下四个基本部分：
  - 1) 线程池管理器（ThreadPool）：用于创建并管理线程池，包括创建线程池，销毁线程池，添加新任务；
  - 2) 工作线程（PoolWorker）：线程池中线程，在没有任务时处于等待状态，可以循环地执行任务；
  - 3) 任务接口（Task）：每个任务必须实现的接口，以供工作线程调度任务的执行，它主要规定了任务的入口，任务执行完后的收尾工作，任务的执行状态等；
  - 4) 任务队列（taskQueue）：用于存放没有处理的任务。提供一种缓冲机制。

大纲

- 资源池技术概述
- 对象池技术
- 数据库连接池技术
- 线程池技术

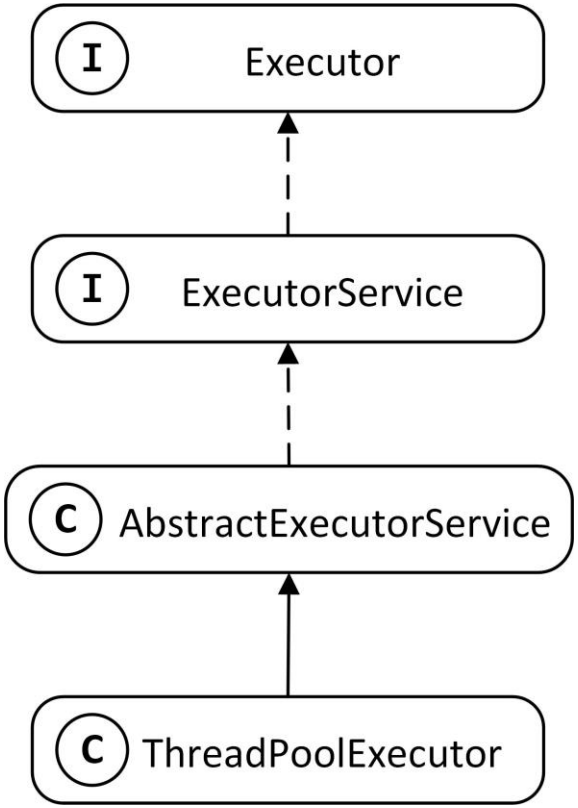
Java线程池技术

- 构造线程池
- 线程池提交任务
- 线程池容量调整
- 任务拒绝策略
- 任务缓存队列
- 关闭线程池
- 编程案例

- 负载均衡技术概述
- 典型负载均衡技术

Java线程池技术

- 构造线程池
- 当创建线程池后，初始时线程池处于RUNNING状态。Java中的线程池核心实现类是java.uitl.concurrent.ThreadPoolExecutor。这个类的设计继承了AbstractExecutorService抽象类和实现了ExecutorService，Executor两个接口，关系大致如右图所示：



Java线程池技术

- ThreadPoolExecutor构造器需要输入以下几个参数：
  - ✓ corePoolSize（线程池的基本大小）：当提交一个任务到线程池时，线程池会创建一个线程来执行任务，即使其它空闲的基本线程能够执行新任务也会创建线程，等到需要执行的任务数大于线程池基本大小时就不再创建。
  - ✓ maximumPoolSize（线程池最大大小）：线程池允许创建的最大线程数。
  - ✓ keepAliveTime（线程活动保持时间）：线程池的工作线程空闲后，保持存活的时间。

Java线程池技术

- unit（线程活动保持时间的单位）：可选的单位有天，小时，分钟，毫秒，微秒和毫微秒
- workQueue：用于保存等待执行的任务的阻塞队列
- threadFactory（线程工厂）：用于设置创建线程的工厂
- handler（饱和策略）：当队列和线程池都满了，说明线程池处于饱和状态，那么必须采取一种策略处理提交的新任务

大纲

- 资源池技术概述
- 对象池技术
- 数据库连接池技术
- 线程池技术

Java线程池技术

- 构造线程池
- 线程池提交任务
- 线程池容量调整
- 任务拒绝策略
- 任务缓存队列
- 关闭线程池
- 编程案例

- 负载均衡技术概述
- 典型负载均衡技术

Java线程池技术

- 线程池提交任务
- 可以通过execute和submit这两个方法向线程池提交任务：

```
void execute(Runnable command);
public <T> Future<T> submit(Runnable task, T result) { };
public <T> Future<T> submit(Callable<T> task) { };
```

- execute方法没有返回值，所以无法判断任务是否被线程池执行成功。
- submit 方法来提交任务会返回一个future。程序可通过future来判断任务是否执行成功。通过future的get方法来获取返回值，get方法会阻塞住直到任务完成。

Java线程池技术

- 一个典型的调用代码是：

```
Future<Object> future = executor.submit(harReturnValuetask);
try {
    Object s = future.get();
} catch (InterruptedException e) {
    // 处理中断异常
} catch (ExecutionException e) {
    // 处理无法执行任务异常
} finally {
    // 关闭线程池
    executor.shutdown();
}
```

Java线程池技术

- 默认情况下，创建线程池之后，线程池中是没有线程的，需要提交任务之后才会创建线程。如果需要线程池创建之后立即创建线程，可以通过以下两个方法：
  - ✓ prestartCoreThread()：初始化一个核心线程；
  - ✓ prestartAllCoreThreads()：初始化所有核心线程。

大纲

- 资源池技术概述
- 对象池技术
- 数据库连接池技术
- 线程池技术

Java线程池技术

- 构造线程池
- 线程池提交任务
- 线程池容量调整
- 任务拒绝策略
- 任务缓存队列
- 关闭线程池
- 编程案例

- 负载均衡技术概述
- 典型负载均衡技术

Java线程池技术

- 线程池容量的动态调整
- ThreadPoolExecutor提供了动态调整线程池容量大小的方法：setCorePoolSize()和setMaximumPoolSize()：
  - ✓ setCorePoolSize：设置核心池大小；
  - ✓ setMaximumPoolSize：设置线程池最大能创建的线程数目大小。

- 当上述参数从小变大时，ThreadPoolExecutor进行线程赋值，还可能立即创建新的线程来执行任务。

大纲

- 资源池技术概述
- 对象池技术
- 数据库连接池技术
- 线程池技术

Java线程池技术

- 构造线程池
- 线程池提交任务
- 线程池容量调整
- 任务拒绝策略
- 任务缓存队列
- 关闭线程池
- 编程案例

- 负载均衡技术概述
- 典型负载均衡技术

Java线程池技术

- 任务拒绝策略
- 当线程池的任务缓存队列已满并且线程池中的线程数目达到maximumPoolSize，如果还有任务到来就会采取任务拒绝策略，Java线程池框架提供了以下4种策略：
  - AbortPolicy：丢弃任务并抛出RejectedExecutionException异常。
  - DiscardPolicy：也是丢弃任务，但是不抛出异常。
  - DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）。
  - CallerRunsPolicy：由调用线程处理该任务。

大纲

- 资源池技术概述
- 对象池技术
- 数据库连接池技术
- 线程池技术

Java线程池技术

- 构造线程池
- 线程池提交任务
- 线程池容量调整
- 任务拒绝策略
- 任务缓存队列
- 关闭线程池
- 编程案例

- 负载均衡技术概述
- 典型负载均衡技术

Java线程池技术

- 任务缓存队列及排队策略
- 任务缓存队列，即workQueue，它用来存放等待执行的任务。可选择的阻塞队列有：
  - ArrayBlockingQueue：一个基于数组结构的有界阻塞队列，此队列按FIFO原则排序元素，队列创建时必须指定大小；
  - LinkedBlockingQueue：一个基于链表结构的无界阻塞队列，此队列按FIFO排序元素。吞吐量通常要高于ArrayBlockingQueue。
  - SynchronousQueue：一个不存储元素的阻塞队列。每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于Linked-BlockingQueue 。
  - PriorityBlockingQueue：一个具有优先级的无限阻塞队列。

大纲

- 资源池技术概述
- 对象池技术
- 数据库连接池技术
- 线程池技术

Java线程池技术

- 构造线程池
- 线程池提交任务
- 线程池容量调整
- 任务拒绝策略
- 任务缓存队列
- 关闭线程池
- 编程案例

- 负载均衡技术概述
- 典型负载均衡技术

Java线程池技术

- 关闭线程池
- 可以通过调用线程池ThreadPoolExecutor的shutdown或shutdownNow方法来关闭线程池。原理是遍历线程池中的工作线程，然后逐个调用线程的interrupt方法来中断线程，所以无法响应中断的任务可能永远无法终止。

- shutdown()方法：线程池处于SHUTDOWN状态，此时线程池不能够接受新的任务，而是会等待所有任务执行完毕。

- shutdownNow()方法：线程池处于STOP状态，此时线程池不能接受新的任务，并且会去尝试终止正在执行的任务。

大纲

- 资源池技术概述
- 对象池技术
- 数据库连接池技术
- 线程池技术

Java线程池技术

- 构造线程池
- 线程池提交任务
- 线程池容量调整
- 任务拒绝策略
- 任务缓存队列
- 关闭线程池
- 编程案例

- 负载均衡技术概述
- 典型负载均衡技术

Java线程池编程案例

- 在下面的线程池案例中，拟用多线程计算1加到1000的总和。程序划分为10个子任务，每个子任务计算100个数的和，然后主线程把各个子任务的和做加总：

```
package chapter09. PoolDemo;
import java. util.concurrent.ThreadPoolExecutor;
import java. util. concurrent. TimeUnit;
import java. util. Random;
import java. util. concurrent. ArrayBlockingQueue;
import java. util. concurrent. Future;
import java. util, concurrent. Callable;
import java, util. concurrent. ExecutionException;
```

Java线程池编程案例

```
public class ThreadPoolFuture {
    final static int coreSize=2; //核心线程数量
    final static int maxSize=5; //最大线程数量
    final static int taskNum=10; //任务数
    public static void main(String[] args) {
        ThreadPoolExecutor executor = new ThreadPoolExecutor(coreSize,
        maxSize,
            200,                TimeUnit.MILLISECONDS,                new
        ArrayBlockingQueue<Runnable>(5));
        Future<Integer>[] results = new Future[taskNum];
        int finalResult=0;
        for (int i = 0; i < taskNum; i++) {
            SomeTask task = new SomeTask(i,i*100+1,i*100+100,executor);
            results[i] = executor.submit(task);
        }//设置计算任务并提交
```

Java线程池编程案例

```
            try {
                for (int i = 0; i < taskNum; i++) {
                    int n=results[i].get();  //获取执行结果
                    finalResult+=n;
                    System.out.println("任务 "+ i+" 运行结果" + n);
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            } catch (ExecutionException e) {
                e.printStackTrace();
            }
            System.out.println(" 加总 最终运算结果:  "+finalResult);
            executor.shutdown();  // 关闭线程池
        }
}
```

Java线程池编程案例

```
class SomeTask implements Callable<Integer> { // 子任务，继承实现Callable
        private int taskNum;
        private int low, high;
        ThreadPoolExecutor executor;  //执行器

        public SomeTask(int id, int low, int high, ThreadPoolExecutor pool) {
                this.taskNum = id;
                this.low=low;
                this.high=high;
                this.executor=pool;
        }
```

```
    @Override
    public Integer call() throws Exception {//回调函数将线程池状态打印到控制台
        System.out.println("任务"+ taskNum+ " 正在执行:" +low+"到" +high +" 的
        加总： ");
        Thread.sleep(1000 +r.nextInt(2000)); //随机休眠1~3s
        int sum = 0;
        for (int i = low; i <= high; i++)
            sum += i;
            System.out.print("task" + taskNum+"执行完毕： ");
            Systen.out.println("池中线程数： "+ executor. getPoolSize()
            + ",等待数:"+ executor.getQueue(). size()
            + ",已完成数:" + executor.getCompletedTaskCount());
        return sum;
    }
}
```

Java线程池编程案例

- 其中，Callable类是java.util.concurrent包下的接口，在它里面声明了一个方法call()。
- Future类是对于具体的Runnable或者Callable任务的执行结果，进行取消、查询是否完成和获取结果。必要时可以通过get方法获取执行结果，该方法会阻塞直到任务返回结果。

Java线程池编程案例

- Future类位于java.util.concurrent包下，它是一个接口：

```
public interface Future<V> {
    boolean cancel(boolean mayInterruptIfRunning);  //取消任务
    boolean isCancelled();  //表示任务是否被取消成功
    boolean isDone();  //表示任务是否已经完成
    V get() throws InterruptedException, ExecutionException;
    //获取执行结果，这个方法会产生阻塞
    V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
    //获取执行结果，如果在指定时间内，还没获取到结果，就直接返回null
}
```

程序执行结果

任务0正在执行: 1到100的加总:
任务8正在执行: 801到900的加总:
任务7正在执行: 701到800的加总:
任务1正在执行: 101到200的加总:
任务9正在执行: 901到1000的加总:
task7执行完毕: 池中线程数:5,等待数:5,已完成数:0
任务2正在执行: 201到300的加总:
task 1执行完毕: 池中线程数:5,等待数:4,已完成数:1
任务3正在执行: 301到400的加总:
task9执行完毕:池中线程数:5,等待数:3,已完成数:2

程序执行结果

任务4正在执行: 401到500的加总:
task0执行完毕: 池中线程数:5,等待数:2,已完成数:3
任务5正在执行: 501到600的加总:
任务0运行结果5050
任务1运行结果15050
task 8执行完毕:池中线程数:5,等待数:1,已完成数:4
任务6正在执行: 601到700的加总:
task 2执行完毕:池中线程数:5,等待数:0,已完成数:5
任务2运行结果25050

程序执行结果

task 5执行完毕: 池中线程数:4,等待数:0,已完成数:6
task 3块行完毕: 池中线程数:4,等待数:0,已完成数:7任各3运行结果35050
task 6执行完毕: 池中线程数:4,等待数:0,已完成数:8
task4执行完毕: 池中线程数:3,等待数:0,已完成数:9
任务4运行结果45050
任务5运行结果55050
任务6运行结果65050
任务7运行结果75050
任务8运行结果85050
任务9运行结果95050
1到1000加总 最终运算结果: 500500

大纲

- 资源池技术概述
- 对象池技术
- 数据库连接池技术
- 线程池技术

- 负载均衡技术概述
- 典型负载均衡技术

负载均衡的概念

- 随着互联网的发展，目前一般会采用计算机集群的方式来应对大量的业务流量与复杂的业务逻辑。但如何将不同的用户的流量分发到不同的服务器上面呢？
- 早期的方法是使用域名系统DNS做负载均衡。但是这种方法有延时性，且调度策略比较简单。

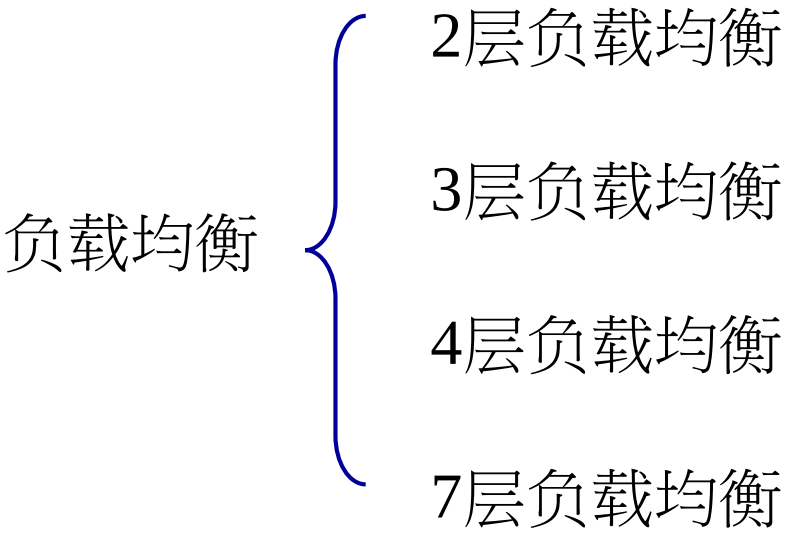
- 负载均衡（Load balancing）是指将工作任务、访问请求等负载进行平衡，分摊到多个服务器和组件等操作单元上进行执行，是解决高性能，单点故障，提高可用性和可扩展性，进行水平伸缩的终极解决方案。

负载均衡的分类

- 从传输的角度看，负载均衡技术主要聚焦在网络传输的协议中进行均衡优化。由于网络层可分为多个层次，因此可以根据网络层次的不同，对负载均衡技术进行分类。
- OSI的七层网络模型如右图所示：

负载均衡的分类

- 常见的负载均衡技术在实现方式中，主要是在应用层（7层）、传输层（4层）、网络层（3层）等做文章。



负载均衡的分类

- **2层负载均衡**
- 负载均衡服务器对外依然提供一个虚IP（VIP），集群中不同的机器采用相同IP地址，但是机器的MAC地址不一样。当负载均衡服务器接受到请求之后，通过改写报文的目标MAC地址的方式将请求转发到目标机器。

- **3层负载均衡**
- 和2层负载均衡类似，负载均衡服务器对外依然提供一个VIP，但是集群中不同的机器采用不同的IP地址。当负载均衡服务器接受到请求之后，根据不同的负载均衡算法，通过IP将请求转发至不同的真实服务器。

负载均衡的分类

- **4层负载均衡**
- 传输层的TCP/UDP协议中除了包含源IP、目标IP以外，还包含源端口号及目的端口号。四层负载均衡服务器在接受到客户端请求后，通过修改数据包的地址信息（IP+端口号）将流量转发到应用服务器。

- **7层负载均衡**
- 应用层协议较多，常用Http、Radius、Dns等。7层负载基于这些协议来均衡负载。比如同一个Web服务器的负载均衡，除了根据IP加端口进行负载外，还可根据7层的URL、浏览器类别、语言来决定是否要进行负载均衡。

大纲

- 资源池技术概述
- 对象池技术
- 数据库连接池技术
- 线程池技术

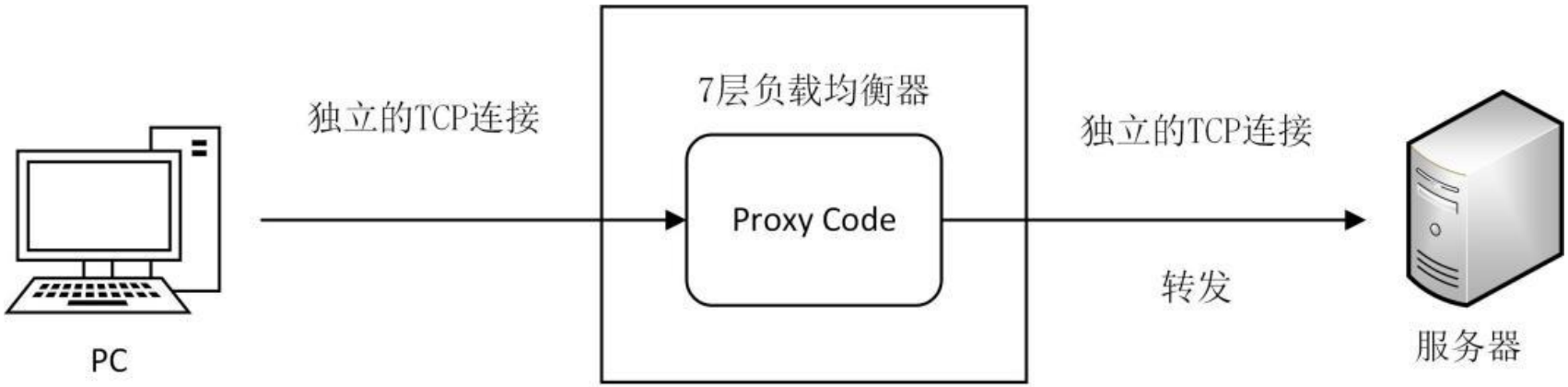
- 负载均衡技术概述
- 典型负载均衡技术

典型负载均衡技术

- 4层负载均衡技术的代表性产品是LVS（开源软件），F5（硬件）。其优点是具有较高的性能、支持各种网络协议。缺点是对网络依赖较大，负载智能化不如7层负载均衡技术。比如不支持对url的个性化负载，且硬件成本较高。

典型负载均衡技术

- 7层负载均衡技术的代表性产品是nginx（软件）、apache（软件）。优点是对网络依赖少，负载智能方案多。缺点是网络协议有限，且性能不如4层负载。



大纲

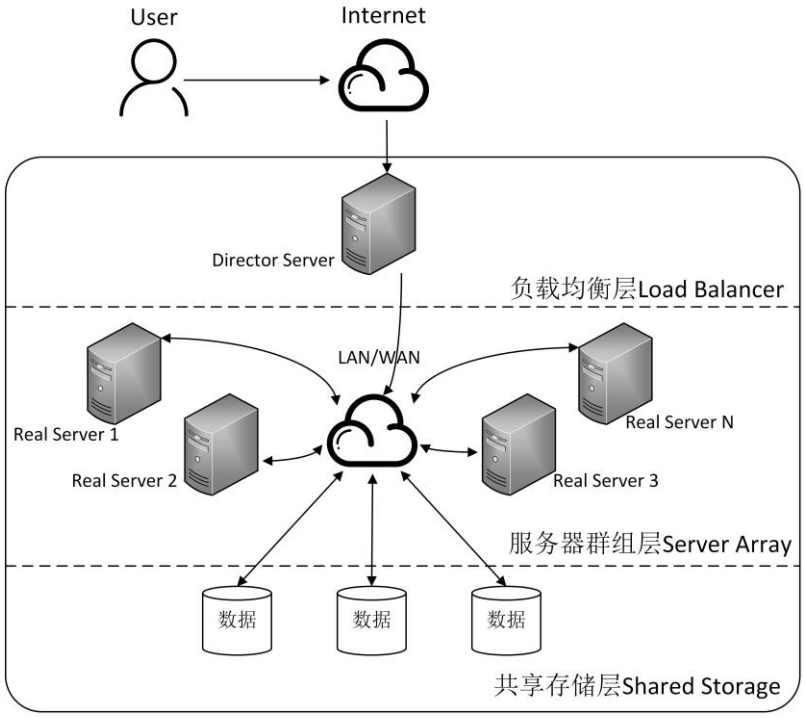
- 资源池技术概述
- 对象池技术
- 数据库连接池技术
- 线程池技术
- 负载均衡技术概述
- 典型负载均衡技术
  - ✓ LVS负载均衡
  - ✓ DNS负载均衡
  - ✓ Nginx负载均衡
  - ✓ F5 BIG-IP负载均衡

LVS负载均衡

- **LVS简介**
- LVS(Linux Virtual Server)，即Linux虚拟服务器，是4层负载均衡的代表性的产品。从Linux2.4以后，LVS成为 Linux标准内核的一部分。
- LVS是一个基于内核级别的应用软件，具有很高的处理性能。如配置百兆网卡，采用VS/TUN或VS/DR调度技术，整个集群系统的吞吐量可高达1 Gbits/s。
- LVS支持大多数的TCP和UDP协议。利用LVS技术可实现高可伸缩的、高可用的网络服务，以低廉的成本实现最优的服务性能。

LVS负载均衡

- **LVS架构**
- LVS架构有三个部分组成：负载均衡层（Load Balancer），服务器群组层（Server Array），数据共享存储层（Shared Storage）。所有的内部应用对用户是透明的。



LVS负载均衡

- Load Balancer层：
- 位于整个集群系统的最前端，由一台或者多台负载调度器（Director Server）组成。LVS模块就安装在Director Server上，而Director Server的主要作用类似于一个路由器，含有完成LVS功能所设定的路由表，通过这些路由表把用户的请求分发给Server Array层的应用服务器（Real Server）上。

LVS负载均衡

- Server Array层：
- 由一组实际运行应用服务的机器组成。Real Server可以是WEB服务器、MAIL服务器等中的一个或者多个，每个Real Server之间通过LAN或分布在各地的WAN相连。在实际的应用中，Director Server也可以同时兼任Real Server的角色。

- Shared Storage层：
- 为所有Real Server提供共享存储空间和内容一致性的存储区域。在物理上，一般由磁盘阵列设备组成。为了提供内容的一致性，可以通过NFS网络文件系统共享数据。

LVS负载均衡

- **IP负载均衡技术**
- LVS的IP负载均衡技术是通过IPVS模块来实现的。IPVS是LVS集群系统的核心软件，主要作用是：安装在Director Server上，同时在Director Server上虚拟出一个IP地址，用户通过虚拟IP（Virtual IP）访问服务。访问的请求首先经过VIP到达负载调度器，然后由负载调度器从Real Server列表中选取一个服务节点响应用户的请求。

IPVS负载均衡机制
- VS/NAT
- VS/TUN
- VS/DR

LVS负载均衡

- VS/NAT（Virtual Server via Network Address Translation）
- 即网络地址翻译技术实现虚拟服务器。当用户请求到达调度器时，调度器将请求报文的目标地址（即虚拟IP地址）改写成选定的Real Server地址，同时报文的目标端口也改成选定的Real Server的相应端口，最后将报文请求发送到选定的Real Server。Real Server返回数据给用户时，执行相反的过程。
- 在NAT方式下，用户请求和响应报文都必须经过Director Server地址重写。当用户请求越来越多时，调度器的处理能力将称为瓶颈。

LVS负载均衡

- VS/TUN（Virtual Server via IP Tunneling）
- 即IP隧道技术实现虚拟服务器。它的连接调度和管理与VS/NAT方式一样，只是它的报文转发方法不同。在VS/TUN方式中，调度器采用IP隧道技术将用户请求转发到某个Real Server，而这个Real Server将直接响应用户的请求，不再经过前端调度器。此外，对Real Server的地域位置没有要求，可以和Director Server位于同一个网段，也可以是独立的一个网络。
- 因此，在TUN方式中，调度器将只处理用户的报文请求，集群系统的吞吐量大大提高。

LVS负载均衡

- VS/DR（Virtual Server via Direct Routing）
- 即用直接路由技术实现虚拟服务器。它的连接调度和管理与VS/NAT和VS/TUN中的一样，但它的报文转发方法又有不同。VS/DR通过改写请求报文的MAC地址，将请求发送到Real Server，而Real Server将响应直接返回给客户，免去了VS/TUN中的IP隧道开销。
- 这种方式是三种负载调度机制中性能最高最好的，但是必须要求Director Server与Real Server都有一块网卡连在同一物理网段上。

LVS负载均衡

- 负载调度算法
- 负载调度器是根据各个服务器的负载情况，动态地选择一台Real Server响应用户请求。动态选择的关键就是负载的调度算法。根据不同的网络服务需求和服务器配置，IPVS实现了8种负载调度算法。这里简要介绍最常用的四种调度算法。

负载调度算法
- 轮叫调度
- 加权轮叫调度
- 最少链接调度
- 加权最少链接调度

LVS负载均衡

- 轮叫调度（Round Robin）
- “轮叫”调度也叫1:1调度。调度器通过“轮叫”调度算法将外部用户请求按顺序1:1的分配到集群中的每个Real Server上，平等地对待每一台Real Server，而不管服务器上实际的负载状况和连接状态。

- 加权轮叫调度（Weighted Round Robin）
- “加权轮叫”调度算法是根据Real Server的不同处理能力来调度访问请求。可以对每台Real Server设置不同的调度权值。同时，调度器还可以自动查询Real Server的负载情况，并动态地调整其权值。

LVS负载均衡

- 最少链接调度（Least Connections）
- "最少连接"调度算法动态地将网络请求调度到已建立的链接数最少的服务器上。如果集群系统的真实服务器具有相近的系统性能，采用"最小连接"调度算法可以较好地均衡负载。
- 加权最少链接调度（Weighted Least Connections）
- "加权最少链接调度"是"最少连接调度"的超集。每个服务节点可以用相应的权值表示其处理能力，而系统管理员可以动态的设置相应的权值，加权最小连接调度在分配新连接请求时尽可能使服务节点的已建立连接数和其权值成正比。

大纲

- 资源池技术概述
- 对象池技术
- 数据库连接池技术
- 线程池技术
- 负载均衡技术概述
- 典型负载均衡技术
  - ✓ LVS负载均衡
  - ✓ DNS负载均衡
  - ✓ Nginx负载均衡
  - ✓ F5 BIG-IP负载均衡

DNS负载均衡

- 域名系统DNS是因特网上把域名和IP地址相互映射的一个分布式数据库，能够使用户更方便地访问互联网。通过主机名，最终得到该主机名对应的IP地址的过程叫做域名解析（或主机名解析）。
- DNS负载均衡技术是最早的负载均衡解决方案。它通过DNS服务中的随机名字解析来实现的。在DNS服务器中，可为同一个域名配置多个不同的地址。查询域名的客户机可获得其中的一个地址。因此对于同一个域名，不同的客户机会得到不同的地址，并访问不同地址上的Web服务器，达到负载均衡的目的。

DNS负载均衡

- DNS负载均衡的优点是实现简单、实施容易、成本低。但缺点也很明显，可能存在以下的问题：
  - 1) 负载分配不均匀：未考虑每个Web服务器当前的负载情况，最慢的Web服务器将成为系统的瓶颈
  - 2) 可靠性低：如果某台Web服务器出现故障，DNS服务器仍然会把请求分配到这台故障服务器上，导致不能响应客户端；
  - 3) 变更生效时间长：更改DNS的配置时，有可能造成相当一部分客户不能使用Web服务；并且由于DNS缓存的原因，所造成的后果要持续相当长一段时间。

大纲

- 资源池技术概述
- 对象池技术
- 数据库连接池技术
- 线程池技术
- 负载均衡技术概述
- 典型负载均衡技术
  - ✓ LVS负载均衡
  - ✓ DNS负载均衡
  - ✓ Nginx负载均衡
  - ✓ F5 BIG-IP负载均衡

Nginx负载均衡

- Nginx是一个高性能的HTTP和反向代理web服务器，同时也提供了IMAP/POP3/SMTP服务。其特点是占有内存少，并发能力强。中国大陆使用nginx网站用户有百度、京东、新浪、网易、腾讯、淘宝等。

- Nginx可以在大多数Unix和Linux操作系统上编译运行，并有 Windows 移植版。在连接高并发的情况下，Nginx是Apache服务不错的替代品，能够支持高达 50,000 个并发连接数的响应。

Nginx负载均衡

- 代理服务就是网络信息的中转站。
- 正向代理用于代理内部网络对外部因特网的连接请求。内部网络的客户机须指定代理服务器，并将本来要直接发送到Web服务器上的http请求发送到代理服务器中。

- 反向代理服务（Reverse Proxy）代理外部网络上的主机对内部网络的访问请求。代理服务器接受英特网上的连接请求，然后将请求转发给内部网络上的服务器，并将从服务器上得到的结果返回给请求连接的客户端。
- 反向代理有多种好处，保护了内部服务器的网页数据。增强了安全性，同时也为负载均衡的实现提供了可能性。

Nginx负载均衡

- **Nginx负载均衡配置**
- 要在Nginx中实现负载均衡，只需修改Nginx的配置文件，其中包含要监听的连接类型以及重定向位置的说明。在Linux环境下，Nginx的默认配置文件位于/usr/local/nginx/conf/nginx.conf中，需要定义以下两个段：upstream 和server 。

Nginx负载均衡

```
worker_processes  1;
events {
    worker_connections  1024;
}
http {
    upstream  backend {
        #这里既可以设置为其他服务器的IP，也可设置为localhost的不同端口
        server  10.1.0.101:80;
        server  10.1.0.102:80;
    }
    #该服务器接受到端口80的所有流量并将其传递给上游upstream
    #请注意，upstream名称和proxy_pass需要匹配
    server {
        listen  80;
        server_name  localhost;

        location  /  {
            proxy_pass  http:// backend;
            proxy_redirect  default;
        }
    }
}
```

Nginx负载均衡

- **Nginx负载调度算法**
- 1）轮叫调度：即轮询方式，详见上述实例。

- 2）加权轮叫调度：根据服务器的不同处理能力来调度访问请求，在下面的例子中，第二台服务器的访问比率将会是第一台的2倍。

```
upstream  backend {
    server  10.1.0.101:80  weight=1;
    server  10.1.0.102:80  weight=2;
}
```

Nginx负载均衡

- 3）最少链接调度：动态地将网络请求调度到已建立的链接数最少的服务器上。最少链接调度可以和加权轮叫调度结合使用。

```
upstream  backend {
    least_conn;
    server  10.1.0.101:80;
    server  10.1.0.102:80;
}
```

- 4）iphash：每个请求都根据访问IP的hash结果分配，每个访问者被定向到同一个后端服务，以提供会话持久性。

```
upstream  backend {
    ip_hash;
    server  10.1.0.101:80;
    server  10.1.0.102:80;
}
```

大纲

- 资源池技术概述
- 对象池技术
- 数据库连接池技术
- 线程池技术
- 负载均衡技术概述
- 典型负载均衡技术
  - ✓ LVS负载均衡
  - ✓ DNS负载均衡
  - ✓ Nginx负载均衡
  - ✓ F5 BIG-IP负载均衡

F5 BIG-IP负载均衡

- F5 BIG-IP是美国 F5 公司的一款集成了网络流量管理、应用程序安全管理、负载均衡等功能的应用交付平台。BIG-IP是一台对流量和内容进行管理分配的设备，提供10种灵活的算法将数据流有效地转发到它所连接的服务器群。
- 用户只需要一台虚拟服务器，客户端的数据流会被BIG-IP灵活地均衡到所有的服务器。BIG-IP除了能够进行不同OSI层面的健康检查之外，还具有扩展内容验证和扩展应用查证两种健康检查方法。

本章小结

- 本章首先介绍了资源池化技术的概念，接着先后介绍了对象池、数据库连接池、线程池等技术，包括Commons Pool的概念和编程，以通过实际编程例子来帮助读者学习理解池化过程。接下来介绍了负载均衡的概念和典型技术方案，包括LVS负载均衡、DNS负载均衡和基于Nginx反向代理的负载均衡技术等。