springboot+mybatis+jpa+hikaricp

什么是Spring Boot

```
   /**
     * Spring Boot makes it easy to create stand-alone,
     * production-grade Spring based Applications that you can "just run".
     *
     * We take an opinionated view of the Spring platform and third-party
libraries
     * so you can get started with minimum fuss. Most Spring Boot applications
need
     * very little Spring configuration.
     */
```

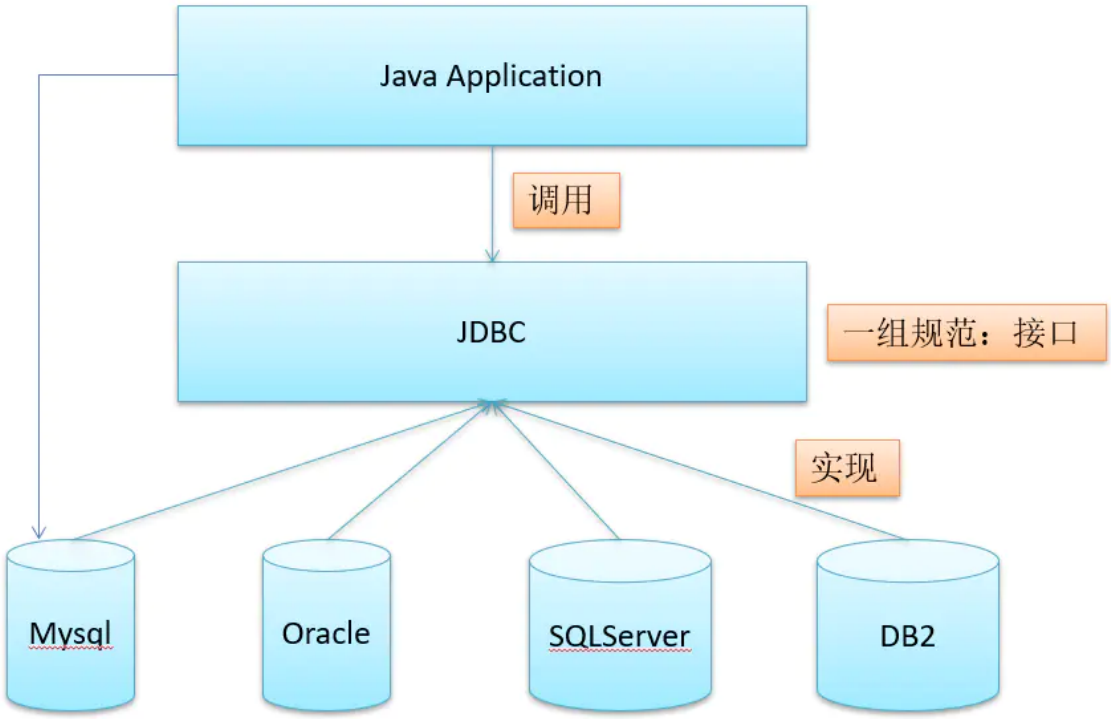
它使用“习惯优于配置”（项目中存在大量的配置，此外还内置了一个习惯性的配置，让你无需手动配置）的理念让你的项目快速运行起来。

它并不是什么新的框架，而是默认配置了很多框架的使用方式，就像Maven整合了所有的jar包一样，Spring Boot整合了所有框架。

Features

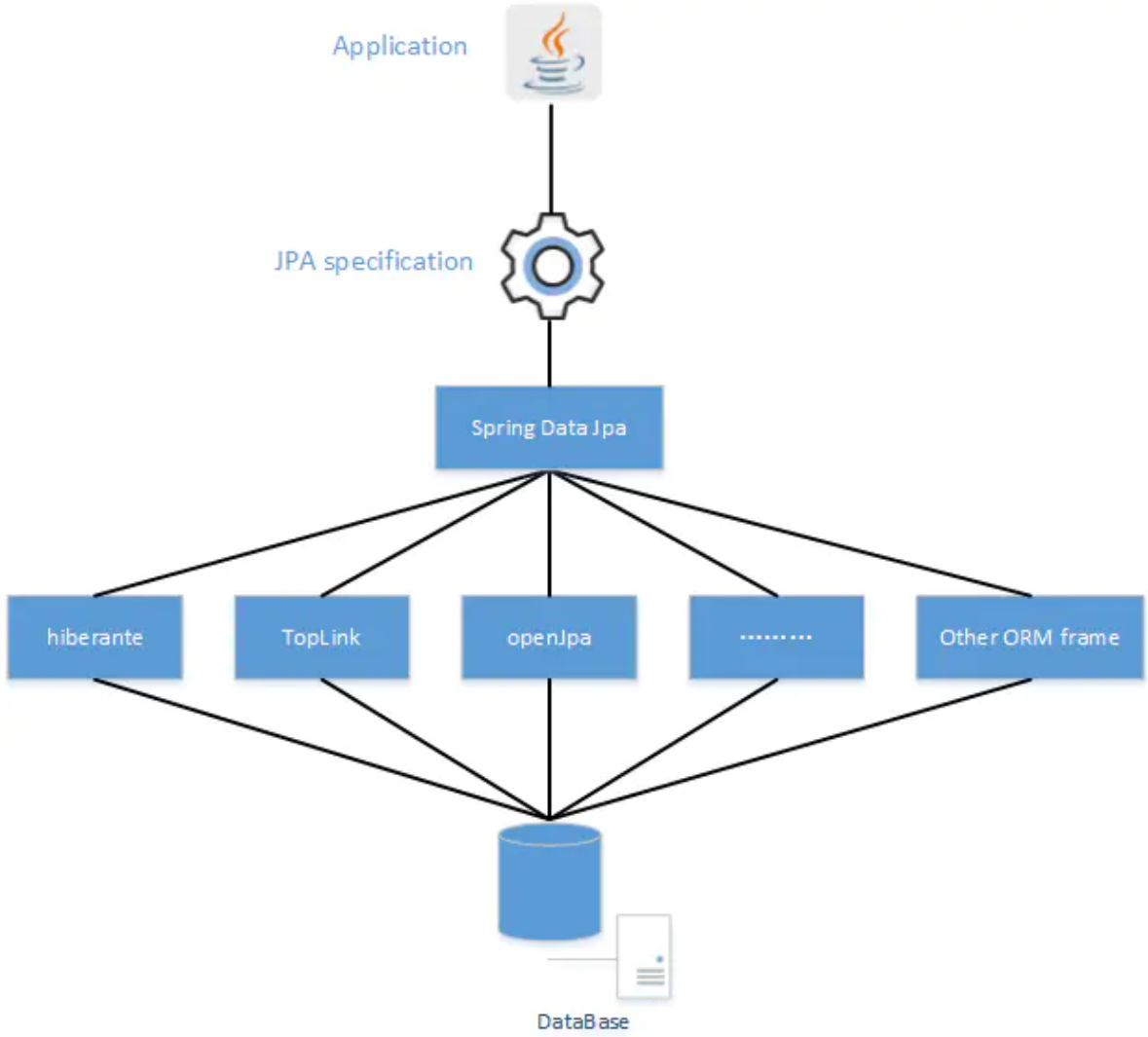
- 创建独立的Spring应用程序
- 直接嵌入Tomcat，Jetty或Undertow（无需部署WAR文件）
- 提供“初始”的POM文件内容，以简化Maven配置
- 尽可能自动配置Spring
- 提供生产就绪的功能，如指标，健康检查和外部化配置
- 绝对无代码生成，也不需要XML配置

JPA-ORM概述



1. JPA是什么

(1) Java Persistence API：用于对象持久化的 API
(2) Java EE 5.0 平台标准的 ORM 规范，使得应用程序以统一的方式访问持久层



2. JPA和Hibernate的关系

(1) JPA 是 hibernate 的一个抽象（就像JDBC和JDBC驱动的关系）：

- JPA 是规范：JPA 本质上就是一种 ORM 规范，不是ORM 框架 —— 因为 JPA 并未提供 ORM 实现，它只是制订了一些规范，提供了一些编程的 API 接口，但具体实现则由 ORM 厂商提供实现
- Hibernate 是实现：Hibernate 除了作为 ORM 框架之外，它也是一种 JPA 实现

(2) 从功能上来说， JPA 是 Hibernate 功能的一个子集

3. JPA的供应商

(1) JPA 的目标之一是制定一个可以由很多供应商实现的 API，目前Hibernate 3.2+、TopLink 10.1+ 以及 OpenJPA 都提供了 JPA 的实现

(2) Hibernate

- JPA 的始作俑者就是 Hibernate 的作者
- Hibernate 从 3.2 开始兼容 JPA

(3) OpenJPA

- OpenJPA 是 Apache 组织提供的开源项目

(4) TopLink

- TopLink 以前需要收费，如今开源了

4. JPA的优势

(1) 标准化: 提供相同的 API，这保证了基于JPA 开发的企业应用能够经过少量的修改就能够在不同的 JPA 框架下运行。

(2) 简单易用，集成方便: JPA 的主要目标之一就是提供更加简单的编程模型，在 JPA 框架下创建实体和创建 Java 类一样简单，只需要使用 javax.persistence.Entity 进行注释；JPA 的框架和接口也都非常简单，

(3) 可媲美JDBC的查询能力: JPA的查询语言是面向对象的，JPA定义了独特的JPQL，而且能够支持批量更新和修改、JOIN、GROUP BY、HAVING 等通常只有 SQL 才能够提供的高级查询特性，甚至还能够支持子查询。

(4) 支持面向对象的高级特性: JPA 中能够支持面向对象的高级特性，如类之间的继承、多态和类之间的复杂关系，最大限度的使用面向对象的模型

5. JPA包括3方面的技术

(1) ORM 映射元数据：JPA 支持 XML 和 JDK 5.0 注解两种元数据的形式，元数据描述对象和表之间的映射关系，框架据此将实体对象持久化到数据库表中。

(2) JPA 的 API：用来操作实体对象，执行CRUD操作，框架在后台完成所有的事情，开发者从繁琐的JDBC和 SQL代码中解脱出来。

(3) 查询语言（JPQL）：这是持久化操作中很重要的一个方面，通过面向对象而非面向数据库的查询语言查询数据，避免程序和具体的 SQL 紧密耦合。

MyBatis

特点:

- mybatis是一种持久层框架，也属于ORM映射。前身是ibatis。
- 相比于hibernatehibernate为全自动化，配置文件书写之后不需要书写sql语句，但是欠缺灵活，很多时候需要优化；
- mybatis为半自动化，需要自己书写sql语句，需要自己定义映射。增加了程序员的一些操作，但是带来了设计上的灵活，并且也是支持hibernate的一些特性，如延迟加载，缓存和映射等；对数据库的兼容性比hibernate差。移植性不好，但是可编写灵活和高性能的sql语句。

简单易学：本身就很小且简单。没有任何第三方依赖，最简单安装只要两个jar文件+配置几个sql映射文件易于学习，易于使用，通过文档和源代码，可以比较完全的掌握它的设计思路和实现。

灵活：mybatis不会对应用程序或者数据库的现有设计强加任何影响。 sql写在xml里，便于统一管理和优化。通过sql基本上可以实现我们不使用数据访问框架可以实现的所有功能，或许更多。

解除sql与程序代码的耦合：通过提供DAO层，将业务逻辑和数据访问逻辑分离，使系统的设计更清晰，更易维护，更易单元测试。sql和代码的分离，提高了可维护性。

提供映射标签，支持对象与数据库的ORM字段关系映射

提供对象关系映射标签，支持对象关系组建维护

提供XML标签，支持编写动态sql。

优缺点：

1.sql语句与代码分离，存放于xml配置文件中：

优点：便于维护管理，不用在java代码中找这些语句；

缺点： JDBC方式可以用用打断点的方式调试，但是Mybatis不能，需要通过log4j日志输出日志信息帮助调试，然后在配置文件中修改。

2.用逻辑标签控制动态SQL的拼接：

优点：用标签代替编写逻辑代码；

缺点：拼接复杂SQL语句时，没有代码灵活，拼写比较复杂。不要使用变通的手段来应对这种复杂的语句。

3.查询的结果集与java对象自动映射：

优点：保证名称相同，配置好映射关系即可自动映射或者，不配置映射关系，通过配置列名=字段名也可完成自动映射。

缺点：对开发人员所写的SQL依赖很强。

4.编写原声SQL：

优点：接近JDBC，比较灵活。

缺点：对SQL语句依赖程度很高；并且属于半自动，数据库移植比较麻烦，比如mysql数据库编程Oracle数据库，部分的sql语句需要调整。

HikariCP

Spring Boot 2默认数据库连接池选择了HikariCP

默认的数据库连接池由Tomcat换成HikariCP. 如果在一个Tomcat应用中用spring.datasource.type来强制使用Hikari连接池, 则可以去掉这个override.

为何选择HikariCP

HiKariCP是数据库连接池的一个后起之秀，号称性能最好，可以完美地PK掉其他连接池，是一个高性能的JDBC连接池，基于BoneCP做了不少的改进和优化。其作者还有另外一个开源作品——高性能的JSON解析器HikariJSON。

它，超快，快到连Spring Boot 2都宣布支持了。

代码体积更是少的可怜，130kb。

HikariCP为什么这么快

JDBC连接池的实现并不复杂，主要是对JDBC中几个核心对象Connection、Statement、PreparedStatement、CallableStatement以及ResultSet的封装与动态代理。接下来从几个方面来看看HikariCP为什么这么快：

优化并精简字节码

动态代理

- asm
- Javassist
- cglib
- jdk

JavassistProxyFactory

JavassistProxyFactory存在于工具包里com.zaxxer.hikari.util里，之所以使用Javassist生成动态代理，是因为其速度更快，相比于JDK Proxy生成的字节码更少，精简了很多不必要的字节码。

javassist

javassist是一个字节码类库，可以用他来动态生成类，动态修改类等等，还有一个比较常见的用途是AOP，比如对一些类统一加权限过滤，加日志监控等等。

Javassist 不仅是一个处理字节码的库，还有一项优点：可以用 Javassist 改变 Java 类的字节码，而无需真正了解关于字节码或者 Java 虚拟机(Java virtual machine JVM)结构的任何内容。比起在单条指令水平上工作的框架，它确实使字节码操作更可具有可行性了。

Javassist 使您可以检查、编辑以及创建 Java 二进制类。检查方面基本上与通过 Reflection API 直接在 Java 中进行的一样，但是当想要修改类而不只是执行它们时，则另一种访问这些信息的方法就很有用了。这是因为 JVM 设计上并没有提供在类装载到 JVM 中后访问原始类数据的任何方法，这项工作需要在 JVM 之外完成。

Javassist 使用 javassist.ClassPool 类跟踪和控制所操作的类。这个类的工作方式与 JVM 类装载器非常相似，但是有一个重要的区别是它不是将装载的、要执行的类作为应用程序的一部分链接，类池使所装载的类可以通过 Javassist API 作为数据使用。可以使用默认的类池，它是从 JVM 搜索路径中装载的，也可以定义一个搜索您自己的路径列表的类池。甚至可以直接从字节数组或者流中装载二进制类，以及从头开始创建新类。

装载到类池中的类由 javassist.CtClass 实例表示。与标准的 Java java.lang.Class 类一样， CtClass 提供了检查类数据（如字段和方法）的方法。不过，这只是 CtClass 的部分内容，它还定义了在类中添加新字段、方法和构造函数、以及改变类、父类和接口的方法。奇怪的是，Javassist 没有提供删除一个类中字段、方法或者构造函数的任何方法。

字段、方法和构造函数分别由 javassist.CtField、javassist.CtMethod 和 javassist.CtConstructor 的实例表示。这些类定义了修改由它们所表示的对象的所有方法的方法，包括方法或者构造函数中的实际字节码内容。

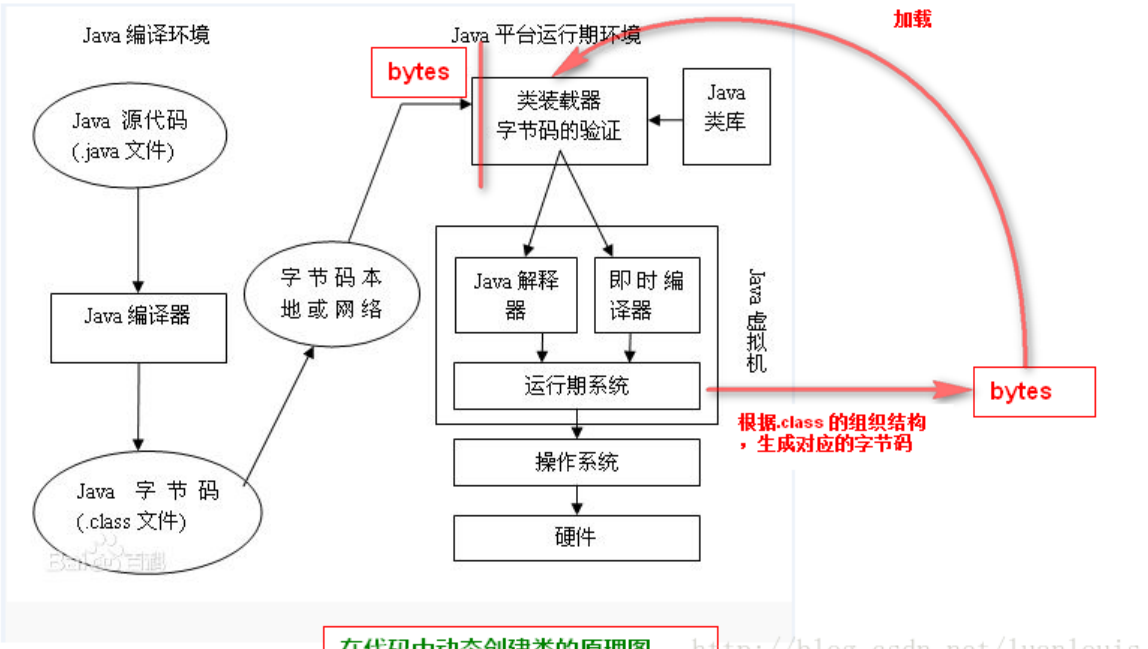
这篇来自阿里的文章做了一个动态代理的性能对比（http://javatar.iteye.com/blog/814426），得出的结论如下：

1. ASM和JAVAASSIST字节码生成方式不相上下，都很快，是CGLIB的5倍。
2. CGLIB次之，是JDK自带的两倍。
3. JDK自带的再次之，因JDK1.6对动态代理做了优化，如果用低版本JDK更慢，要注意的是JDK也是通过字节码生成来实现动态代理的，而不是反射。
4. JAVAASSIST提供者动态代理接口最慢，比JDK自带的还慢。 (这也是为什么网上有人说JAVAASSIST比JDK还慢的原因，用JAVAASSIST最好别用它提供的动态代理接口，而可以考虑用它的字节码生成方式)

差异的原因是各方案生成的字节码不一样，像JDK和CGLIB都考虑了很多因素，以及继承或包装了自己的一些类，所以生成的字节码非常大，而我们很多时候用不上这些，而手工生成的字节码非常小，所以速度快。

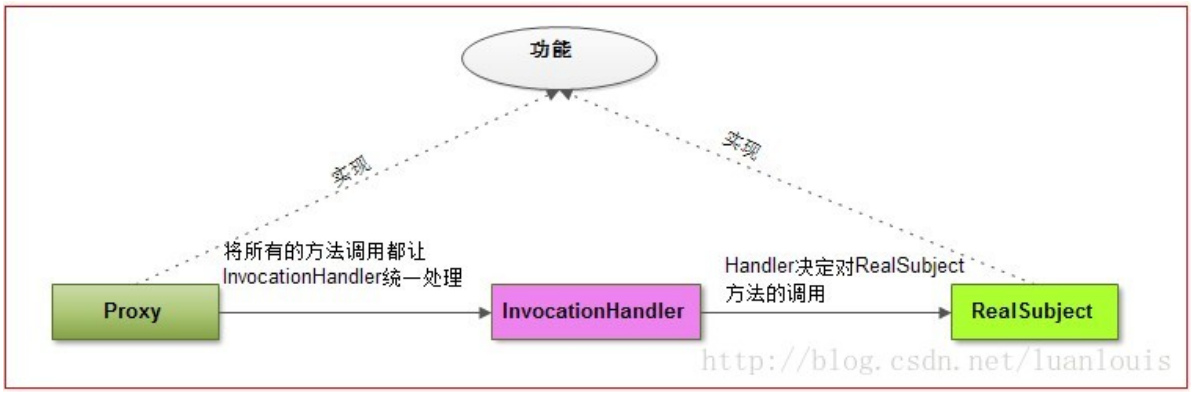
最终该阿里团队决定使用JAVAASSIST的字节码生成代理方式，虽然ASM稍快，但并没有快一个数量级，而JAVAASSIST的字节码生成方式比ASM方便，JAVAASSIST只需用字符串拼接出Java源码，便可生成相应字节码，而ASM需要手工写字节码。

字节码操作：ASM JAVAASSIST



在代码中动态创建类的原理图 http://blog.csdn.net/luanlouis

动态代理：



java动态代理是利用反射机制生成一个实现代理接口的匿名类，在调用具体方法前调用InvokeHandler来处理。

而cglib动态代理是利用asm开源包，对代理对象类的class文件加载进来，通过修改其字节码生成子类来处理。

1、如果目标对象实现了接口，默认情况下会采用JDK的动态代理实现AOP
2、如果目标对象实现了接口，可以强制使用CGLIB实现AOP

3、如果目标对象没有实现了接口，必须采用CGLIB库，spring会自动在JDK动态代理和CGLIB之间转换

如何强制使用CGLIB实现AOP?
（1）添加CGLIB库，SPRING_HOME/cglib/*.jar
（2）在spring配置文件中加入<aop:aspectj-autoproxy proxy-target-class="true"/>

JDK动态代理和CGLIB字节码生成的区别?
（1）JDK动态代理只能对实现了接口的类生成代理，而不能针对类
（2）CGLIB是针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的方法
因为是继承，所以该类或方法最好不要声明成final

ConcurrentBag：更好的并发集合类实现

ConcurrentBag的实现借鉴于C#中的同名类，是一个专门为连接池设计的lock-less集合，实现了比LinkedBlockingQueue、LinkedTransferQueue更好的并发性能。ConcurrentBag内部同时使用了ThreadLocal和CopyOnWriteArrayList来存储元素，其中CopyOnWriteArrayList是线程共享的。ConcurrentBag采用了queue-stealing的机制获取元素：首先尝试从ThreadLocal中获取属于当前线程的元素来避免锁竞争，如果没有可用元素则再次从共享的CopyOnWriteArrayList中获取。此外，ThreadLocal和CopyOnWriteArrayList在ConcurrentBag中都是成员变量，线程间不共享，避免了伪共享(false sharing)的发生。

HikariCP连接池是基于自主实现的ConcurrentBag完成的数据连接的多线程共享交互，是HikariCP连接管理快速的其中一个关键点。

ConcurrentBag是一个专门的并发包裹，在连接池（多线程数据交互）的实现上具有比LinkedBlockingQueue和LinkedTransferQueue更优越的性能。
ConcurrentBag通过拆分 CopyOnWriteArrayList、ThreadLocal和SynchronousQueue进行并发数据交互。

- CopyOnWriteArrayList：负责存放ConcurrentBag中全部用于出借的资源
- ThreadLocal：用于加速线程本地化资源访问
- SynchronousQueue：用于存在资源等待线程时的第一手资源交接

```
private final CopyOnWriteArrayList<T> sharedList;
private final ThreadLocal<List<Object>> threadList;
private final SynchronousQueue<T> handoffQueue;
```

ConcurrentBag中全部的资源均只能通过add方法进行添加，只能通过remove方法进行移出。

```
public void add(final T bagEntry)
```

```
{
   if (closed) {
      LOGGER.info("ConcurrentBag has been closed, ignoring add()");
      throw new IllegalStateException("ConcurrentBag has been closed, ignoring
add()");
   }

   sharedList.add(bagEntry); //新添加的资源优先放入CopyOnWriteArrayList

   // 当有等待资源的线程时，将资源交到某个等待线程后才返回（SynchronousQueue）
   while (waiters.get() > 0 && !handoffQueue.offer(bagEntry)) {
      yield();
   }
}

public boolean remove(final T bagEntry)
{
   // 如果资源正在使用且无法进行状态切换，则返回失败
   if (!bagEntry.compareAndSet(STATE_IN_USE, STATE_REMOVED) &&
!bagEntry.compareAndSet(STATE_RESERVED, STATE_REMOVED) && !closed) {
      LOGGER.warn("Attempt to remove an object from the bag that was not
borrowed or reserved: {}", bagEntry);
      return false;
   }

   final boolean removed = sharedList.remove(bagEntry); // 从CopyOnWriteArrayList
中移出
   if (!removed && !closed) {
      LOGGER.warn("Attempt to remove an object from the bag that does not exist:
{}", bagEntry);
   }

   return removed;
}
```

ConcurrentBag中通过borrow方法进行数据资源借用，通过requite方法进行资源回收，注意其中borrow方法只提供对象引用，不移除对象，因此使用时通过borrow取出的对象必须通过requite方法进行放回，否则容易导致内存泄露!

```
public T borrow(long timeout, final TimeUnit timeUnit) throws
InterruptedException
{
   // 优先查看有没有可用的本地化的资源
   final List<Object> list = threadList.get();
   for (int i = list.size() - 1; i >= 0; i--) {
      final Object entry = list.remove(i);
      @SuppressWarnings("unchecked")
      final T bagEntry = weakThreadLocals ? ((WeakReference<T>) entry).get() :
(T) entry;
      if (bagEntry != null && bagEntry.compareAndSet(STATE_NOT_IN_USE,
STATE_IN_USE)) {
         return bagEntry;
      }
   }
```

```
   final int waiting = waiters.incrementAndGet();
   try {
      // 当无可用本地化资源时，遍历全部资源，查看是否存在可用资源
      // 因此被一个线程本地化的资源也可能被另一个线程“抢走”
      for (T bagEntry : sharedList) {
         if (bagEntry.compareAndSet(STATE_NOT_IN_USE, STATE_IN_USE)) {
            if (waiting > 1) {
                // 因为可能“抢走”了其他线程的资源，因此提醒包裹进行资源添加
               listener.addBagItem(waiting - 1);
            }
            return bagEntry;
         }
      }

      listener.addBagItem(waiting);

      timeout = timeUnit.toNanos(timeout);
      do {
         final long start = currentTime();
         // 当现有全部资源全部在使用中，等待一个被释放的资源或者一个新资源
         final T bagEntry = handoffQueue.poll(timeout, NANOSECONDS);
         if (bagEntry == null || bagEntry.compareAndSet(STATE_NOT_IN_USE,
STATE_IN_USE)) {
            return bagEntry;
         }

         timeout -= elapsedNanos(start);
      } while (timeout > 10_000);

      return null;

   }
   finally {
      waiters.decrementAndGet();
   }
}

public void requite(final T bagEntry)
{
   // 将状态转为未在使用
   bagEntry.setState(STATE_NOT_IN_USE);

   // 判断是否存在等待线程，若存在，则直接转手资源
   for (int i = 0; waiters.get() > 0; i++) {
      if (bagEntry.getState() != STATE_NOT_IN_USE ||
handoffQueue.offer(bagEntry)) {
         return;
      }
      else if ((i & 0xff) == 0xff) {
         parkNanos(MICROSECONDS.toNanos(10));
      }
      else {
         yield();
      }
   }

   // 否则，进行资源本地化
   final List<Object> threadLocalList = threadList.get();
```

```
    threadLocalList.add(weakThreadLocals ? new WeakReference<>(bagEntry) :
bagEntry);
}
```

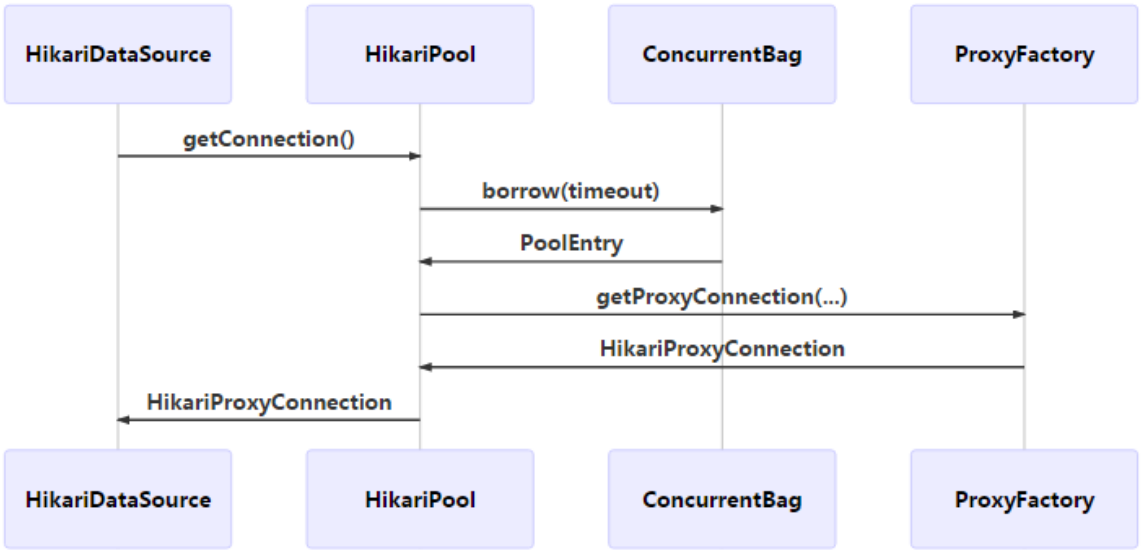
上述代码中的 weakThreadLocals 是用来判断是否使用弱引用，通过下述方法初始化：

```
private boolean useWeakThreadLocals()
{
    try {
        // 人工指定是否使用弱引用，但是官方不推荐进行自主设置。
        if (System.getProperty("com.dareway.concurrent.useWeakReferences") !=
null) {
            return Boolean.getBoolean("com.dareway.concurrent.useWeakReferences");
        }

        // 默认通过判断初始化的ClassLoader是否是系统的ClassLoader来确定
        return getClass().getClassLoader() != ClassLoader.getSystemClassLoader();

    }
    catch (SecurityException se) {
        return true;
    }
}
```

使用FastList替代ArrayList

FastList是一个List接口的精简实现，只实现了接口中必要的几个方法。JDK ArrayList每次调用get()方法时都会进行rangeCheck检查索引是否越界，FastList的实现中去除了这一检查，只要保证索引合法那么rangeCheck就成为了不必要的计算开销(当然开销极小)。此外，HikariCP使用List来保存打开的Statement，当Statement关闭或Connection关闭时需要将对应的Statement从List中移除。通常情况下，同一个Connection创建了多个Statement时，后打开的Statement会先关闭。ArrayList的remove(Object)方法是从头开始遍历数组，而FastList是从数组的尾部开始遍历，因此更为高效。

调用链路



竞品比较

功能类别	功能	Druid	HikariCP	DBCP	Tomcat-jdbc	C3P0
性能	PSCache	是	否	是	是	是
LRU		是	是	是	是	是
SLB负载均衡支持		是	否	否	否	否
稳定性	ExceptionSorter	是	否	否	否	否
扩展	扩展	Filter			JdbcIntercepter	
监控	监控方式	jmx/log/http	jmx/metrics	jmx	jmx	jmx
支持SQL级监控		是	否	否	否	否
Spring/Web关联监控		是	否	否	否	否
	诊断支持	LogFilter	否	否	否	否
连接泄露诊断	logAbandoned	是	否	否	否	否
安全	SQL防注入	是	无	无	无	无
支持配置加密		是	否	否	否	否

前瞻

站在巨人肩膀上的第二代连接池HikariCP和druid到底孰强孰弱？其实我觉得这是一个不必讨论的问题。

我们先来看看未来的趋势：单机的操作系统将会被抛弃，取而代之的是容器调度加编排的云操作系统。裸机或者虚拟机的运行时也将会被容器取代。通信方面将会使用Service Mesh。

也就是说中间件最后的趋势一定是弱化到无感知，这才是最终的一个大道至简的方向。那些maven依赖问题，把二方库写在pom里，监控等代码的硬编码进应用里都将逐渐弱化到不复存在，取而代之的那些java agent（如pinpoint、skywalking之类），抑或是service mesh这种side car模式都是可以做中间件（包括连接池）的监控的。

一个有赞的朋友告诉我，在有赞核心应用，用HikariCP替换durid后，RT出现断崖式下滑（1.5ms ~ 1.2ms） 并且持续稳定毛刺少。性能测试与压测之后，一核心系统与druid相比，性能提高一倍左右。

只统计java文件和xml文件，druid(alibaba-druid)总行数:430289，HikariCP(brettwooldridge-HikariCP)总行数:18372。 只统计java代码，druid(alibaba-druid)总行数:428749，HikariCP(brettwooldridge-HikariCP)总行数:17556。 再过滤一下test目录，(alibaba-druid)总行数:215232，(brettwooldridge-HikariCP)总行数:7960。 光一个DruidDataSource就3000行，且不说性能，druid是在jdbc的基础上，自己编码做得增强。

如果这么说，druid准确的说是生活在第一代和第二代连接池的面向过程的年代。druid可能忘了松耦合这个概念，把监控和数据库连接池做在一个项目里，本身就是紧耦合。既然微服务提倡业务隔离性，那么这种难道不应该隔离么？让组件工具一次只做一件事不好么？监控的事情在service mesh的将来毕竟是有别的其天然的监控手法的而不是硬编码在一个小小的连接池里。综上所述，放在现在或是未来的趋势去拼，大概率比不过拥抱springboot 2.0以及大道至简精简到极致的HikariCP。

未来的中间件，一定是和spring生态圈和servich mesh一样，大道至简，越来越薄，升级中间件不再是需要用户强行升级maven依赖解决依赖冲突，而是通过mesh的方式极致到升级让业务方无感知。所以那些热部署、潘多拉boot、容器隔离等解决依赖冲突的妥协方式也将可能大概率被置换掉。