高并发下连接池：性能飞升的魔法秘籍

原创 往事敬秋风 深度Linux 2025年03月19日 15:30 湖南

点击上方蓝字 免费订阅 选择 置顶公众号

在当今数字化浪潮中，高并发场景无处不在，从电商平台促销时的海量订单处理，到在线直播的实时互动，再到社交平台的消息推送。这些场景下，系统如同置身于汹涌的流量洪峰之中，面临着前所未有的压力。而在这其中，数据库连接操作就像是系统的“交通枢纽”，每一次连接的创建、使用与释放，都关乎着系统的整体性能。

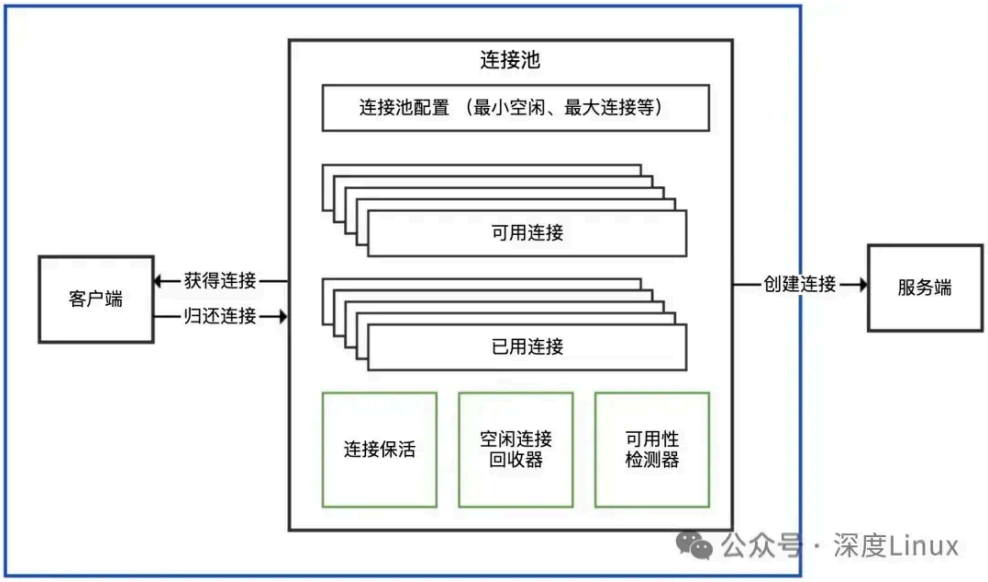
传统模式下频繁创建和销毁数据库连接，就如同在交通高峰期不断开辟和关闭道路，不仅效率低下，还极易引发“交通堵塞”，导致系统响应迟缓甚至崩溃。这时连接池宛如一位智能交通指挥官，通过复用连接资源，极大提升了系统处理高并发请求的效率。今天，就让我们一同深入连接池的设计与优化世界，探寻让系统性能飞升的魔法秘籍 。

深度Linux
研究领域：Windows&Linux平台、C/C++后端开发、嵌入式和Linux系统内核等。
370篇原创内容

公众号

一、为什么要使用连接池？

数据库连接是一种关键的有限的昂贵的资源，这一点在多用户的网页应用程序中体现得尤为突出。一个数据库连接对象均对应一个物理数据库连接，每次操作都打开一个物理连接，使用完都关闭连接，这样造成系统的 性能低下。数据库连接池的解决方案是在应用程序启动时建立足够的数据库连接，并将这些连接组成一个连接池(简单说：在一个“池”里放了好多半成品的数据库连接对象)，由应用程序动态地对池中的连接进行申请、使用和释放。对于多于连接池中连接数的并发请求，应该在请求队列中排队等待。



并且应用程序可以根据池中连接的使用率，动态增加或减少池中的连接数。连接池技术尽可能多地重用了消耗内存地资源，大大节省了内存，提高了服务器地服务效率，能够支持更多的客户服务。通过使用连接池，将大大提高程序运行效率，同时，我们可以通过其自身的管理机制来监视数据库连接的数量、使用情况等。

由于数据库连接的建立和关闭都会消耗系统资源，而且一个数据库服务器能够同时创建的连接数量也是有限的。传统数据库访问的方式是一个数据访问对应一个物理连接，每次操作数据库都需要打开和关闭物理连接，系统性能严重受损。特别是对于复杂的数据库应用，频繁建立关闭连接会极大地降低系统的性能，因此对于连接的使用成了系统性能的瓶颈。通过建立一个数据库连接池以及一套连接使用管理策略，使一个数据库连接可以得到高效且安全的复用，避免了数据库连接频繁建立和关闭带来的开销。

在普通的数据库访问程序的过程中，客户端程序得到的连接是物理连接，调用连接对象的close()方法将关闭连接。而采用连接池技术，客户端程序得到的连接对象是连接池中物理连接的一个句柄，调用连接对象的close()方法，物理连接并没有关闭。数据源的实现只是删除了客户端程序中的连接对象和池中的对象之间的联系。

针对资源共享的设计模式“资源池”为了解决资源频繁分配和释放所造成的问题，将“资源池”模式应用到数据库连接管理领域，也就是建立一个数据库连接池，提供一套高效的连接分配和使用策略，最终实现连接的高效和安全的复用。数据库连接池的基本原理是在内部对象池中维护一定数量的数据库连接，并对外暴露数据库连接获取和返回的方法。

数据库连接池的基本思想是什么样的呢？

连接池的基本思想是在系统初始化时，将数据库连接作为对象存储到内存中，当用户需要访问数据库时，并非建立一个新的连接，而是从连接池中取出一个已经建立的空闲连接对象。当使用完毕后，用户也并非将连接关闭，而是将连接放回连接池中供下一个请求访问使用。连

接池中的连接的建立和断开都时由连接池自身来管理，同时可以通过设置连接池的参数来控制连接池中的初始连接数、连接的上下限数量、每个连接的最大使用次数、最大空闲时间等等。也可以通过自身的管理机制来监控数据库连接的数量和使用情况。

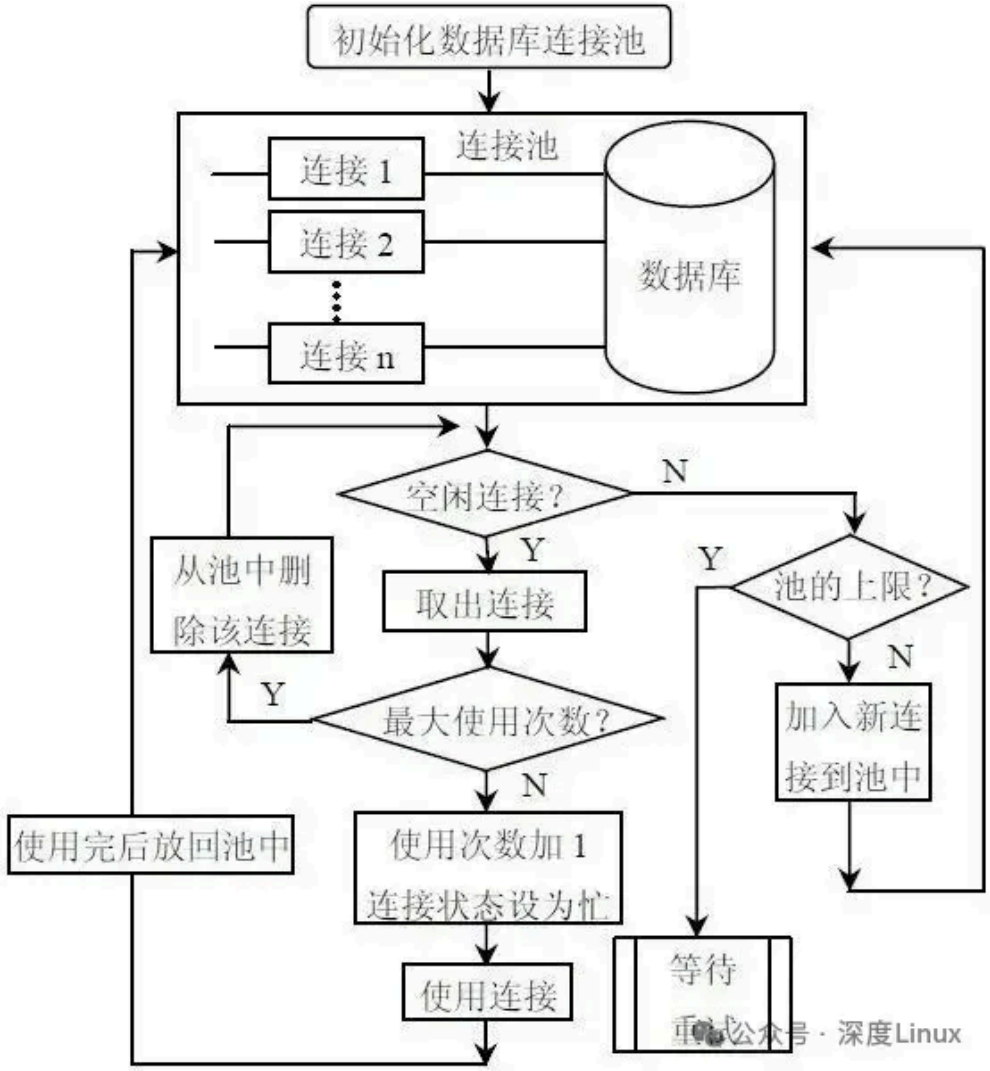
数据库连接池的运行机制是什么样的呢？

程序初始化时创建数据库连接池，服务器启动时建立数据库连接池对象，按照实现设定的参数创建初始数量的数据库连接也就是空闲连接数。

使用时向连接池申请可用连接，对于一个数据库访问请求，直接从连接池中得到一个连接。如果数据库连接池对象中没有空闲的连接，而且连接数没有得到最大活跃连接数，则创建一个新的数据库连接。

使用完毕将连接返还给连接池。数据库存取后关闭，释放所有数据库连接，此时的关闭数据库连接并非真正关闭，而是将其放入空闲队列中。如果实际空闲连接数大于初始空闲连接数则释放连接。

程序退出时断开所有连接并释放资源，释放数据库连接池对象：服务器停止、维护期间释放数据库连接池对象并释放所有连接。



数据库连接池流程

连接池中的连接对象实际上是存放在内存中的，在内存中划分出一块缓存对象，应用程序每次从池中获得连接对象而不是直接从数据库获取，这样不占用服务器的内存资源。

数据库连接池带来的好处是什么呢？

首先来看下如果不使用连接池会出现什么样的情况：占用服务器的内存资源导致服务器速度非常慢

资源重用，由于数据库连接得到重用，避免了频繁地创建、释放连接引起的大量性能开销。在减少系统消耗的基础上，另一方面也增进了系统运行环境的平稳性，减少内存碎片以及数据库临时进程/线程的数量。

更快的系统响应速度，数据库连接池在初始化过程中，往往已经创建了若干数据库连接置于池中备用。此时连接的初始化工作均已完成。对于业务请求处理而言，直接利用现有可用连接，避免了数据库连接初始化和释放过程的时间开销，从而缩减了系统整体响应事件。

新的资源分配手段，对于多应用共享同一数据库的系统而言，可在应用层通过数据库连接的配置，实现数据库连接池。

统一的连接管理避免数据库连接泄漏，在数据库连接池中可根据预先的连接占用超时设定，强制收回被占用连接，从而避免了常规数据库练级操作中可能出现的资源泄漏。

数据库连接池的影响因素是什么呢？

数据库连接池在初始化时创建一定数量的连接并放入连接池中，这些连接的数量是由最小数据库连接数制约的。无论这些连接是否被使用，连接池都将一致保存至少拥有这么多的连接数量。连接池的最大数据库连接数量限定了连接池能占有的最大连接数，当应用程序向连接池请求的连接数量超过最大连接数量时，这些请求将被加入到等待队列中。

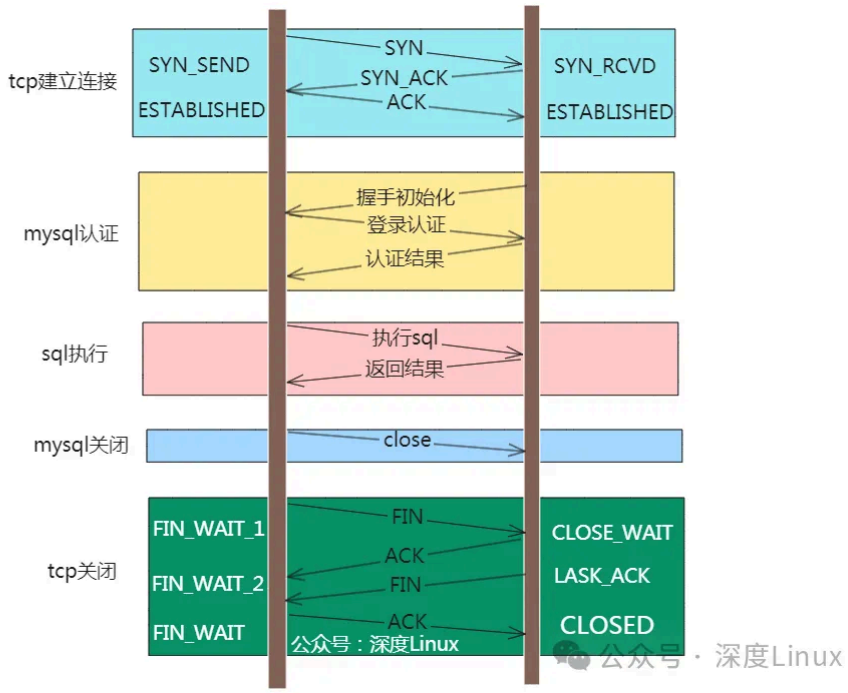
数据库连接池的最小连接数和最大连接数的设置要考虑到以下几个因素：

- 最小连接数量是连接池一直保持的数据库连接，如果应用程序对数据库连接的使用量不大，将会有大量的数据库连接资源被浪费。
- 最大连接数量是连接池能申请到的最大连接数，如果数据库连接请求超过此参数，后面的数据库连接请求将会被加入到等待队列中，这会影响之后的数据库操作。
- 最小连接数量与最大连接数量相差太大的话，那么最先的连接请求将会获利，之后超过最小连接数量的连接请求等价于创建一个新的数据库连接。不过，这些大于最小连接数的连接在使用完毕后不会立即被释放掉，它们将会被放到连接池中等待重复使用或是空闲超时后被释放。

二、运行机制区别

2.1不使用连接池流程

下面以访问MySQL为例，执行一个SQL命令，如果不使用连接池，需要经过哪些流程。



不使用数据库连接池的步骤：

- TCP建立连接的三次握手
- MySQL认证的三次握手
- 真正的SQL执行

- MySQL的关闭
- TCP的四次握手关闭

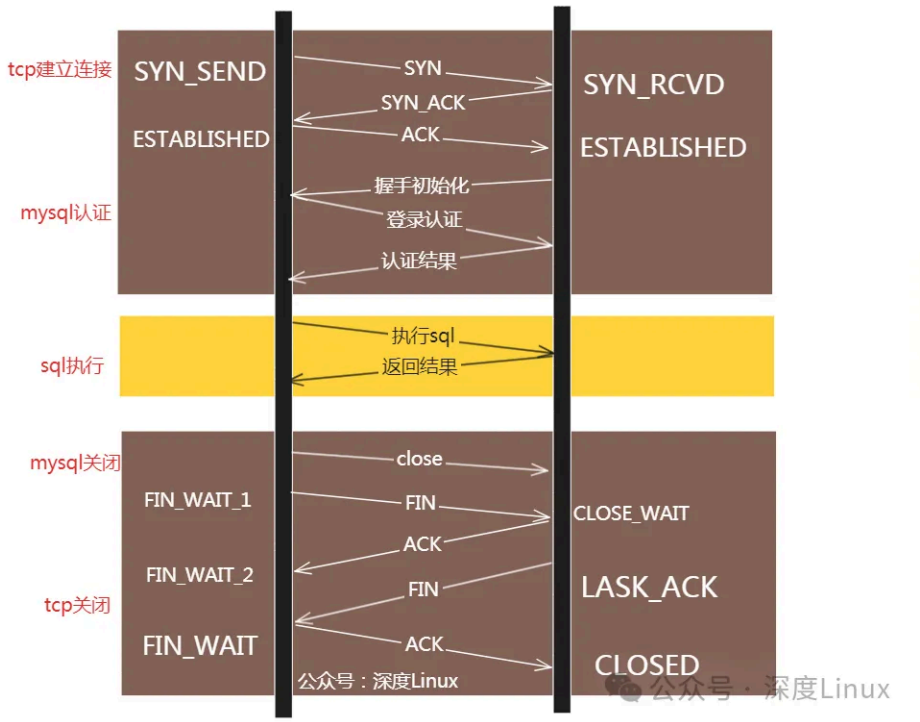
可以看到，为了执行一条SQL，却多了非常多我们不关心的网络交互。

优点：实现简单

缺点：

- 网络IO较多
- 数据库的负载较高
- 响应时间较长及QPS较低
- 应用频繁的创建连接和关闭连接，导致临时对象较多，GC频繁
- 在关闭连接后，会出现大量TIME_WAIT 的TCP状态（在2个MSL之后关闭）

2.2使用连接池流程



使用数据库连接池的步骤：第一次访问的时候，需要建立连接。但是之后的访问，均会复用之前创建的连接，直接执行SQL语句。

优点：

- 较少了网络开销
- 系统的性能会有一个实质的提升
- 没了麻烦的TIME_WAIT状态

三、数据库连接池的工作原理

连接池的工作原理主要由三部分组成，分别为：

- 连接池的建立
- 连接池中连接的使用管理
- 连接池的关闭

第一、连接池的建立。 一般在系统初始化时，连接池会根据系统配置建立，并在池中创建了几个连接对象，以便使用时能从连接池中获取。连接池中的连接不能随意创建和关闭，这样避免了连接随意建立和关闭造成的系统开销。Java中提供了很多容器类可以方便的构建连接池，例如Vector、Stack等。

第二、连接池的管理。 连接池管理策略是连接池机制的核心，连接池内连接的分配和释放对系统的性能有很大的影响。

其管理策略是：

> 当客户请求数据库连接时，首先查看连接池中是否有空闲连接，如果存在空闲连接，则将连接分配给客户使用；如果没有空闲连接，则查看当前所开的连接数是否已经达到最大连接数，如果没达到就重新创建一个连接给请求的客户；如果达到就按设定的最大等待时间进行等待，如果超出最大等待时间，则抛出异常给客户。
> 当客户释放数据库连接时，先判断该连接的引用次数是否超过了规定值，如果超过就从连接池中删除该连接，否则保留为其他客户服务。
> 该策略保证了数据库连接的有效复用，避免频繁的建立、释放连接所带来的系统资源开销。

第三、连接池的关闭。 当应用程序退出时，关闭连接池中所有的连接，释放连接池相关的资源，以便连接可以返回池中重复利用。我们可以通过Connection对象的Close或Dispose方法，也可以通过C#的using语句来关闭连接。该过程正好与创建相反。

注意： 移除无效连接

> 无效连接，即不能正确连接到数据库服务器的连接。对于连接池来说，存储的与数据库服务器的连接的数量是有限的。因此，对于无效连接，如果如不及时移除，将会浪费连接池的空间。其实你不用担心，连接池管理器已经很好的为我们处理了这些问题。如果连接长时间空闲，或检测到与服务器的连接已断开，连接池管理器会将该连接从池中移除。

3.1连接池的诞生

数据库连接的创建和销毁操作之所以开销巨大，是因为这不仅仅是简单的建立或断开一个通道。以常见的关系型数据库 MySQL 为例，当应用程序请求创建一个数据库连接时，首先要进行 TCP 三次握手建立网络连接，这个过程涉及到网络层和传输层的交互，会消耗一定的时间和网络资源。接着，数据库服务器需要对客户端进行身份验证，验证用户名和密码是否正确，这涉及到数据库内部的权限管理系统，需要查询相关的数据表和进行加密解密操作。成功验证后，数据库服务器还需要为该连接分配内存等资源，用于存储连接状态、执行 SQL 语句的上下文信息等 。当连接使用完毕进行销毁时，同样需要进行一系列的资源释放和状态清理工作。

在高并发场景下，假设一个电商系统在促销活动期间，每秒有数千个用户同时进行商品查询、下单等操作，如果每次操作都新建和销毁数据库连接，系统的 CPU 将大部分时间都耗费在这些连接管理操作上，而不是真正执行数据库查询和业务逻辑。这会导致系统响应时间大幅增加，原本可能几毫秒就能完成的数据库查询，由于连接创建和销毁的开销，可能会延长到几百毫秒甚至秒

级，用户在页面上点击查询后，长时间得不到响应，极大地影响用户体验。同时，频繁的连接创建和销毁还可能导致数据库服务器的资源耗尽，如内存不足、文件描述符用尽等问题，进而引发整个系统的崩溃。

连接池的出现，就像是为数据库连接找到了一个高效的“管家”。它通过复用连接，在系统初始化时就预先创建一定数量的连接，并将这些连接存储在一个“池子”里。当应用程序需要连接时，直接从池子里获取，使用完后再归还到池子中。这样就避免了重复进行昂贵的连接创建和销毁操作，大大节省了系统资源，提高了系统的响应速度和吞吐量。例如，在上述电商系统中，使用连接池后，连接获取的时间可以从几十毫秒甚至几百毫秒降低到几毫秒，系统能够轻松应对高并发的请求，保证了系统的稳定运行和良好的用户体验。

(1)DBCP连接池

DBCP 是 Apache 软件基金组织下的开源连接池实现，使用DBCP数据源，应用程序应在系统中增加如下两个 jar 文件：

- Commons-dbcp.jar：连接池的实现
- Commons-pool.jar：连接池实现的依赖库

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。

> 核心类：BasicDataSource
> 使用步骤

> 引入jar文件
> commons-dbcp-1.4.jar

> commons-pool-1.5.6.jar

```
public class App_DBCP {

    // 1. 硬编码方式实现连接池
    @Test
    public void testDbcp() throws Exception {
        // DBCP连接池核心类
        BasicDataSource dataSouce = new BasicDataSource();
        // 连接池参数配置：初始化连接数、最大连接数 / 连接字符串、驱动、用户、密码
        dataSouce.setUrl("jdbc:mysql:///jdbc_demo");            //数据库连接字符串
        dataSouce.setDriverClassName("com.mysql.jdbc.Driver");  //数据库驱动
        dataSouce.setUsername("root");
 //数据库连接用户
        dataSouce.setPassword("root");
//数据库连接密码
        dataSouce.setInitialSize(3);  // 初始化连接
```

```
        dataSouce.setMaxActive(6);       // 最大连接
        dataSouce.setMaxIdle(3000);   // 最大空闲时间

        // 获取连接
        Connection con = dataSouce.getConnection();
                con.prepareStatement("delete  from  admin  where
id=3").executeUpdate();
        // 关闭
        con.close();
    }

    @Test
    // 2. 【推荐】配置方式实现连接池  ,  便于维护
    public void testProp() throws Exception {
        // 加载prop配置文件
        Properties prop = new Properties();
        // 获取文件流
                                    InputStream    inStream    =
App_DBCP.class.getResourceAsStream("db.properties");
        // 加载属性配置文件
        prop.load(inStream);
        // 根据prop配置，直接创建数据源对象
                                    DataSource    dataSouce    =
BasicDataSourceFactory.createDataSource(prop);

        // 获取连接
        Connection con = dataSouce.getConnection();
                con.prepareStatement("delete  from  admin  where
id=4").executeUpdate();
        // 关闭
        con.close();
    }
}
```

配置方式实现DBCP连接池， 配置文件中的key与BaseDataSouce中的属性一样：

```
#基本配置
driverClassName=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/mydb1
username=root
password=123

#初始化池大小，即一开始池中就会有10个连接对象
默认值为0
initialSize=0

#最大连接数，如果设置maxActive=50时，池中最多可以有50个连接，当然这50个连接中
包含被使用的和没被使用的（空闲）
```

```
#你是一个包工头，你一共有50个工人，但这50个工人有的当前正在工作，有的正在空闲
#默认值为8，如果设置为非正数，表示没有限制！即无限大
maxActive=8

#最大空闲连接
#当设置maxIdle=30时，你是包工头，你允许最多有20个工人空闲，如果现在有30个空闲工人，那么要开除10个
#默认值为8，如果设置为负数，表示没有限制！即无限大
maxIdle=8

#最小空闲连接
#如果设置minIdel=5时，如果你的工人只有3个空闲，那么你需要再去招2个回来，保证有5个空闲工人
#默认值为0
minIdle=0

#最大等待时间
#当设置maxWait=5000时，现在你的工作都出去工作了，又来了一个工作，需要一个工人。
#这时就要等待有工人回来，如果等待5000毫秒还没回来，那就抛出异常
#没有工人的原因：最多工人数为50，已经有50个工人了，不能再招了，但50人都出去工作了。
#默认值为-1，表示无限期等待，不会抛出异常。
maxWait=-1

#连接属性
#就是原来放在url后面的参数，可以使用connectionProperties来指定
#如果已经在url后面指定了，那么就不用在这里指定了。
#useServerPrepStmts=true，MySQL开启预编译功能
#cachePrepStmts=true，MySQL开启缓存PreparedStatement功能，
#prepStmtCacheSize=50，缓存PreparedStatement的上限
#prepStmtCacheSqlLimit=300，当SQL模板长度大于300时，就不再缓存它
connectionProperties=useUnicode=true;characterEncoding=UTF8;useServerPrepStmts=true;cachePrepStmts=true;prepStmtCacheSize=50;prepStmtCacheSqlLimit=300

#连接的默认提交方式
#默认值为true
defaultAutoCommit=true

#连接是否为只读连接
#Connection有一对方法：setReadOnly(boolean)和isReadOnly()
#如果是只读连接，那么你只能用这个连接来做查询
#指定连接为只读是为了优化！这个优化与并发事务相关！
#如果两个并发事务，对同一行记录做增、删、改操作，是不是一定要隔离它们啊？
#如果两个并发事务，对同一行记录只做查询操作，那么是不是就不用隔离它们了？
#如果没有指定这个属性值，那么是否为只读连接，这就由驱动自己来决定了。即Connection的实现类自己来决定！
defaultReadOnly=false
```

```
#指定事务的事务隔离级别
# 可 选 值 ： NONE,READ_UNCOMMITTED,  READ_COMMITTED,  REPEATABLE_READ,
SERIALIZABLE
#如果没有指定，那么由驱动中的Connection实现类自己来决定
defaultTransactionIsolation=REPEATABLE_READ
```

(2)C3P0连接池

> C3P0连接池：最常用的连接池技术！Spring框架，默认支持C3P0连接池技术！
> C3P0连接池，核心类：CombopooledDataSource ds

> 使用：下载，引入jar文件: c3p0-0.9.1.2.jar

使用连接池，创建连接

- a)硬编码方式
- b) 配置方式(xml)

配置文件要求：

- 文件名称：必须叫c3p0-config.xml
- 文件位置：必须在src下

c3p0也可以指定配置文件，而且配置文件可以是properties，也可以　　xml的。当然xml的高级一些了。但是c3p0的配置文件名必须为c3p0-config.xml，并且必须放在类路径下。

下面是源码：

```
<?xml version="1.0" encoding="UTF-8"?>
<c3p0-config>
    <default-config>
                                                        <property
name="jdbcUrl">jdbc:mysql://localhost:3306/mydb1</property>
                                                        <property
name="driverClass">com.mysql.jdbc.Driver</property>
        <property name="user">root</property>
        <property name="password">123</property>
        <property name="acquireIncrement">3</property>
        <property name="initialPoolSize">10</property>
        <property name="minPoolSize">2</property>
        <property name="maxPoolSize">10</property>
    </default-config>
    <named-config name="oracle-config">
                                                        <property
name="jdbcUrl">jdbc:mysql://localhost:3306/mydb1</property>
                                                        <property
name="driverClass">com.mysql.jdbc.Driver</property>
        <property name="user">root</property>
```

```
        <property name="password">123</property>
        <property name="acquireIncrement">3</property>
        <property name="initialPoolSize">10</property>
        <property name="minPoolSize">2</property>
        <property name="maxPoolSize">10</property>
    </named-config>
</c3p0-config>

public class App {

    @Test
    //1. 硬编码方式，使用C3P0连接池管理连接
    public void testCode() throws Exception {
        // 创建连接池核心工具类
                        ComboPooledDataSource   dataSource   =   new
ComboPooledDataSource();
        // 设置连接参数：url、驱动、用户密码、初始连接数、最大连接数

dataSource.setJdbcUrl("jdbc:mysql://localhost:3306/jdbc_demo");
        dataSource.setDriverClass("com.mysql.jdbc.Driver");
        dataSource.setUser("root");
        dataSource.setPassword("root");
        dataSource.setInitialPoolSize(3);
        dataSource.setMaxPoolSize(6);
        dataSource.setMaxIdleTime(1000);

        // ---> 从连接池对象中，获取连接对象
        Connection con = dataSource.getConnection();
        // 执行更新
                  con.prepareStatement("delete  from  admin  where
id=7").executeUpdate();
        // 关闭
        con.close();
    }

    @Test
    //2. XML配置方式，使用C3P0连接池管理连接
    public void testXML() throws Exception {
        // 创建c3p0连接池核心工具类
        // 自动加载src下c3p0的配置文件【c3p0-config.xml】
                        ComboPooledDataSource   dataSource   =   new
ComboPooledDataSource();// 使用默认的配置

        // 获取连接
        Connection con = dataSource.getConnection();
        // 执行更新
                  con.prepareStatement("delete  from  admin  where
id=5").executeUpdate();
        // 关闭
```

```
            con.close();
        }    //获取配置文件中“orcale-cofig"的配置信息
    public void fun2() throws PropertyVetoException, SQLException {

                                ComboPooledDataSource    ds    =    new
ComboPooledDataSource("orcale-config");

        Connection con = ds.getConnection();

        System.out.println(con);

        con.close();
    }
}
```

3.2核心组件与运作机制

连接池包含多个核心组件，每个组件都在连接池的高效运作中发挥着关键作用。连接容器是连接池的基础组成部分，它就像一个仓库，用于存储和管理数据库连接。常见的连接容器可以是一个集合，如 Java 中的 ArrayList 或 LinkedList，也可以是更复杂的数据结构，如线程安全的 ConcurrentLinkedQueue。这些数据结构能够方便地进行连接的添加、移除和查找操作。

连接池管理器则是连接池的“指挥官”，负责连接池的整体管理和协调。它维护着连接池的各种状态信息，比如当前连接池中的连接总数、空闲连接数、正在使用的连接数等。连接池管理器根据这些状态信息以及预先设定的配置策略，来决定何时创建新的连接、何时销毁空闲连接以及如何分配连接给应用程序。例如，当应用程序请求连接时，连接池管理器会检查连接容器中是否有空闲连接，如果有，则直接将空闲连接分配给应用程序；如果没有空闲连接，且当前连接数未达到最大连接数限制，那么连接池管理器会创建新的连接并分配给应用程序。

连接池的运作过程涵盖了连接获取、释放、创建和销毁等多个关键操作。当应用程序需要数据库连接时，会向连接池发起获取连接的请求。连接池管理器首先检查连接容器中是否有空闲连接，如果有，就从连接容器中取出一个空闲连接，并将其标记为正在使用状态，然后返回给应用程序。如果连接容器中没有空闲连接，连接池管理器会根据配置策略进行处理。如果当前连接数小于最大连接数，连接池管理器会创建新的连接，将其标记为正在使用状态后返回给应用程序；如果当前连接数已经达到最大连接数，应用程序可能需要等待，直到有其他连接被释放回连接池，或者等待超时后抛出异常。

当应用程序使用完数据库连接后，会将连接释放回连接池。连接池管理器接收到释放的连接后，会将其标记为空闲状态，并将其放回连接容器中，供后续的请求复用。在连接的使用过程中，连接池还会定期对连接进行检测，确保连接的有效性。如果发现某个连接已经失效，比如因为网络故障或数据库服务器重启导致连接断开，连接池会将该连接从连接容器中移除，并销毁该连接资源，以避免无效连接占用资源。

连接池的创建操作通常在系统初始化阶段进行。根据配置的初始连接数，连接池管理器会创建相应数量的数据库连接，并将它们存储在连接容器中。在系统运行过程中，如果连接池中的连接数量不足，连接池管理器也会根据需要创建新的连接。而连接的销毁操作则是在连接不再需要时进行。比如，当连接池中的空闲连接数量超过了配置的最大空闲连接数，连接池管理器会选择一些

空闲时间较长的连接进行销毁，以释放资源；当系统关闭时，连接池管理器会销毁连接池中的所有连接，确保资源的正确释放。

四、设计要点：打造高性能连接池

4.1连接池大小的抉择

- 最小连接数Min Pool Size:默认为0。是连接池一直保持的数据库连接,所以如果应用程序对数据库连接的使用量不大,将会有大量的数据库连接资源被浪费
- 最大连接数Max Pool Size:默认为100。：是连接池能申请的最大连接数,如果数据库连接请求超过次数,后面的数据库连接请求将被加入到等待队列中,这会影响以后的数据库操作
- 最大空闲时间获取连接超时时间Connection Timeout：连接请求等待超时时间。默认为15秒，单位为秒。
- 超时重试连接次数Pooling:是否启用连接池。http://ADO.NET默认是启用连接池的，因此，你需要手动设置Pooling=false来禁用连接池。

这里放一个我们项目druid的配置：

```
spring.datasource.druid.initial-size=50
spring.datasource.druid.min-idle=20
spring.datasource.druid.max-active=100
spring.datasource.druid.max-wait=60000
spring.datasource.druid.time-between-eviction-runs-millis=60000
spring.datasource.druid.min-evictable-idle-time-millis=300000
spring.datasource.druid.connection-init-sqls=SET NAMES utf8mb4 COLLATE utf8mb4_unicode_ci
spring.datasource.druid.validation-query=SELECT 1 FROM DUAL
spring.datasource.druid.test-while-idle=true
spring.datasource.druid.test-on-borrow=false
spring.datasource.druid.test-on-return=false
spring.datasource.druid.remove-abandoned=true
spring.datasource.druid.remove-abandoned-timeout=180
spring.datasource.druid.log-abandoned=true
spring.datasource.druid.pool-prepared-statements=true
spring.datasource.druid.max-pool-prepared-statement-per-connection-size=20
spring.datasource.druid.filters=stat,wall,log4j
spring.datasource.druid.filter.stat.merge-sql=true
spring.datasource.druid.filter.stat.slow-sql-millis=5000

spring.datasource.druid.web-stat-filter.enabled=true
spring.datasource.druid.web-stat-filter.exclusions=*.js,*.gif,*.jpg,*.png,*.css,*.ico

spring.datasource.druid.stat-view-servlet.allow=223.71.219.238/30,223.71.219.240/29,36.110.63.64/28,1.202.169.232/29
spring.datasource.druid.stat-view-servlet.enabled=true
spring.datasource.druid.stat-view-servlet.login-username=joy
spring.datasource.druid.stat-view-servlet.login-password=joyshebao2018
spring.datasource.druid.stat-view-servlet.reset-enable=false
```

公众号 · 深度Linux

连接池大小的设置是连接池设计中至关重要的一环，它直接关系到系统在高并发场景下的性能表现。如果连接池大小设置得过小，当大量并发请求涌入时，连接池中的连接很快就会被耗尽，后续的请求就不得不进入等待队列，等待其他连接被释放后才能获取到连接进行数据库操作。这就好比一家热门餐厅，只有少量的餐桌（连接），用餐高峰时（高并发场景），大量顾客（请求）只能在门口排队等待，导致顾客等待时间过长，体验感极差。在实际的电商系统中，可能会出现用户在促销活动期间下单时，由于连接池大小不足，长时间等待订单处理结果，甚至出现超时错误，这不仅影响用户购买的积极性，还可能导致订单丢失，给商家带来经济损失。

相反，如果连接池大小设置得过大，虽然可以满足大量并发请求的连接需求，但会消耗过多的系统资源。每个数据库连接都需要占用一定的内存、网络资源以及数据库服务器的资源。过多的连接会导致服务器内存紧张，增加内存管理的开销，甚至可能引发内存溢出等问题。同时，过多的连接还可能对数据库服务器造成过大的压力，影响数据库的性能和稳定性。例如，在一个企业级应用中，如果连接池大小设置不合理，过大的连接池会使数据库服务器忙于处理大量的连接请求，而无法高效地执行真正的业务查询，导致数据库响应变慢，整个系统的性能也随之下降。

为了确定合适的连接池大小，需要综合考虑多个因素。首先，要深入分析应用的并发需求。可以通过性能测试工具模拟不同并发场景下的请求量，观察系统的性能指标，如响应时间、吞吐量等，来确定系统在不同负载下所需的最佳连接数。同时，还需要了解数据库服务器的处理能力，包括服务器的硬件配置（如 CPU、内存、磁盘 I/O 等）以及数据库的并发处理能力。如果数据库服务器的硬件配置较低，或者数据库本身对并发连接的支持有限，那么即使设置了很大的连接池大小，也无法充分利用，反而会浪费资源。此外，还可以参考一些经验公式，如连接池大小 =（2 * CPU 核数） + 1，但这只是一个大致的参考，实际应用中还需要根据具体情况进行调整和优化。

4.2连接生命周期管理

连接生命周期管理是连接池设计中的另一个关键方面，它主要涉及到最小和最大连接数、连接空闲超时、连接回收等参数的设置，这些参数对于确保连接池的高效运行和资源的合理利用起着至关重要的作用。

最小连接数是连接池在空闲状态下始终保持的连接数量。设置合适的最小连接数可以避免在高并发请求突然到来时，因为需要临时创建大量连接而导致的性能延迟。例如，在一个实时监控系统中，可能会有持续的少量请求不断地查询数据库获取最新的监控数据。如果最小连接数设置得过小，每次请求都需要创建新的连接，这会增加连接建立的时间开销，导致监控数据的获取不及时。而设置较大的最小连接数，可以保证在请求到来时，有足够的空闲连接可供使用，提高系统的响应速度。但是，如果最小连接数设置过大，会导致在系统空闲时，仍然占用大量的数据库连接资源，浪费系统资源。

最大连接数则限制了连接池中可以同时存在的最大连接数量。它是防止系统资源被过度消耗的重要防线。当并发请求数量超过最大连接数时，新的请求将无法立即获取到连接，需要等待其他连接被释放。合理设置最大连接数可以避免因连接过多而导致数据库服务器资源耗尽。比如，在一个在线教育平台中，同时有大量学生进行课程学习、作业提交等操作，如果没有设置合理的最大连接数，过多的连接可能会使数据库服务器不堪重负，出现死机或崩溃的情况，影响整个平台的正常运行。

连接空闲超时是指连接在池中空闲的最长时间，超过这个时间，连接将被回收。这一参数的设置可以有效地避免连接长时间占用资源而不被使用的情况。在一个企业的办公自动化系统中，可能会有一些用户长时间登录系统，但实际操作并不频繁，其对应的数据库连接可能会长时间处于空闲状态。如果没有设置连接空闲超时，这些空闲连接会一直占用资源，导致连接池中的有效连接减少。通过设置合理的连接空闲超时，当连接空闲时间超过设定值时，连接池会自动回收这些连接，释放资源，供其他需要的请求使用。

连接回收机制是连接生命周期管理的重要组成部分。除了根据空闲超时回收连接外，还可以定期对连接池中的连接进行健康检查，对于那些已经失效的连接，如因为网络故障或数据库服务器重启导致连接断开的连接，及时进行回收和清理。这样可以保证连接池中的连接都是有效的，避免将无效连接分配给应用程序，从而提高系统的稳定性和可靠性。

4.3并发控制策略

在高并发场景下，连接池面临着严峻的并发访问挑战。多个线程同时请求获取和释放连接，可能会导致数据不一致、连接泄漏、资源竞争等问题。为了确保连接池的线程安全性和高效运行，需要采用有效的并发控制策略。

锁机制是实现并发控制的常用手段之一。在连接池中，可以使用互斥锁（如 Java 中的 synchronized 关键字或 ReentrantLock）来保证在同一时刻只有一个线程能够访问连接池的关键资源，如连接的获取、释放和连接池状态的修改等操作。当一个线程获取到锁后，其他线程必须等待锁的释放才能进行相应的操作。

例如，在一个多线程的 Web 应用中，当多个线程同时请求从连接池中获取连接时，通过锁机制可以确保每次只有一个线程能够成功获取连接，避免了多个线程同时操作连接池导致的混乱。然而，锁机制也存在一定的局限性，它会引入额外的开销，如线程上下文切换和锁的竞争，可能会降低系统的并发性能。当锁的竞争激烈时，会导致大量线程处于等待状态，增加线程的等待时间，降低系统的吞吐量。

为了减少锁的竞争，提高并发性能，可以采用一些更细粒度的并发控制策略。例如，使用线程安全的数据结构来管理连接池中的连接。像 Java 中的 ConcurrentLinkedQueue 就是一种线程安全的队列，它可以在多线程环境下高效地进行元素的添加和移除操作，无需使用锁来保证线程安全。在连接池中，可以使用这种数据结构来存储空闲连接，当线程请求获取连接时，可以直接从队列中获取，而不需要获取锁，从而提高并发性能。此外，还可以采用分段锁的策略，将连接池划分为多个段，每个段使用独立的锁进行控制。当不同线程请求不同段的连接时，不会发生锁的竞争，只有当线程请求同一分段的连接时才会竞争锁，这样可以大大减少锁的竞争范围，提高系统的并发处理能力。

除了锁机制和线程安全的数据结构，还可以采用信号量机制来控制并发访问。信号量可以限制同时访问某个资源的线程数量。在连接池中，可以设置一个信号量，其初始值为连接池的最大连接数。当一个线程请求获取连接时，首先尝试获取信号量，如果信号量的值大于 0，则表示有可用连接，线程可以获取连接并将信号量的值减 1；如果信号量的值为 0，则表示连接池已满，线程需要等待，直到有其他线程释放连接并归还信号量。这种机制可以有效地控制并发请求的数量，避免连接池被过度使用，同时也可以减少线程的等待时间，提高系统的性能。

五、优化策略：让连接池飞起来

5.1连接池参数调优实战

在实际应用中，不同的连接池框架有着各自独特的配置参数，深入理解这些参数的含义并进行合理优化，是提升连接池性能的关键。以 HikariCP 和 Druid 这两款广泛使用的连接池框架为例，我们来详细探讨它们的常见参数及优化方法。

HikariCP 作为一款高性能的连接池，其配置参数简洁而高效。其中，maximumPoolSize（最大连接数）决定了连接池中能够容纳的最大连接数量。在一个高并发的电商系统中，如果设置过小，如设置为 10，当同时有 50 个用户进行商品查询、下单等操作时，超过 10 个请求就需要等待连接释放，这会导致大量请求超时，用户体验极差。一般来说，对于中等规模的电商系统，根据服务器硬件配置和业务并发量，maximumPoolSize可以设置在 50 - 100 之间。而 minimumIdle（最小空闲连接数）表示连接池中保持的最小空闲连接数量。若设置为 1，在业务高峰时，可能会频繁创建和销毁连接，增加系统开销。通常建议设置为 10 - 20，以保证在业务量波动时，系统能够快速响应请求。

connectionTimeout（连接超时时间）用于设置从连接池获取连接的最大等待时间，默认值为 30000 毫秒（30 秒）。如果设置过短，如 5000 毫秒（5 秒），在高并发且连接池繁忙时，应用

程序可能因为短时间内获取不到连接而报错，影响业务正常进行。根据实际业务场景，可适当调整为 10000 - 20000 毫秒，以平衡等待时间和系统响应效率。

Druid 连接池功能丰富，除了基本的连接管理功能外，还提供了强大的监控和扩展功能。其maxActive（最大连接数）与 HikariCP 的maximumPoolSize类似，控制着连接池的最大连接数量。在一个企业级的 OA 系统中，若maxActive设置为 30，当多个部门同时进行文件审批、数据查询等操作时，可能会出现连接不足的情况。

根据 OA 系统的并发特点和用户规模，可将maxActive设置在 30 - 50 之间。minIdle（最小空闲连接数）同样影响着系统的响应速度和资源利用效率。若设置为 5，在系统空闲时，可能会保留过多的空闲连接，浪费资源；若设置为 1，在业务量突然增加时，可能无法及时提供足够的连接。一般可根据业务的平均负载情况，设置为 10 - 15。maxWait（最长等待时间）表示获取连接时的最大等待时间，单位为毫秒。若设置为 10000 毫秒（10 秒），当连接池满且请求频繁时，部分请求可能会因为等待时间过长而失败。可根据业务对响应时间的要求，调整为 15000 - 20000 毫秒，确保请求有足够的等待时间获取连接。

5.2连接有效性检测与维护

在高并发场景下，确保连接池中的连接始终有效是至关重要的，这直接关系到系统的稳定性和可靠性。定期检测连接有效性能够及时发现并处理失效连接，避免将无效连接分配给应用程序，从而防止因连接问题导致的业务故障。

心跳检测是一种常见的连接有效性检测方法，它通过定期向数据库发送一个简单的心跳包，如一个轻量级的 SQL 查询（如SELECT 1）或特定的数据库命令（如 MySQL 的PING命令），来判断连接是否仍然可用。如果数据库能够正常响应心跳包，则说明连接有效；若在规定时间内未收到响应，则认为连接已失效。以一个在线游戏服务器为例，每 5 秒向数据库发送一次心跳包，若连续 3 次未收到响应，则判定连接失效，将该连接从连接池中移除，并重新创建新的连接。这种方式能够快速检测出连接的异常情况，保证连接池中的连接始终处于可用状态。

SQL 查询检测则是通过执行一些特定的 SQL 查询来验证连接的有效性。这些查询通常选择那些对数据库资源消耗较小，但又能准确反映数据库状态的语句，如查询一个固定的系统表或执行一个简单的计数查询（如SELECT COUNT(*) FROM some_table）。在一个金融交易系统中，每隔 10 秒执行一次 SQL 查询检测连接有效性，若查询失败或返回异常结果，则认为连接失效，立即进行处理。通过这种方式，可以确保连接池中的连接能够正常执行数据库操作，避免因连接问题导致的交易错误或数据不一致。

除了检测连接有效性，及时移除失效连接也是维护连接池健康的关键环节。当检测到连接失效时，连接池应立即将其从连接容器中移除，并释放相关的资源，如关闭网络连接、释放数据库资源等。同时，为了保证连接池的性能，还可以在移除失效连接后，根据配置策略及时创建新的连接，以确保连接池中的连接数量始终满足业务需求。例如，在一个大型电商平台的连接池中，当检测到某个连接失效后，会立即将其移除，并根据当前连接池的状态和业务负载情况，决定是否创建新的连接。如果当前连接池中的连接数量低于最小空闲连接数，则会立即创建新的连接，以保证系统的正常运行。

5.3事务管理与连接池的协同

事务管理在高并发场景下对于保证数据的一致性和完整性起着不可或缺的作用，而连接池与事务管理的协同工作则是确保系统高效稳定运行的关键因素。

在高并发环境中，多个事务可能同时对数据库进行操作，如果事务管理不当，可能会导致数据不一致、脏读、幻读等问题。例如，在一个银行转账系统中，当用户 A 向用户 B 转账时，涉及到两个账户的资金变动，这两个操作必须作为一个事务来处理，要么都成功，要么都失败。如果在高并发情况下，事务管理出现问题，可能会导致 A 账户的钱扣除了，但 B 账户却没有收到钱，从而造成数据不一致。因此，正确的事务管理能够保证在高并发场景下，数据库操作的原子性、一致性、隔离性和持久性。

在连接池中管理事务的生命周期需要遵循一定的原则和方法。首先，事务的开始和结束应该与连接的获取和释放紧密配合。当应用程序开始一个事务时，应从连接池中获取一个连接，并在整个事务过程中保持对该连接的使用。例如，在一个电商订单处理系统中，当用户下单时，开始一个事务，从连接池中获取一个连接，然后在这个连接上执行插入订单数据、更新库存等操作。在事务执行过程中，应确保所有相关的数据库操作都在同一个连接上进行，以保证事务的原子性。

其次，要避免事务长时间占用连接而影响整体性能。如果一个事务执行时间过长，会导致连接长时间被占用，其他请求无法及时获取到连接，从而影响系统的并发处理能力。在一个在线预订系统中，如果某个事务因为复杂的业务逻辑或数据库操作而执行了几分钟，期间该连接一直被占用，那么在这段时间内，其他用户的预订请求可能会因为无法获取连接而等待，导致系统响应变慢，用户体验下降。为了避免这种情况，可以对事务的执行时间进行监控和限制，当事务执行时间超过一定阈值时，及时进行处理，如回滚事务或优化业务逻辑，以尽快释放连接。

此外，还需要注意事务的隔离级别设置。不同的隔离级别会对数据的一致性和并发性能产生不同的影响。在高并发场景下，应根据业务需求合理选择事务隔离级别。例如，对于一些对数据一致性要求较高的业务，如金融交易、库存管理等，可以选择较高的隔离级别，如可串行化隔离级别，以确保数据的准确性；而对于一些对并发性能要求较高，对数据一致性要求相对较低的业务，如商品浏览、统计分析等，可以选择较低的隔离级别，如读已提交隔离级别，以提高系统的并发处理能力。

六、实战案例：看大厂如何做

6.1案例背景与挑战

某知名互联网电商公司，业务覆盖全球多个地区，拥有数亿用户。在每年的 “双 11”“618” 等大型促销活动期间，平台会迎来流量的爆发式增长，并发请求量瞬间可达每秒数十万甚至数百万次。在如此高并发的业务场景下，连接池成为保障系统性能的关键环节。

在早期，该公司使用的是一款开源的连接池框架，并采用了默认的连接池配置。随着业务的不断发展和用户量的急剧增加，在高并发场景下，系统逐渐暴露出性能瓶颈和挑战。连接获取时间大幅增加，平均响应时间从正常情况下的几十毫秒延长到了几百毫秒甚至秒级，这导致大量用户在页面上长时间等待商品查询结果、订单提交反馈等，用户体验急剧下降。同时，由于连接池的配置不合理，频繁的连接创建和销毁操作使得系统资源消耗巨大，服务器的 CPU 和内存使用率长时间处于高位，甚至出现了因资源耗尽而导致部分服务不可用的情况。

6.2优化方案与实施过程

针对这些问题，该公司的技术团队进行了深入的分析和研究，制定了一系列全面的优化方案并付诸实施。

在参数调整方面，技术团队对连接池的各项参数进行了细致的优化。他们根据服务器的硬件配置和业务的并发特点，对最大连接数、最小空闲连接数、连接超时时间等关键参数进行了重新设置。经过多次性能测试和调优，将最大连接数从原来的 50 调整为 200，最小空闲连接数从 10 调整为 50，连接超时时间从 30 秒缩短为 15 秒。这样的调整使得连接池能够更好地适应高并发场景下的业务需求，既保证了有足够的连接可供使用，又避免了过多的空闲连接占用资源。

连接池框架更换也是优化的重要举措之一。技术团队经过对多种连接池框架的性能测试和对比分析，最终选择了性能更优的 HikariCP 连接池框架替换原来的框架。HikariCP 以其高效的连接管理和低延迟的特性，在高并发场景下表现出色。在更换框架的过程中，技术团队对系统的相关代码进行了全面的修改和适配，确保新框架能够与现有系统无缝集成。

在架构改进方面，为了进一步提高系统的性能和稳定性，技术团队引入了读写分离和负载均衡技术。他们将数据库的读操作和写操作分别分配到不同的数据库实例上，通过负载均衡器将并发请求均匀地分发到各个数据库实例上，从而减轻单个数据库的压力。同时，采用了连接池集群技术，将多个连接池组合成一个集群，实现连接资源的共享和动态分配。在一个大型促销活动中，当某个连接池的负载过高时，请求可以自动被分配到其他负载较低的连接池中，确保系统能够稳定地处理高并发请求。

6.3优化效果与经验总结

经过一系列的优化措施实施后，该电商平台的系统性能得到了显著提升。在后续的 “双 11” 活动中，系统的平均响应时间从优化前的几百毫秒缩短到了 50 毫秒以内，吞吐量增加了 3 倍以上，能够轻松应对每秒数百万次的并发请求。用户在页面上的操作响应迅速，订单提交成功率大幅提高，极大地提升了用户体验。

从这个案例中，我们可以总结出许多宝贵的经验。深入了解业务需求和系统性能瓶颈是优化的基础，只有准确把握问题所在，才能制定出针对性的优化方案。合理的参数调整和优秀的连接池框架选择对于提升连接池性能至关重要，需要通过充分的性能测试和对比分析来确定最佳配置。此外，架构层面的改进能够从根本上提高系统的并发处理能力和稳定性，引入先进的技术和架构理念是应对高并发挑战的有效途径。

七、连接池需要注意的点

7.1并发问题

为了使连接管理服务具有最大的通用性，必须考虑多线程环境，即并发问题。这个问题相对比较好解决，因为各个语言自身提供了对并发管理的支持像java,c#等等，使用synchronized(java)lock(C#)关键字即可确保线程是同步的。使用方法可以参考，相关文献。

7.2事务处理

我们知道，事务具有原子性，此时要求对数据库的操作符合“ALL-OR-NOTHING”原则,即对于一组SQL语句要么全做，要么全不做。

我们知道当2个线程共用一个连接Connection对象，而且各自都有自己的事务要处理时候，对于连接池是一个很头疼的问题，因为即使Connection类提供了相应的事务支持，可是我们仍然不能确定那个数据库操作是对应那个事务的，这是由于我们有 2 个线程都在进行事务操作而引起的。为此我们可以使用每一个事务独占一个连接来实现，虽然这种方法有点浪费连接池资源但是可以大大降低事务管理的复杂性。

7.3连接池的分配与释放

连接池的分配与释放，对系统的性能有很大的影响。合理的分配与释放，可以提高连接的复用度，从而降低建立新连接的开销，同时还可以加快用户的访问速度。

对于连接的管理可使用一个List。即把已经创建的连接都放入List中去统一管理。每当用户请求一个连接时，系统检查这个List中有没有可以分配的连接。如果有就把那个最合适的连接分配给他（如何能找到最合适的连接文章将在关键议题中指出）；如果没有就抛出一个异常给用户，List中连接是否可以被分配由一个线程来专门管理捎后我会介绍这个线程的具体实现。

7.4连接池的配置与维护

连接池中到底应该放置多少连接，才能使系统的性能最佳？系统可采取设置最小连接数（minConnection）和最大连接数（maxConnection）等参数来控制连接池中的连接。比方说，最小连接数是系统启动时连接池所创建的连接数。如果创建过多，则系统启动就慢，但创建后系统的响应速度会很快；如果创建过少，则系统启动的很快，响应起来却慢。这样，可以在开发时，设置较小的最小连接数，开发起来会快，而在系统实际使用时设置较大的，因为这样对访问客户来说速度会快些。最大连接数是连接池中允许连接的最大数目，具体设置多少，要看系统的访问量，可通过软件需求上得到。

如何确保连接池中的最小连接数呢？有动态和静态两种策略。动态即每隔一定时间就对连接池进行检测，如果发现连接数量小于最小连接数，则补充相应数量的新连接,以保证连接池的正常运转。静态是发现空闲连接不够时再去检查。

Linux C/C++开发 182　性能优化 47　项目实战 96

Linux C/C++开发 · 目录

上一篇
用Qt打造高效用户界面：让你的应用更出色!

下一篇
从0到1学CMake：开启高效跨平台构建之旅