拓展 7：妙手仁心 —— 优雅地使用 Jedis

本节面向 Java 用户，主题是如何优雅地使用 Jedis 编写应用程序，既可以让代码看起来赏心悦目，又可以避免使用者犯错。

Jedis 是 Java 用户最常用的 Redis 开源客户端。它非常小巧，实现原理也很简单，最重要的是很稳定，而且使用的方法参数名称和官方的文档非常 match，如果有什么方法不会用，直接参考官方的指令文档阅读一下就会了，省去了非必要的重复学习成本。不像有些客户端把方法名称都换了，虽然表面上给读者带来了便捷，但是需要挨个重新学习这些 API，提高了学习成本。

Java 程序一般都是多线程的应用程序，意味着我们很少直接使用 Jedis，而是要用到 Jedis 的连接池 —— JedisPool。同时因为 Jedis 对象不是线程安全的，当我们要使用 Jedis 对象时，需要从连接池中拿出一个 Jedis 对象独占，使用完毕后再将这个对象还给连接池。

用代码表示如下：

java

```
import redis.clients.jedis.Jedis;
import redis.clients.jedis.JedisPool;

public class JedisTest {

  public static void main(String[] args) {
    JedisPool pool = new JedisPool();
    Jedis jedis = pool.getResource(); // 拿出 Jedis 链接对象
    doSomething(jedis);
    jedis.close(); // 归还链接
  }

  private static void doSomething(Jedis jedis) {
    // code it here
  }

}
```

上面的代码有个问题，如果 `doSomething` 方法抛出了异常的话，从连接池中拿出来的 Jedis 对象将无法归还给连接池。如果这样的异常发生了好几次，连接池中的所有链接都被持久占用了，新的请求过来时就会阻塞等待空闲的链接，这样的阻塞一般会直接导致应用程序卡死。

为了避免这种情况的发生，程序员需要在使用 JedisPool 里面的 Jedis 链接时，应该使用 `try-with-resource` 语句来保护 Jedis 对象。

```java
import redis.clients.jedis.Jedis;
import redis.clients.jedis.JedisPool;

public class JedisTest {

  public static void main(String[] args) {
    JedisPool pool = new JedisPool();
    try (Jedis jedis = pool.getResource()) { // 用完自动 close
      doSomething(jedis);
    }
  }

  private static void doSomething(Jedis jedis) {
    // code it here
  }

}
```

这样 Jedis 对象肯定会归还给连接池 (死循环除外)，避免应用程序卡死的惨剧发生。

但是当一个团队够大的时候，并不是所有的程序员都会非常有经验，他们可能因为各种原因忘记了使用 `try-with-resource` 语句，惨剧就会突然冒出来让运维人员措手不及。我们需要在代码上加上一层硬约束，通过这层约束，当程序员想要访问 Jedis 对象时，不会再出现使用了 Jedis 对象而不归还。

```
import redis.clients.jedis.Jedis;
import redis.clients.jedis.JedisPool;

interface CallWithJedis {
  public void call(Jedis jedis);
}

class RedisPool {
```

```
  private JedisPool pool;

  public RedisPool() {
    this.pool = new JedisPool();
  }

  public void execute(CallWithJedis caller) {
    try (Jedis jedis = pool.getResource()) {
      caller.call(jedis);
    }
  }

}

public class JedisTest {

  public static void main(String[] args) {
    RedisPool redis = new RedisPool();
    redis.execute(new CallWithJedis() {

      @Override
      public void call(Jedis jedis) {
        // do something with jedis
      }

    });
  }

}
```

我们通过一个特殊的自定义的 RedisPool 对象将 JedisPool 对象隐藏起来，避免程序员直接使用它的 `getResource` 方法而忘记了归还。程序员使用 RedisPool 对象时需要提供一个回调类来才能使用 Jedis 对象。

但是每次访问 Redis 都需要写一个回调类，真是特别繁琐，代码也显得非常臃肿。幸好 Java8 带来了 Lambda 表达式，我们可以使用 Lambda 表达式简化上面的代码。

```
public class JedisTest {

  public static void main(String[] args) {
    Redis redis = new Redis();
    redis.execute(jedis -> {
      // do something with jedis
    });
  }
```

```
}
```

这样看起来就简洁优雅多了。但是还有个问题，Java 不允许在闭包里修改闭包外面的变量。比如下面的代码，我们想从 Redis 里面拿到某个 zset 对象的长度，编译器会直接报错。

java

```
public class JedisTest {

  public static void main(String[] args) {
    Redis redis = new Redis();
    long count = 0;
    redis.execute(jedis -> {
      count = jedis.zcard("codehole");  // 此处应该报错
    });
    System.out.println(count);
  }

}
```

编译器暴露出来的错误时： `Local variable count defined in an enclosing scope must be final or effectively final` ，告诉我们 count 变量必须设置成 final 类型才可以让闭包来访问。

如果这时我们将 count 设置成 final 类型，结果编辑器又报错了： `The final local variable count cannot be assigned. It must be blank and not using a compound assignment` ，告诉我们 final 类型的变量在闭包里面不能被修改。

那该怎么办呢?

这里需要定义一个 Holder 类型，将需要修改的变量包装起来。

java

```
class Holder<T> {
  private T value;

  public Holder() {
  }

  public Holder(T value) {
    this.value = value;
  }
```

```
    public void value(T value) {
      this.value = value;
    }

    public T value() {
      return value;
    }
  }

  public class JedisTest {

    public static void main(String[] args) {
      Redis redis = new Redis();
      Holder<Long> countHolder = new Holder<>();
      redis.execute(jedis -> {
        long count = jedis.zcard("codehole");
        countHolder.value(count);
      });
      System.out.println(countHolder.value());
    }

  }
```

有了上面定义的 Holder 包装类，就可以绕过闭包对变量修改的限制。只不过代码上要多一层略显繁琐的变量包装过程。这些都是对程序员的硬约束，他们必须这么做才可以得到自己想要的数据。

重试

我们知道 Jedis 默认没有提供重试机制，意味着如果网络出现了抖动，就会大范围报错，或者一个后台应用因为链接过于空闲被服务端强制关闭了链接，当重新发起新请求时就第一个指令会出错。而 Redis 的 Python 客户端 redis-py 提供了这种重试机制，redis-py 在遇到链接错误时会尝试进行重连，然后再重发指令。

那如果我们希望在 Jedis 上面增加重试机制，该如何做呢？有了上面的 RedisPool 对象，重试就非常容易进行了。

```
  class Redis {

    private JedisPool pool;
```

```
    public Redis() {
      this.pool = new JedisPool();
    }

    public void execute(CallWithJedis caller) {
      Jedis jedis = pool.getResource();
      try {
        caller.call(jedis);
      } catch (JedisConnectionException e) {
        caller.call(jedis);  // 重试一次
      } finally {
        jedis.close();
      }
    }

  }
```

上面的代码我们只重试了一次，如有需要也可以重试多次，但是也不能无限重试，就好比人逝不可复生，要节哀顺变。

作业

囿于精力，以上代码并没有做到非常细致，比如 Redis 的链接参数都没有提及，连接池的大小以及超时参数等也没有配置，这些细节工作就留给读者们作为本节的作业，自己动手完成一个完善的封装吧。