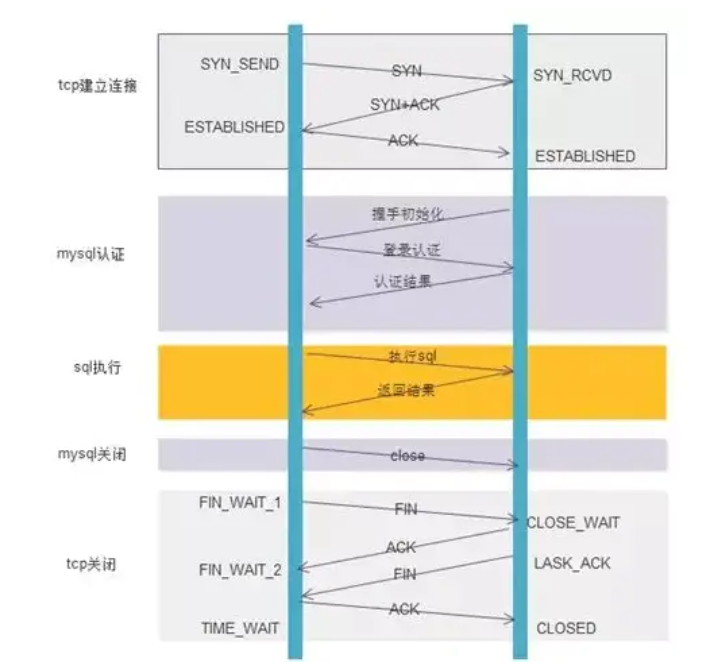
# 背景

## 传统连接机制

**从网络的角度分析**

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）

早期数据库连接：



**原理：**

一般来说，Java应用程序访问数据库的过程是：

①装载数据库驱动程序；

②通过jdbc建立数据库连接；

③访问数据库，执行sql语句；

④断开数据库连接。

**代码：**

       Public void FindAllUsers(){  
              //1、装载sqlserver驱动对象  
              DriverManager.registerDriver(new SQLServerDriver());               
              //2、通过JDBC建立数据库连接  
              Connection con =DriverManager.getConnection("jdbc:sqlserver://192.168.2.6:1433;DatabaseName=customer", "sa", "123");              
              //3、创建状态  
              Statement state =con.createStatement();             
              //4、查询数据库并返回结果  
              ResultSet result =state.executeQuery("select \* from users");             
              //5、输出查询结果  
              while(result.next()){  
                     System.out.println(result.getString("email"));  
              }              
              //6、断开数据库连接  
              result.close();  
              state.close();  
              con.close();  
        }

**分析：**

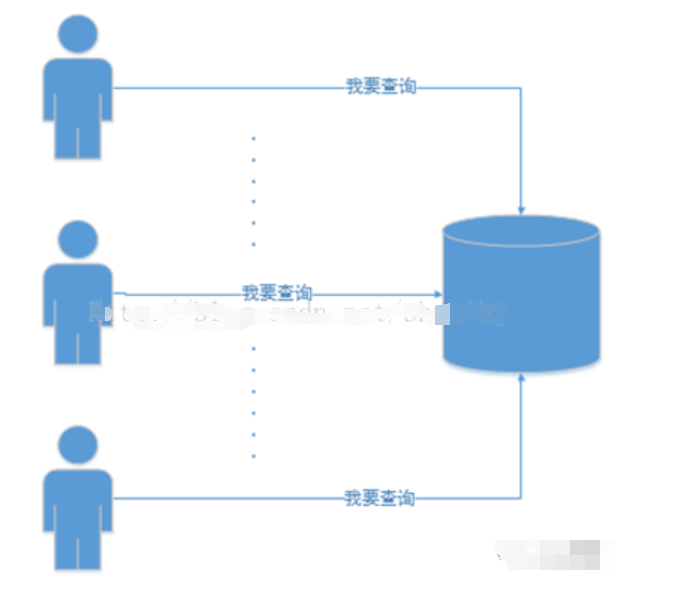
程序开发过程中，存在很多问题：

首先，每一次web请求都要建立一次数据库连接。建立连接是一个费时的活动，每次都得花费0.05s～1s的时间，而且系统还要分配内存资源。这个时间对于一次或几次数据库操作，或许感觉不出系统有多大的开销。可是对于现在的web应用，尤其是大型电子商务网站，同时有几百人甚至几千人在线是很正常的事。

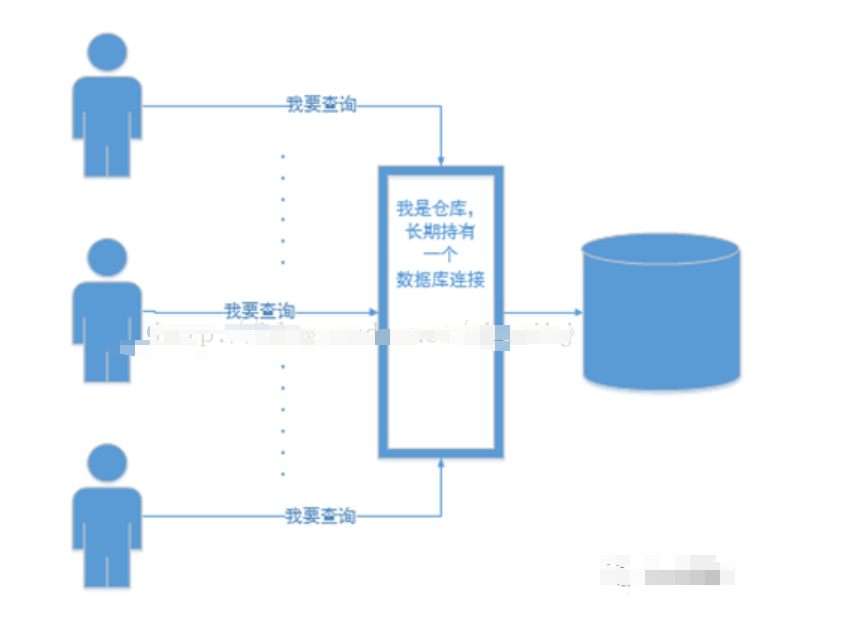
在这种情况下，频繁的进行数据库连接操作势必占用很多的系统资源，网站的响应速度必定下降，严重的甚至会造成服务器的崩溃。不是危言耸听，这就是制约某些电子商务网站发展的技术瓶颈问题。

其次，对于每一次数据库连接，使用完后都得断开。否则，如果程序出现异常而未能关闭，将会导致数据库系统中的内存泄漏，最终将不得不重启数据库。 还有，这种开发不能控制被创建的连接对象数，系统资源会被毫无顾及的分配出去，如连接过多，也可能导致内存泄漏，服务器崩溃。

上述的用户查询案例，如果同时有1000人访问，就会不断的有数据库连接、断开操作：



通过上面的分析，我们可以看出来，“数据库连接”是一种稀缺的资源，为了保障网站的正常使用，应该对其进行妥善管理。其实我们查询完数据库后，如果不关闭连接，而是暂时存放起来，当别人使用时，把这个连接给他们使用。就避免了一次建立数据库连接和断开的操作时间消耗。原理如下：



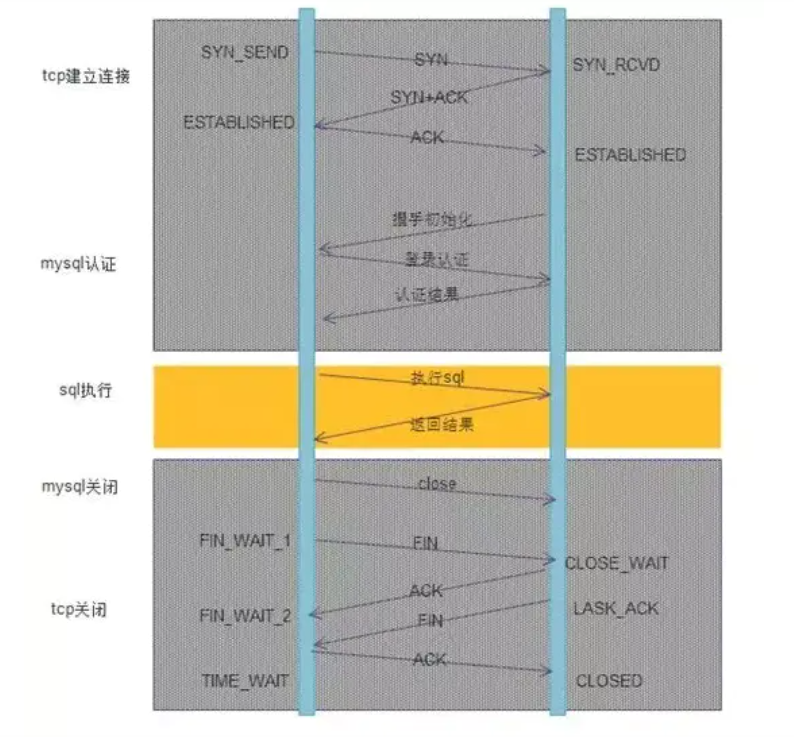
## 数据库连接池

技术演进出来的数据库连接池：

由上面的分析可以看出，问题的根源就在于对数据库连接资源的低效管理。我们知道，对于共享资源，有一个很著名的设计模式：资源池（resource pool）。该模式正是为了解决资源的频繁分配、释放所造成的问题。

为解决上述问题，可以采用数据库连接池技术。

数据库连接池的基本思想就是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。我们可以通过设定连接池最大连接数来防止系统无尽的与数据库连接。更为重要的是我们可以通过连接池的管理机制监视数据库的连接的数量、使用情况，为系统开发、测试及性能调整提供依据。



**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

1、我们自己尝试开发一个连接池，来为上面的查询业务提供数据库连接服务：

① 编写class实现DataSource接口

② 在class构造器一次性创建10个连接，将连接保存LinkedList中

③ 实现getConnection从LinkedList中返回一个连接

④ 提供将连接放回连接池中方法

连接池代码

    public class MyDataSource implements DataSource {  
              //链表 --- 实现栈结构  
              privateLinkedList<Connection> dataSources = new LinkedList<Connection>();  
              //初始化连接数量  
              publicMyDataSource() {  
                     //一次性创建10个连接  
                     for(int i = 0; i < 10; i++) {  
                            try {  
                               //1、装载sqlserver驱动对象  
                               DriverManager.registerDriver(new SQLServerDriver());  
                               //2、通过JDBC建立数据库连接  
                               Connection con =DriverManager.getConnection(  
                                  "jdbc:sqlserver://192.168.2.6:1433;DatabaseName=customer", "sa", "123");  
                               //3、将连接加入连接池中  
                               dataSources.add(con);  
                            } catch (Exception e) {  
                               e.printStackTrace();  
                            }  
                     }  
              }  
  
              @Override  
              publicConnection getConnection() throws SQLException {  
                     //取出连接池中一个连接  
                     finalConnection conn = dataSources.removeFirst();

// 删除第一个连接返回  
                     returnconn;  
              }  
  
              //将连接放回连接池  
              publicvoid releaseConnection(Connection conn) {  
                     dataSources.add(conn);  
                     }  
       }

2、使用连接池重构我们的用户查询函数

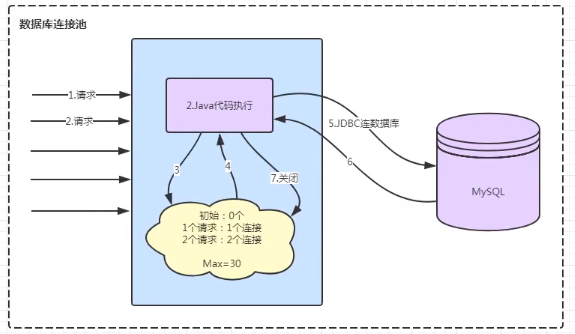
//查询所有用户  
       Public void FindAllUsers(){  
              //1、使用连接池建立数据库连接  
              MyDataSource dataSource = new MyDataSource();  
              Connection conn =dataSource.getConnection();          
              //2、创建状态  
              Statement state =con.createStatement();             
              //3、查询数据库并返回结果  
              ResultSet result =state.executeQuery("select \* from users");             
              //4、输出查询结果  
              while(result.next()){  
                     System.out.println(result.getString("email"));  
              }              
              //5、断开数据库连接  
              result.close();  
              state.close();  
              //6、归还数据库连接给连接池  
              dataSource.releaseConnection(conn);  
        }

这就是数据库连接池的原理，它大大提供了数据库连接的利用率，减小了内存吞吐的开销。我们在开发过程中，就不需要再关心数据库连接的问题，自然有数据库连接池帮助我们处理。当然，线程池还需要考虑并发、事务、分配释放以及维护等问题。

# 概述

数据库连接是一种昂贵的资源，创建数据库连接是一个耗时的操作，在多线程并发情况下尤为突出，对数据库连接的高效管理影响到程序的性能指标，数据库连接池正是针对该问题提出的。

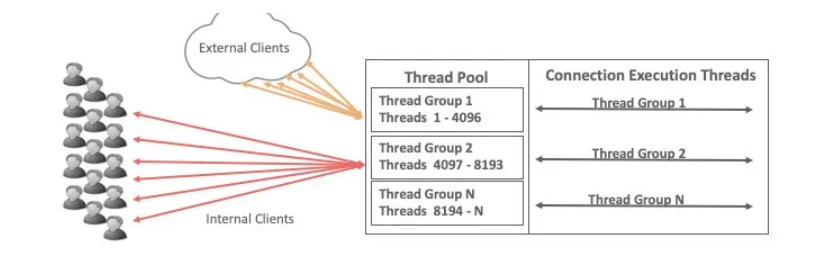
数据库连接池负责分配、管理和释放数据库连接，它允许应用程序重复使用一个现有的数据库连接，而不是再重新建立一个新连接，利用数据库连接池将明显提高对数据库操作的性能。



## 线程池与连接池

MySQL默认的连接控制方式采用的是每个连接使用一个线程执行客户端的请求。MySQL的线程池是包含在企业版里面的服务器插件。使用线程池的目的是为了改善大量并发连接所带来的性能下降。在大量并发连接的工作负载下，使用线程池可以解决无法利用CPU缓存、上下文切换开销过大以及资源争用等问题。

线程池功能由插件库文件、服务器系统变量及Performance Schema里面的检测点组成。



## 作用

1. 资源重用

由于数据库连接得到重用，避免了频繁创建、释放连接引起的大量性能开销。在减少系统性能消耗的基础上，增进了系统环境的平稳性（减少内存碎片以及数据库临时进程、线程的数量）。

1. 更快的访问速度

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内备用。此时，连接池的初始化操作均已完成。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的响应时间。

1. 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，实现数据库连接技术。

1. 统一的连接管理，避免数据库连接泄露

在较为完备的数据库连接池实现中，可根据预先的连接占用超时设定，强制收回被占用的连接，从而避免了常规数据库连接操作中可能出现的资源泄露。

线程池好处：重复利用，提高响应效率，统一管理。

## 优点

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

## 计算公式

<https://mp.weixin.qq.com/s/2JpPMauLq5aPkJTEMm6NuA>

下面的公式是由PostgreSQL提供的，不过我们认为可以广泛地应用于大多数数据库产品。你应该模拟预期的访问量，并从这一公式开始测试你的应用，寻找最合适的连接数值。

连接数 = ((核心数 \* 2) + 有效磁盘数)

核心数不应包含超线程(hyper thread)，即使打开了hyperthreading也是。如果活跃数据全部被缓存了，那么有效磁盘数是0，随着缓存命中率的下降，有效磁盘数逐渐趋近于实际的磁盘数。这一公式作用于SSD时的效果如何尚未有分析。按这个公式，你的4核i7数据库服务器的连接池大小应该为((4 \* 2) + 1) = 9。取个整就算是是10吧。是不是觉得太小了？跑个性能测试试一下，我们保证它能轻松搞定3000用户以6000TPS的速率并发执行简单查询的场景。如果连接池大小超过10，你会看到响应时长开始增加，TPS开始下降。

注：  
 这一公式其实不仅适用于数据库连接池的计算，大部分涉及计算和I/O的程序，线程数的设置都可以参考这一公式。

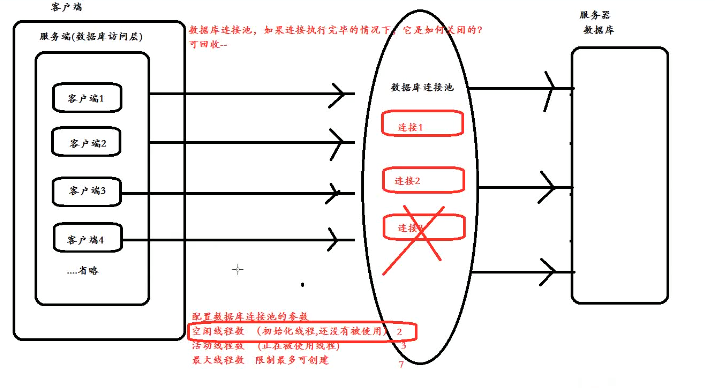
如果你有10000个并发用户，设置一个10000的连接池显然不合理，即使100也太多了。你需要一个10来个连接的小连接池，然后让剩下的业务线程都在队列里等待。**连接池中的连接数量应该等于你的数据库能够有效同时进行的查询任务数（通常不会高于2\*CPU核心数）**。我们经常见到一些小规模的web应用，应付着大约十来个的并发用户，却使用着一个100连接数的连接池。这会对你的数据库造成极其不必要的负担。

## 连接池大小

连接池的大小最终与系统特性相关。

比如一个混合了长事务和短事务的系统，通常是任何连接池都难以进行调优的。最好的办法是创建两个连接池，一个服务于长事务，一个服务于短事务。再例如一个系统执行一个任务队列，只允许一定数量的任务同时执行，此时并发任务数应该去适应连接池连接数，而不是反过来。

# 原理



数据库连接池实现原理（实现代码角度）：

1. 定义核心参数：

空闲线程（容器）：没有被使用的连接

活动县城：正在使用的连接

1. 初始化线程池（初始化空闲线程）
2. 调用getConnection获取连接
3. 调用releaseConnection释放连接：

获取activeConnection集合连接，转移到freeConnection集合中

## 考虑问题

### 并发问题

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java语言自身提供了对并发管理的支持，使用synchronized关键字即可确保线程是同步的。使用方法为直接在类方法前面加上synchronized关键字，如：

public synchronized connection getconnection()

注：在Linux中使用条件变量和互斥锁实现。

### 多数据库服务器和多用户

2、多数据库服务器和多用户

对于大型的企业级应用，常常需要同时连接不同的数据库（如连接oracle和sybase）。如何连接不同的数据库呢？

我们采用的策略是：设计一个符合单例模式的连接池管理类，在连接池管理类的唯一实例被创建时读取一个资源文件，其中资源文件中存放着多个数据库的url地址等信息。根据资源文件提供的信息，创建多个连接池类的实例，每一个实例都是一个特定数据库的连接池。连接池管理类实例为每个连接池实例取一个名字，通过不同的名字来管理不同的连接池。

对于同一个数据库有多个用户使用不同的名称和密码访问的情况，也可以通过资源文件处理，即在资源文件中设置多个具有相同url地址，但具有不同用户名和密码的数据库连接信息。

### 事务处理

3、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-all-nothing”原则即对于一组sql语句要么全做，要么全不做。

在java语言中，connection类本身提供了对事务的支持，可以通过设置connection的autocommit属性为false 然后显式的调用commit或rollback方法来实现。但要高效的进行connection复用，就必须提供相应的事务支持机制。可采用每一个事务独占一个连接来实现，这种方法可以大大降低事务管理的复杂性。

### 连接池的分配与释放

4、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空闲池。即把已经创建但尚未分配出去的连接按创建时间存放到一个空闲池中。每当用户请求一个连接时，系统首先检查空闲池内有没有空闲连接。如果有就把建立时间最长（通过容器的顺序存放实现）的那个连接分配给他（实际是先做连接是否有效的判断，如果可用就分配给用户，如不可用就把这个连接从空闲池删掉，重新检测空闲池是否还有连接）；如果没有则检查当前所开连接池是否达到连接池所允许的最大连接数（maxconn）如果没有达到，就新建一个连接，如果已经达到，就等待一定的时间（timeout）。

如果在等待的时间内有连接被释放出来就可以把这个连接分配给等待的用户，如果等待时间超过预定时间timeout 则返回空值（null）。系统对已经分配出去正在使用的连接只做计数，当使用完后再返还给空闲池。对于空闲连接的状态，可开辟专门的线程定时检测，这样会花费一定的系统开销，但可以保证较快的响应速度。也可采取不开辟专门线程，只是在分配前检测的方法。

### 连接池的配置与维护

5、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（minconn）和最大连接数（maxconn）来控制连接池中的连接。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，可通过反复测试，找到最佳点。

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。

## 实现

可以采用wait和notifyAll方式实现，也可以通过阻塞队列实现。

注：可以采用代理模式实现数据库连接池。

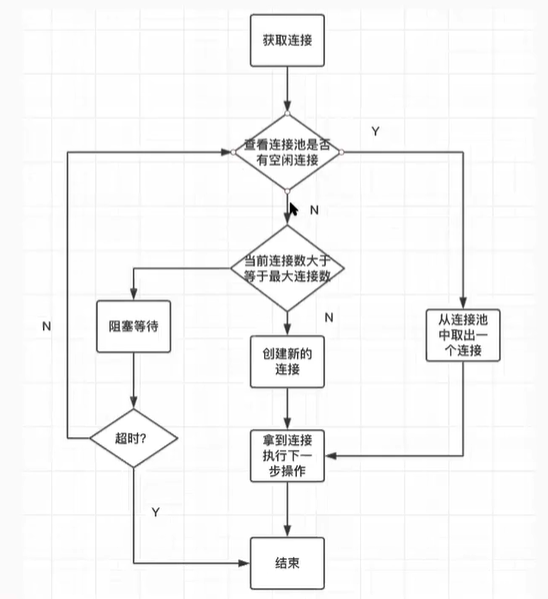
### 初始化连接池

配置数据库连接池的参数：

1. 空闲线程数（初始化线程，还没有被使用）
2. 活动线程数（正在被使用线程）
3. 最大线程数（限制最多可创建）



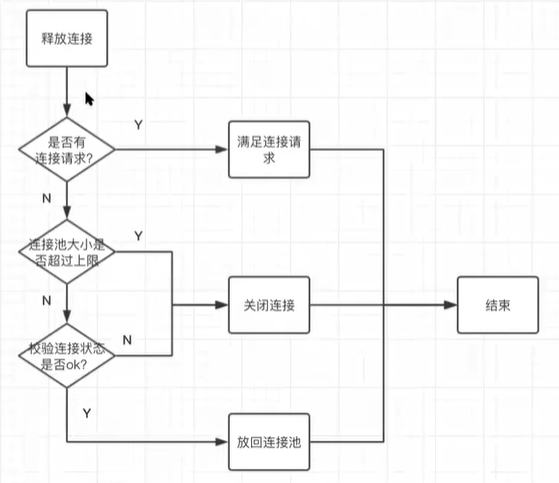
### 获取连接



线程池的建立：

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。

### 释放连接/线程池管理



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。

### 连接池关闭

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，该过程正好与创建相反。

## 参数

使用连接池时，要配置一下参数：

### 最小连接数

最小连接数：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

最大连接数：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

### 超时重试连接次数

# 技术方案

数据库连接池属于一种池化技术。

池化技术：**HTTP访问（httpclient）、redis访问（redispool）、线程（线程池）**。

## 概述

区分一个数据库连接池是属于第一代产品还是代二代产品有一个最重要的特征就是看它在架构和设计时采用的线程模型，因为这直接影响的是并发环境下存取数据库连接的性能。

一般来讲采用单线程同步的架构设计都属于第一代连接池，二采用多线程异步架构的则属于第二代。比较有代表性的就是Apache Commons DBCP，在1.x版本中，一直延续着单线程设计模式，到2.x才采用多线程模型。

用版本发布时间来辨别区分两代产品，则一个偷懒的好方法。以下是这些常见数据库连接池最新版本的发布时间：



从表中可以看出，C3P0已经很久没有更新了。DBCP更新速度很慢，基本处于不活跃状态，而Druid和HikariCP处于活跃状态的更新中，这就是我们说的二代产品了。

二代产品对一代产品的超越是颠覆性的，除了一些“历史原因”，你很难再找到第二条理由说服自己不选择二代产品，但任何成功都不是偶然的，二代产品的成功很大程度上得益于前代产品们打下的基础，站在巨人的肩膀上，新一代的连接池的设计师们将这一项“工具化”的产品，推向了极致。其中，最具代表性的两款产品是：

HikariCP

Druid

## 连接池

为了解决资源的频繁分配、释放所造成的问题，可以采用数据库连接池技术。数据库连接池的基本思想是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，只需要从“缓冲池”中取出一个，使用完毕之后再放回去。我们可以通过设定连接池最大连接数来防止系统无尽的与数据库连接。更为重要的是我们可以通过连接池的管理机制监视数据库的连接数量、使用情况，为系统开发、测试以及性能调优提供依据。

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。对数据库连接的管理能显著影响到整个应用程序的伸缩性和健壮性，影响到程序的性能指标。数据库连接池正式针对这个问题提出来的。数据库连接池负责分配、管理和释放数据库连接，它允许应用程序重复使用一个现有的数据库连接，而不再是重新建立一个；释放空闲时间超过最大空闲时间的数据库连接来避免因为没有释放数据库连接而引起的数据库资源泄漏。这项技术能够明显提高对数据库操作的性能。

目前市场上有很多开源的Java数据连接池，广泛使用的主要有C3P0、DBCP、BoneCP、Proxool等。虽然它们在与数据库建链和连接释放上的性能差别很大（十分之一毫秒到数毫秒不等），但是在业务访问数据库的整个流程中，性能瓶颈从来都是在连接池上，实际进行带业务的测试性能在不同连接池之间的性能差异不会超过百分之三。因此我们在数据库连接池选型的时候，更多应该关注它们的稳定性和易用性。

### JDBC Template

参考：

<https://juejin.cn/post/6844903602939494414>

JDBC中的MySQL数据库连接Connection和Statement、ResultSet使用完毕之后一定要关闭，否则会占用大量内存资源，导致内存溢出。所以动态创建的连接要及时释放。

JDBC连接池提供了JDBC连接定义和数目有限的连接，如果数量不够，就需要长时间的等待。不正常关闭JDBC连接会导致等待回收无效的JDBC连接。只有正常的关闭和释放JDBC连接，JDBC资源才可以被快速的重用使性能得到改善。

失败的关闭和释放JDBC连接可能导致其他用户的连接经历长时间的等待。虽然超时的JDBC连接会被websphere application server退回而被回收，但是必须等待这种情形发生。使用完JDBC资源后关闭它们，还可以显式关闭JDBC ResultSet。如果没有显式关闭语句，则在完成了相关语句之后会释放ResultSets。

所以请确保你构建的代码在所有情况下，甚至在异常和错误条件下，都能关闭和释放JDBC资源。下面的代码显示了JDBC资源的获得和使用都封装在“Try-Catch-Finally”结构中，其中，在finally子句中处理JDBC资源的关闭，使所有情况下关闭都将发生。该方法需要在所有的finally块中被使用：

public static void close(AutoCloseable close) {

if (null != close) {

try {

close.close();

}catch (SQLException e) {

E.printStackTrace();

}

}

}

### C3P0

C3P0在很长一段时间内，它一直是Java领域内数据库连接池的代名词，当年盛极一时的Hibernate都将其作为内置的数据库连接池，可以业内对它的稳定性还是认可的。

C3P0功能简单易用，可以设置最大和最小连接，稳定性好这是它的优点，但是性能上的缺点却让它彻底被打入冷宫，不提供连接池监控。C3P0的性能很差，差到即便是同时代的产品相比它也是垫底的，更不用和Druid、HikariCP等相比了。

正常来讲，有问题很正常，改就是了，但c3p0最致命的问题就是架构设计过于复杂，让重构变成了一项不可能完成的任务。随着国内互联网大潮的涌起，性能有硬伤的c3p0彻底的退出了历史舞台。

步骤：

1. 导入jar包：c3p0-0.9.5.2.jar+mchange-commons-java-0.2.12.jar
2. 定义配置文件：

名称：c3p0.properties、c3p0-config.xml

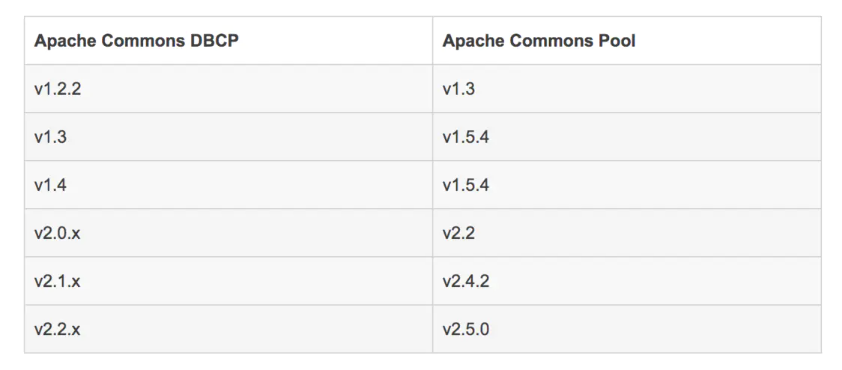
路径：直接将文件放在src目录下即可

1. 创建核心对象：数据库连接池对象ComboPooledDataSource
2. 获取连接：getConnection

### DBCP

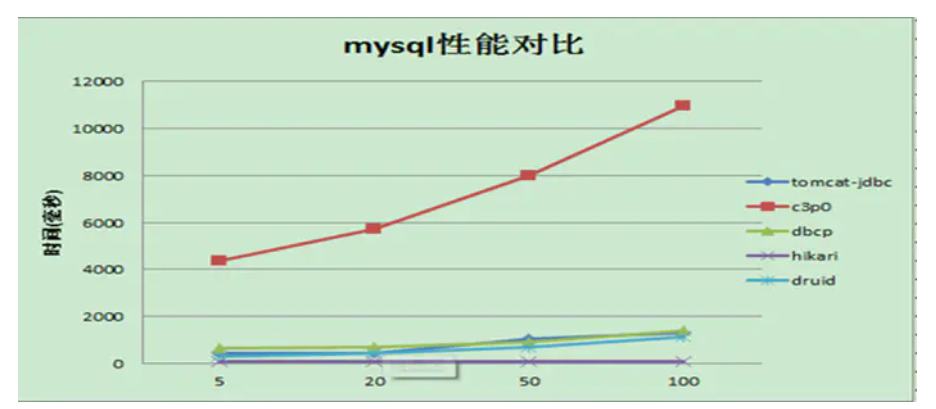
DBCP（DataBase Connection Pool）属于Apache顶级项目Commons中的核心子项目（最早在Jakarta Commons里就有）,在Apache的生态圈中的影响里十分广泛，比如最为大家所熟知的Tomcat就在内部集成了DBCP，实现JPA规范的OpenJPA，也是默认集成DBCP的。

但DBCP并不是独立实现连接池功能的，它内部依赖于Commons中的另一个子项目Pool，连接池最核心的“池”，就是由Pool组件提供的，因此，DBCP的性能实际上就是Pool的性能，DBCP和Pool的依赖关系如下表：



可以看到，因为核心功能依赖于Pool，所以DBCP本身只能做小版本的更新，真正大版本的更迭则完全依托于pool。有很长一段时间，pool都还是停留在1.x版本，这直接导致DBCP也更新乏力。

很多依赖DBCP的应用在遇到性能瓶颈之后，别无选择，只能将其替换掉，DBCP忠实的拥趸tomcat就在其tomcat 7.0版本中，自己重新设计开发出了一套连接池（Tomcat JDBC Pool）。好在，在2013年事情终于迎来转机，13年9月Commons-Pool 2.0版本发布，14年2月份，DBCP也终于迎来了自己的2.0版本，基于新的线程模型全新设计的“池”让DBCP重焕青春，虽然和新一代的连接池相比仍有一定差距，但差距并不大，DBCP2.x版本已经稳稳达到了和新一代产品同级别的性能指标（见下图）。



DBCP终于靠Pool咸鱼翻身，打了一个漂亮的翻身仗，但长时间的等待已经完全消磨了用户的耐心，与新一代的产品项目相比，DBCP没有任何优势，试问，谁会在有选择的前提下，去选择那个并不优秀的呢？也许，现在还选择DBCP2的唯一理由，就是情怀吧。

DBCP可以设置最大和最小连接，连接等待时间等，基本的功能都有，此连接池的持续运行的稳定性还可以，不过速度稍慢，在大并发的压力下稳定性能有所下降，此外不提供连接池监控。

### Proxool

这是一个Java SQL Driver驱动程序，提供了对你选择的其他类型的驱动程序的连接池封装。可以非常简单的移植到现存的代码中。完全可配置，快速，成熟，健壮。可以透明地为你现存的JDBC驱动程序增加连接池功能。

连接池可以设置最大和最小连接，连接等待时间等，基本功能都有，连接池的持续运行的稳定性有一定问题，有一个优势：连接池监控。

注：由于稳定性有一定问题，不推荐使用。

### Tomcat JDBC Pool

### BoneCP

BoneCP是一个快速，开源的数据库连接池，比C3P0/DBCP连接池快25倍。这是一种新的数据连接技术，以其效率高、速度快著称，连接池可以设置最大和最小连接，连接等待时间等，基本功能都有。缺点是，引入的包比较多，管理比较麻烦。

### Weblogic

Weblogic连接池参数非常丰富，连接池可以设置最大和最小连接，连接等待时间等，基本功能都有，还能够支持一些特殊的特性，比如连接固定线程。同时，weblogic连接池支持一套完成的监控体系。

### WebSphere

IBM的软件平台，它包含了编写、运行和监控全天候的工业强度的随需应变web应用程序和跨平台、跨产品解决方案所需要的整个中间件基础设置，如服务器、服务和工具。Websphere提供了可靠、灵活和健壮的软件。

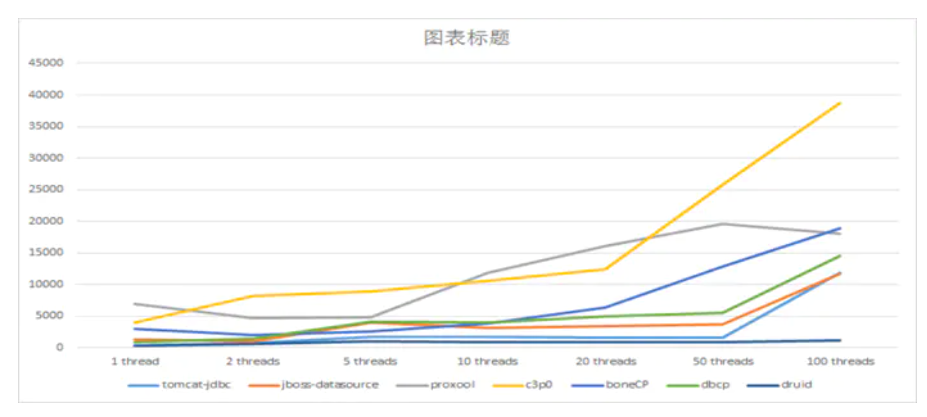
可以设置最大和最小连接，连接等待时间等，基本功能都有，而且支持自定义参数，参数比weblogic支持要少。

### Druid

近几年，阿里在开源项目上动作频频，除了有像fastJson、dubbo这类项目，更有像AliSQL这类的大型软件，今天说的Druid，就是阿里众多优秀开源项目中的一个。它除了提供性能卓越的连接池功能外，还集成了SQL监控，黑名单拦截等功能，用它自己的话说，Druid是“为监控而生”。借助于阿里这个平台的号召力，产品一经发布就赢得了大批用户的拥趸，从用户使用的反馈来看，Druid也确实没让用户失望。

相较于其他产品，Druid另一个比较大的优势，就是中文文档比较全面（毕竟是国人的项目么），在github的wiki页面，列举了日常使用中可能遇到的问题，对一个新用户来讲，上面提供的内容已经足够指导它完成产品的配置和使用了。

下图为Druid自己提供的性能测试数据：



现在项目开发中，我还是比较倾向于使用Durid，它不仅仅是一个数据库连接池，它还包含一个ProxyDriver，一系列内置的JDBC组件库，一个SQL Parser。

Druid 相对于其他数据库连接池的优点：

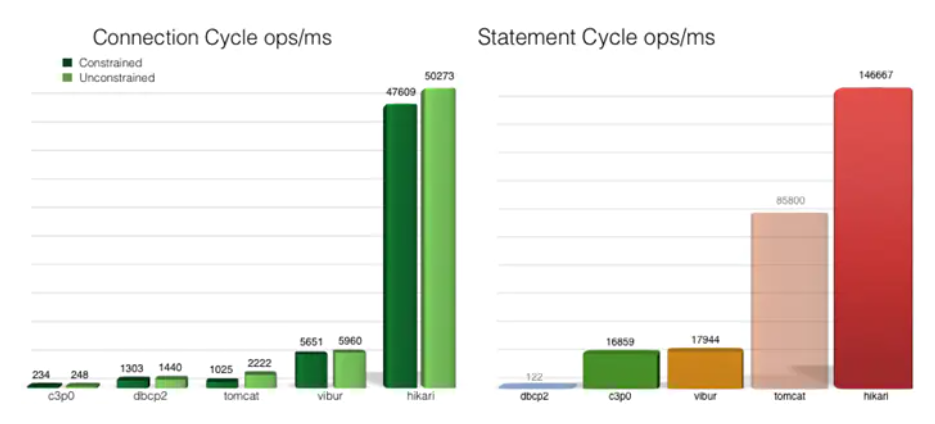
1、强大的监控特性，通过Druid提供的监控功能，可以清楚知道连接池和SQL的工作情况。  
 a. 监控SQL的执行时间、ResultSet持有时间、返回行数、更新行数、错误次数、错误堆栈信息；  
 b. SQL执行的耗时区间分布。什么是耗时区间分布呢？比如说，某个SQL执行了1000次，其中01毫秒区间50次，110毫秒800次，10100毫秒100次，1001000毫秒30次，1~10秒15次，10秒以上5次。通过耗时区间分布，能够非常清楚知道SQL的执行耗时情况；  
 c. 监控连接池的物理连接创建和销毁次数、逻辑连接的申请和关闭次数、非空等待次数、PSCache命中率等。

2、方便扩展。Druid提供了Filter-Chain模式的扩展API，可以自己编写Filter拦截JDBC中的任何方法，可以在上面做任何事情，比如说性能监控、SQL审计、用户名密码加密、日志等等。

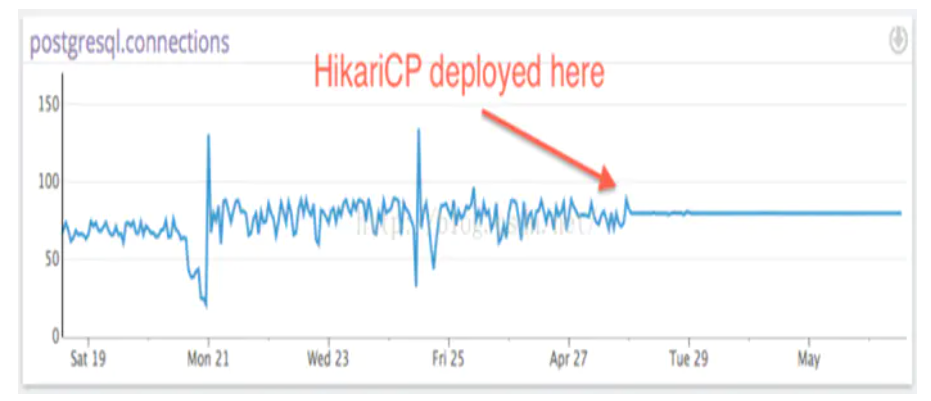
3、Druid集合了开源和商业数据库连接池的优秀特性，并结合阿里巴巴大规模苛刻生产环境的使用经验进行优化。

### HikariCP

HikariCP号称“性能杀手”（It’s Faster），它的表现究竟如何呢，先来看下官网提供的数据：



不光性能强劲，稳定性也不差：



那它是怎么做到如此强劲的呢？官网给出的说明如下：

字节码精简：优化代码，直到编译后的字节码最少，这样，CPU缓存可以加载更多的程序代码；

优化代理和拦截器：减少代码，例如HikariCP的Statement proxy只有100行代码；

自定义数组类型（FastStatementList）代替ArrayList：避免每次get()调用都要进行range check，避免调用remove()时的从头到尾的扫描；

自定义集合类型（ConcurrentBag）：提高并发读写的效率；

其他缺陷的优化，比如对于耗时超过一个CPU时间片的方法调用的研究（但没说具体怎么优化）。

可以看到，上述这几点优化，和现在能找到的资料来看，HakariCP在性能上的优势应该是得到共识的，再加上它自身小巧的身形，在当前的“云时代、微服务”的背景下，HakariCP一定会得到更多人的青睐。

## 总结

时至今日，虽然每个应用（需要RDBMS的）都离不开连接池，但在实际使用的时候，连接池已经可以做到“隐形”了。也就是说在通常情况下，连接池完成项目初始化配置之后，就再不需要再做任何改动了。不论你是选择Druid或是HikariCP，甚至是DBCP，它们都足够稳定且高效！之前讨论了很多关于连接池的性能的问题，但这些性能上的差异，是相较于其他连接池而言的，对整个系统应用来说，第二代连接池在使用过程中体会到的差别是微乎其微的，基本上不存在因为连接池的自身的配饰和使用导致系统性能下降的情况，除非是在单点应用的数据库负载足够高的时候（压力测试的时候），但即便是如此，通用的优化的方式也是单点改集群，而不是在单点的连接池上死扣。

# 分布式数据库实践

## GoldenDB

在GoldenDB分布式数据库中，proxy后端采用连接池与DB保持连接，提高并发性能。

### 设计思路

基本包含以下几个线程分别处理不同操作：

1. 创建连接池：实现动态地连接池增删
2. 管理线程（1个线程池1个管理线程）
3. KA保活线程（有全局保活链表控制，通过互斥锁和条件变量通信）
4. GC垃圾回收线程

### 基本流程

基本步骤如下：

1. 调用Event获取事件号
2. 根据不同事件号分别执行不同的操作：

2.1 初始化资源

2.1.1 加载配置文件配置项

2.1.2 申请连接池资源管理结构体s\_pool\_res（加互斥锁申请）

2.1.3 创建保活（KA）线程：

1）初始化保活全局结构ka\_pools

2）调用pthread\_create创建线程

2.1）统计每个pool上的空闲连接数

2.2）通过ping操作实现心跳检测即使发现异常

2.2.1）如果连接池关闭，将其从心跳队列移除

中断连接池中所有活跃的连接

将该连接池转移到垃圾回收队列中，等待垃圾回收线程释放该连接池对象

2.2.2）调用ping检测心跳

2.2.3）检查连接池中剩余的空闲连接句柄是否可用

2.2.3.1）实际连接数只计算当前空闲队列的连接数：高并发时，空闲队列大概率用尽，连接都在使用队列，此时只把空闲队列中连接当作实际连接数，可以避免频繁地创建和回收

2.2.3.2）用空闲时间的间隔来逐渐释放连接，可以起缓冲作用

2.2.3.3）没有在连接池中conn\_pool\_put释放，释放太快了，需要释放慢点，尽量高并发的时候复用已有的连接

2.3）睡眠1s继续（所以中间有1s链路有问题风险）

2.1.4 创建垃圾回收线程

调用pthread\_cond\_timeout定期清理

2.2 OMM管理流程：绑定和解绑集群

2.2.1 根据PM发过来的消息，找到对应的需要创建后端连接池的元数据信息（USER和所属DB等）

2.2.2 根据配置，创建多个连接池对象

2.2.3 设置连接参数并创建管理连接（每个连接池1个管理线程）

2.2.4 将连接池加入到资源库

2.3 OMM管理流程：增加和删除GROUP

2.4 OMM管理流程：增加和删除USER

2.5 dbtool统计断链信息

2.6 主备切换时，删除旧主已建后端数据库链路