二

09 案例分析：池化对象的应用场景

在我们平常的编码中，通常会将一些对象保存起来，这主要考虑的是对象的创建成本。比如像线程资源、数据库连接资源或者 TCP 连接等，这类对象的初始化通常要花费比较长的时间，如果频繁地申请和销毁，就会耗费大量的系统资源，造成不必要的性能损失。

并且这些对象都有一个显著的特征，就是通过轻量级的重置工作，可以循环、重复地使用。这个时候，我们就可以**使用一个虚拟的池子，将这些资源保存起来，当使用的时候，我们就从池子里快速获取一个即可。**

在 Java 中，**池化技术**应用非常广泛，常见的就有数据库连接池、线程池等，本节课主讲连接池，线程池我们将在 12 课时进行介绍。

公用池化包 Commons Pool 2.0

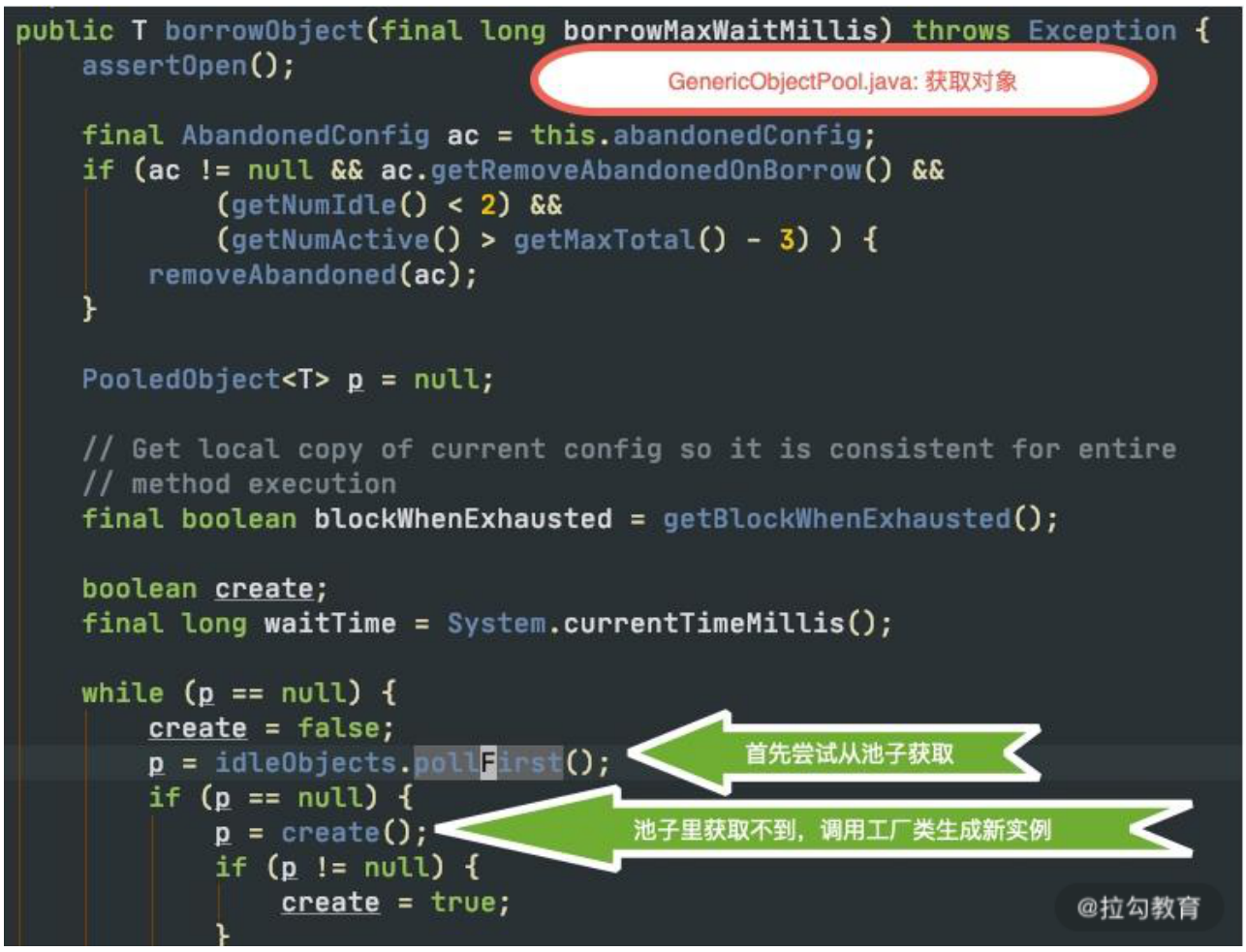
我们首先来看一下 Java 中公用的池化包 Commons Pool 2.0，来了解一下对象池的一般结构。根据我们的业务需求，使用这套 API 能够很容易实现对象的池化管理。

GenericObjectPool 是对象池的核心类，通过传入一个对象池的配置和一个对象的工厂，即可快速创建对象池。

```
public GenericObjectPool(
            final PooledObjectFactory<T> factory,
            final GenericObjectPoolConfig<T> config)
```

Redis 的常用客户端 Jedis，就是使用 Commons Pool 管理连接池的，可以说是一个最佳实践。下图是 Jedis 使用工厂**创建对象**的主要代码块。对象工厂类最主要的方法就是 makeObject，它的返回值是 PooledObject 类型，可以将对象使用 new DefaultPooledObject<>(obj) 进行简单包装返回。

我们再来介绍一下对象的生成过程，如下图，对象在进行**获取**时，将首先尝试从对象池里拿出一个，如果对象池中没有空闲的对象，就使用工厂类提供的方法，生成一个新的。

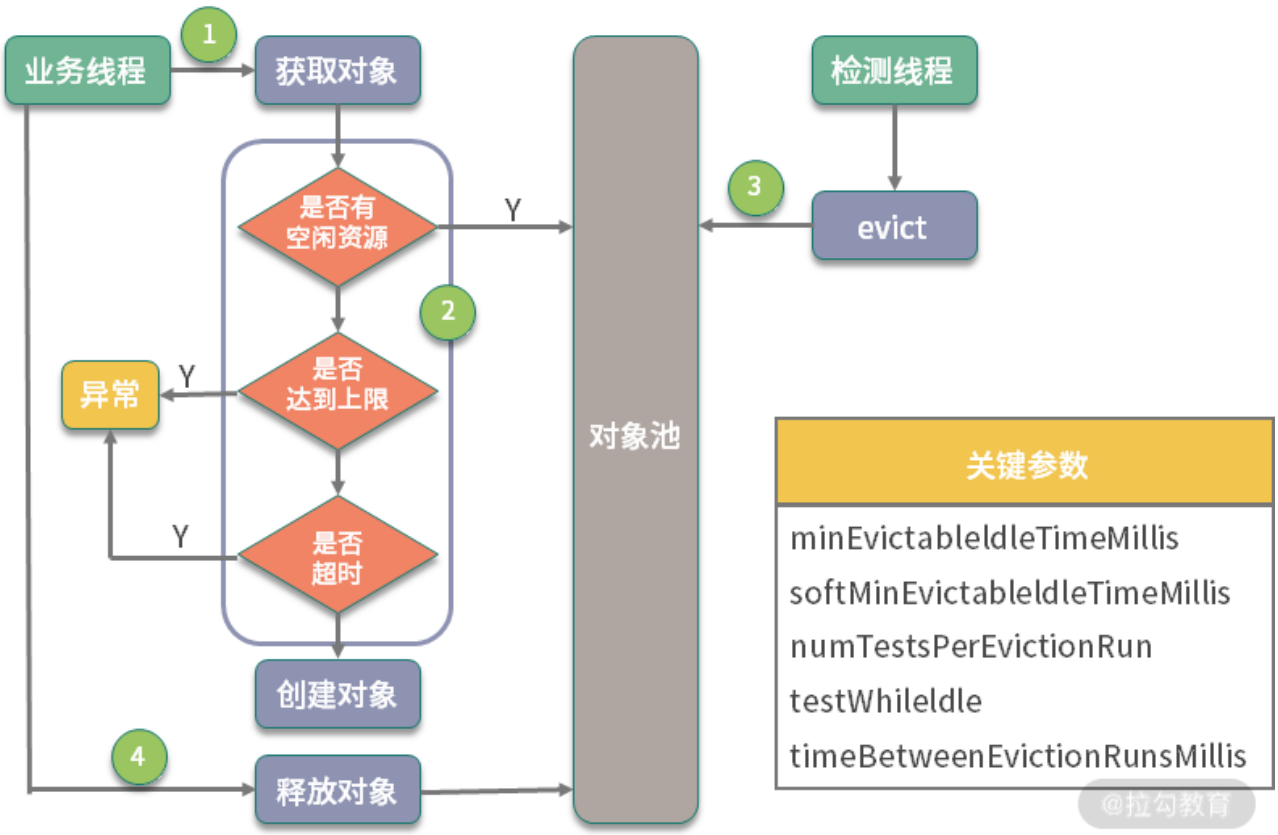


那对象是存在什么地方的呢？这个存储的职责，就是由一个叫作 LinkedBlockingDeque的结构来承担的，它是一个双向的队列。

接下来看一下 GenericObjectPoolConfig 的主要属性：

```
private int maxTotal = DEFAULT_MAX_TOTAL;
private int maxIdle = DEFAULT_MAX_IDLE;
private int minIdle = DEFAULT_MIN_IDLE;
private boolean lifo = DEFAULT_LIFO;
private boolean fairness = DEFAULT_FAIRNESS;
private long maxWaitMillis = DEFAULT_MAX_WAIT_MILLIS;
private long minEvictableIdleTimeMillis =          DEFAULT_MIN_EVICTABLE_IDLE_TIME_
private long evictorShutdownTimeoutMillis =          DEFAULT_EVICTOR_SHUTDOWN_TIMEO
private long softMinEvictableIdleTimeMillis =          DEFAULT_SOFT_MIN_EVICTABLE_ID
private int numTestsPerEvictionRun =           DEFAULT_NUM_TESTS_PER_EVICTION_RUN;
private EvictionPolicy<T> evictionPolicy = null; // Only 2.6.0 applications set thi
private String evictionPolicyClassName = DEFAULT_EVICTION_POLICY_CLASS_NAME;
private boolean testOnCreate = DEFAULT_TEST_ON_CREATE;
private boolean testOnBorrow = DEFAULT_TEST_ON_BORROW;
private boolean testOnReturn = DEFAULT_TEST_ON_RETURN;
private boolean testWhileIdle = DEFAULT_TEST_WHILE_IDLE;
private long timeBetweenEvictionRunsMillis = DEFAULT_TIME_BETWEEN_EVICTION_RUNS_MIL
private boolean blockWhenExhausted = DEFAULT_BLOCK_WHEN_EXHAUSTED;
```

参数很多，要想了解参数的意义，我们首先来看一下一个池化对象在整个池子中的生命周期。如下图所示，池子的操作主要有两个：一个是**业务线程**，一个是**检测线程**。



对象池在进行初始化时，要指定三个主要的参数：

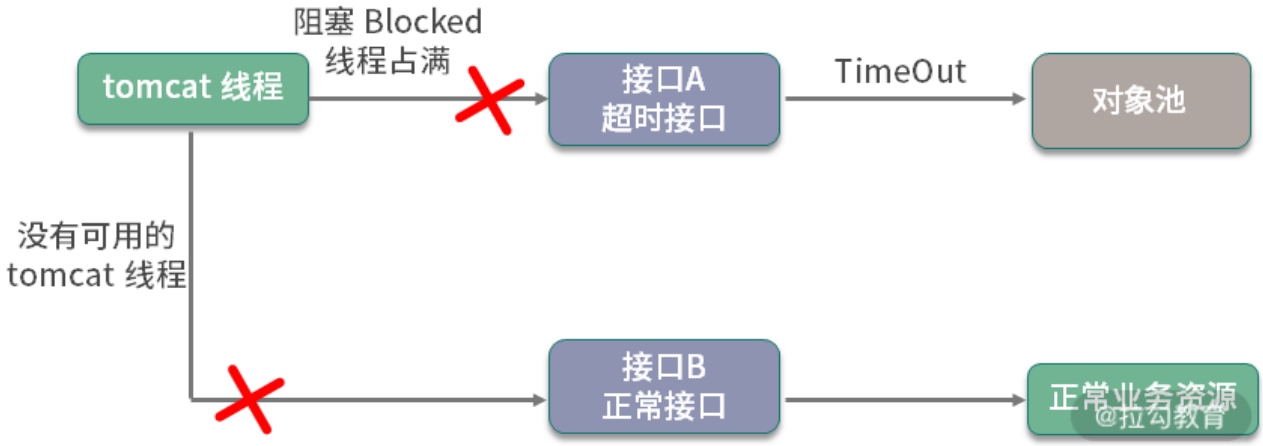
- **maxTotal** 对象池中管理的对象上限
- **maxIdle** 最大空闲数

- **minIdle** 最小空闲数

其中 **maxTotal** 和业务线程有关，当业务线程想要获取对象时，会首先检测是否有空闲的对象。如果有，则返回一个；否则进入创建逻辑。此时，如果池中个数已经达到了最大值，就会创建失败，返回空对象。

对象在获取的时候，有一个非常重要的参数，那就是**最大等待时间（maxWaitMillis）**，这个参数对应用方的性能影响是比较大的。该参数默认为 -1，表示永不超时，直到有对象空闲。

如下图，如果对象创建非常缓慢或者使用非常繁忙，业务线程会持续阻塞（blockWhenExhausted 默认为 true），进而导致正常服务也不能运行。



一般面试官会问：你会把超时参数设置成多大呢？

我一般都会把最大等待时间，设置成接口可以忍受的最大延迟。比如，一个正常服务响应时间 10ms 左右，达到 1 秒钟就会感觉到卡顿，那么这个参数设置成 500~1000ms 都是可以的。超时之后，会抛出 NoSuchElementException 异常，请求**会快速失败**，不会影响其他业务线程，这种 Fail Fast 的思想，在互联网应用非常广泛。

带有**evcit** 字样的参数，主要是处理对象逐出的。池化对象除了初始化和销毁的时候比较昂贵，在运行时也会占用系统资源。比如，**连接池**会占用多条连接，**线程池**会增加调度开销等。业务在突发流量下，会申请到超出正常情况的对象资源，放在池子中。等这些对象不再被使用，我们就需要把它清理掉。

超出 **minEvictableIdleTimeMillis** 参数指定值的对象，就会被强制回收掉，这个值默认是 30 分钟；**softMinEvictableIdleTimeMillis** 参数类似，但它只有在当前对象数量大于 minIdle 的时候才会执行移除，所以前者的动作要更暴力一些。

还有 4 个 test 参数：**testOnCreate**、**testOnBorrow**、**testOnReturn**、**testWhileIdle**，分

别指定了在创建、获取、归还、空闲检测的时候，是否对池化对象进行有效性检测。

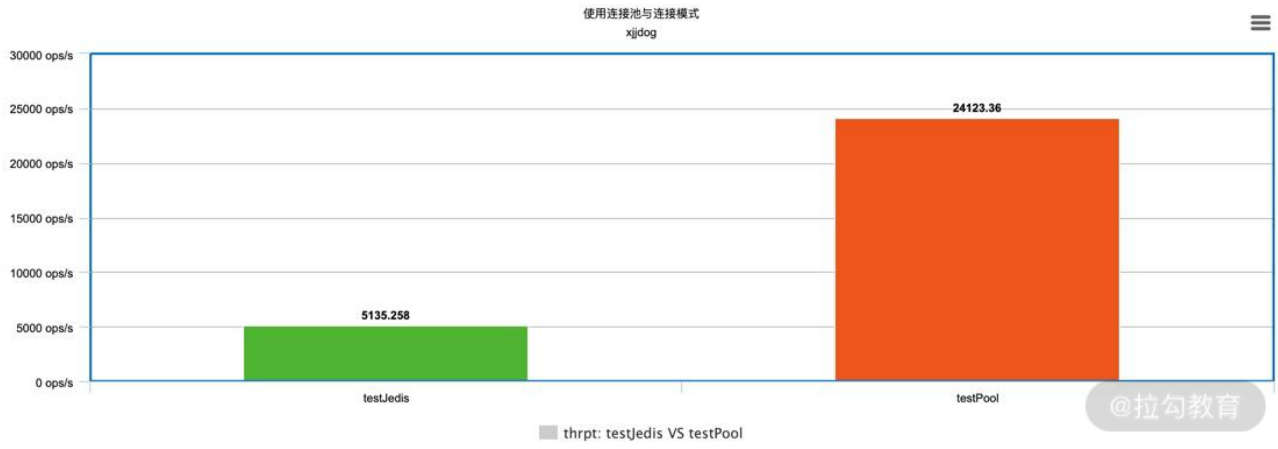
开启这些检测，能保证资源的有效性，但它会耗费性能，所以默认为 false。生产环境上，建议只将 **testWhileIdle** 设置为 true，并通过调整**空闲检测时间间隔（timeBetweenEvictionRunsMillis）**，比如 1 分钟，来保证资源的可用性，同时也保证效率。

Jedis JMH 测试

使用连接池和不使用连接池，它们之间的性能差距到底有多大呢？下面是一个简单的 JMH 测试例子（见仓库），进行一个简单的 set 操作，为 redis 的 key 设置一个随机值。

```
@Fork(2)
@State(Scope.Benchmark)
@Warmup(iterations = 5, time = 1)
@Measurement(iterations = 5, time = 1)
@BenchmarkMode(Mode.Throughput)
public class JedisPoolVSJedisBenchmark {
    JedisPool pool = new JedisPool("localhost", 6379);
    @Benchmark
    public void testPool() {
        Jedis jedis = pool.getResource();
        jedis.set("a", UUID.randomUUID().toString());
        jedis.close();
    }
    @Benchmark
    public void testJedis() {
        Jedis jedis = new Jedis("localhost", 6379);
        jedis.set("a", UUID.randomUUID().toString());
        jedis.close();
    }
...
```

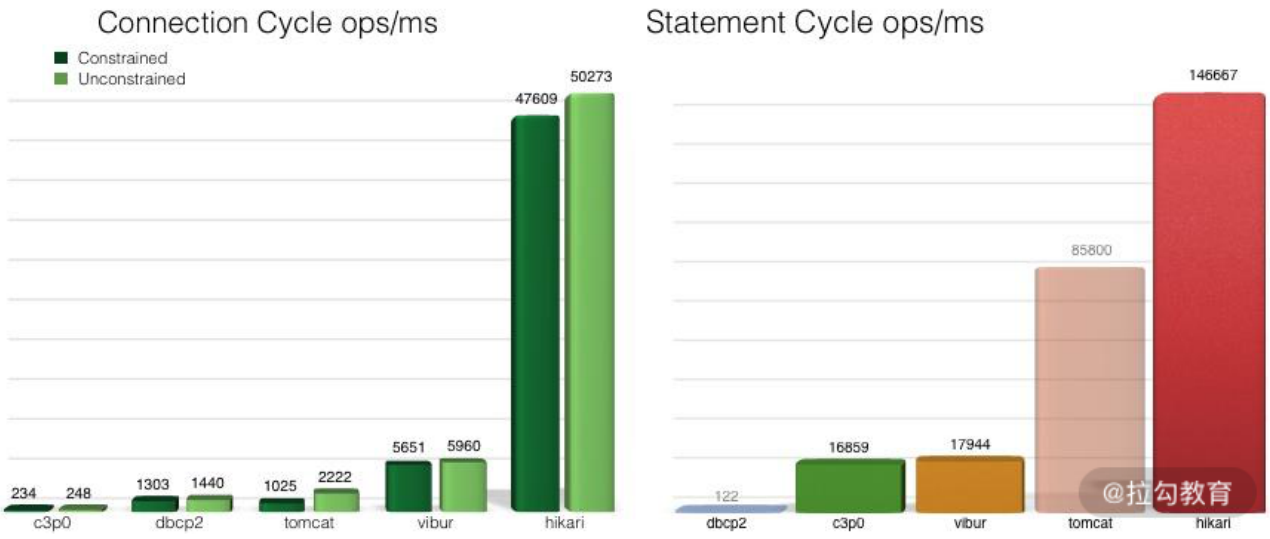
将测试结果使用 meta-chart 作图，展示结果如下图所示，可以看到使用了连接池的方式，它的吞吐量是未使用连接池方式的 5 倍！



数据库连接池 HikariCP

HikariCP 源于日语“光”的意思（和光速一样快），它是 SpringBoot 中默认的数据库连接池。数据库是我们工作中经常使用到的组件，针对数据库设计的客户端连接池是非常多的，它的设计原理与我们在本课时开头提到的基本一致，可以有效地减少数据库连接创建、销毁的资源消耗。

同是连接池，它们的性能也是有差别的，下图是 HikariCP 官方的一张测试图，可以看到它优异的性能，官方的 JMH 测试代码见 Github，我也已经拷贝了一份到仓库中。



一般面试题是这么问的： HikariCP 为什么快呢？主要有三个方面：

- 它使用 FastList 替代 ArrayList，通过初始化的默认值，减少了越界检查的操作；
- 优化并精简了字节码，通过使用 Javassist，减少了动态代理的性能损耗，比如使用 invokestatic 指令代替 invokevirtual 指令；
- 实现了无锁的 ConcurrentBag，减少了并发场景下的锁竞争。

HikariCP 对性能的一些优化操作，是非常值得我们借鉴的，在之后的 16 课时，我们将详细分析几个优化场景。

数据库连接池同样面临一个最大值（maximumPoolSize）和最小值（minimumIdle）的问题。**这里同样有一个非常高频的面试题：你平常会把连接池设置成多大呢？**

很多同学认为，**连接池的大小设置得越大越好，有的同学甚至把这个值设置成 1000 以上，这是一种误解。**根据经验，数据库连接，只需要 20~50 个就够用了。具体的大小，要根据业务属性进行调整，但大得离谱肯定是不合适的。

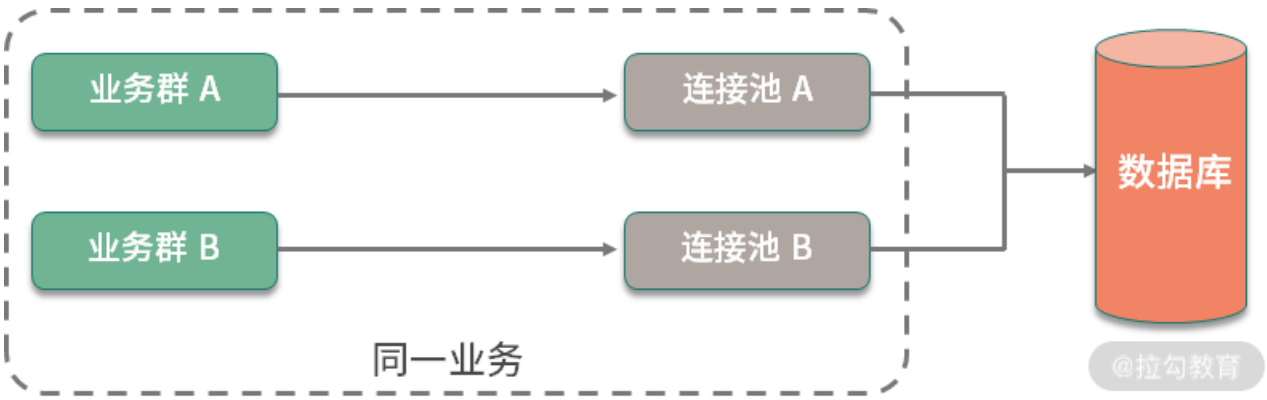
HikariCP 官方是不推荐设置 minimumIdle 这个值的，它将被默认设置成和

maximumPoolSize 一样的大小。如果你的数据库Server端连接资源空闲较大，不妨也可以去掉连接池的动态调整功能。

另外，根据数据库查询和事务类型，一个应用中是可以配置多个数据库连接池的，这个优化技巧很少有人知道，在此简要描述一下。

业务类型通常有两种：一种需要快速的响应时间，把数据尽快返回给用户；另外一种是可以在后台慢慢执行，耗时比较长，对时效性要求不高。如果这两种业务类型，共用一个数据库连接池，就容易发生资源争抢，进而影响接口响应速度。虽然微服务能够解决这种情况，但大多数服务是没有这种条件的，这时就可以对连接池进行拆分。

如图，在同一个业务中，根据业务的属性，我们分了两个连接池，就是来处理这种情况的。



HikariCP 还提到了另外一个知识点，在 JDBC4 的协议中，通过 Connection.isValid() 就可以检测连接的有效性。这样，我们就不用设置一大堆的 test 参数了，HikariCP 也没有提供这样的参数。

结果缓存池

到了这里你可能会发现池（Pool）与缓存（Cache）有许多相似之处。

它们之间的一个共同点，就是将对象加工后，存储在相对高速的区域。我习惯性将**缓存**看作是**数据对象**，而把**池中的对象**看作是**执行对象**。缓存中的数据有一个命中率问题，而池中的对象一般都是对等的。

考虑下面一个场景，jsp 提供了网页的动态功能，它可以在执行后，编译成 class 文件，加快执行速度；再或者，一些媒体平台，会将热门文章，定时转化成静态的 html 页面，仅靠 nginx 的负载均衡即可应对高并发请求（动静分离）。

这些时候，你很难说清楚，**这是针对缓存的优化，还是针对对象进行了池化，它们在本质**上只是保存了某个执行步骤的结果，使得下次访问时不需要从头再来。我通常把这种技术叫作

结果缓存池（Result Cache Pool），属于多种优化手段的综合。

小结

下面我来简单总结一下该课时的内容重点：

我们从 Java 中最通用的公用池化包 **Commons Pool 2.0** 说起，介绍了它的一些实现细节，并对一些重要参数的应用做了讲解；**Jedis** 就是在 Commons Pool 2.0 的基础上封装的，通过 JMH 测试，我们发现对象池化之后，有了接近 5 倍的性能提升；接下来介绍了数据库连接池中速度速快的 **HikariCP** ，它在池化技术之上，又通过编码技巧进行了进一步的性能提升，HikariCP 是我重点研究的类库之一，我也建议你加入自己的任务清单中。

总体来说，当你遇到下面的场景，就可以考虑使用池化来增加系统性能：

- 对象的创建或者销毁，需要耗费较多的系统资源；
- 对象的创建或者销毁，耗时长，需要繁杂的操作和较长时间的等待；
- 对象创建后，通过一些状态重置，可被反复使用。

将对象池化之后，只是开启了第一步优化。要想达到最优性能，就不得不调整池的一些关键参数，合理的池大小加上合理的超时时间，就可以让池发挥更大的价值。和缓存的命中率类似，对池的监控也是非常重要的。

如下图，可以看到数据库连接池连接数长时间保持在高位不释放，同时等待的线程数急剧增加，这就能帮我们快速定位到数据库的事务问题。

平常的编码中，有很多类似的场景。比如 Http 连接池，Okhttp 和 Httpclient 就都提供了连接池的概念，你可以类比着去分析一下，关注点也是在连接大小和超时时间上；在底层的中间件，比如 RPC，也通常使用连接池技术加速资源获取，比如 Dubbo 连接池、 Feign 切换成 httppclient 的实现等技术。

你会发现，在不同资源层面的池化设计也是类似的。比如**线程池**，通过队列对任务进行了二层缓冲，提供了多样的拒绝策略等，线程池我们将在 12 课时进行介绍。线程池的这些特

性，你同样可以借鉴到连接池技术中，用来缓解请求溢出，创建一些溢出策略。现实情况中，我们也会这么做。那么具体怎么做？有哪些做法？这部分内容就留给大家思考了，欢迎你在下方留言，与大家一起分享讨论，我也会针对你的思考进行一一点评。

但无论以何种方式处理对象，让对象保持精简，提高它的复用度，都是我们的目标，所以下一课时，我将系统讲解大对象的复用和注意点。

上一页 下一页