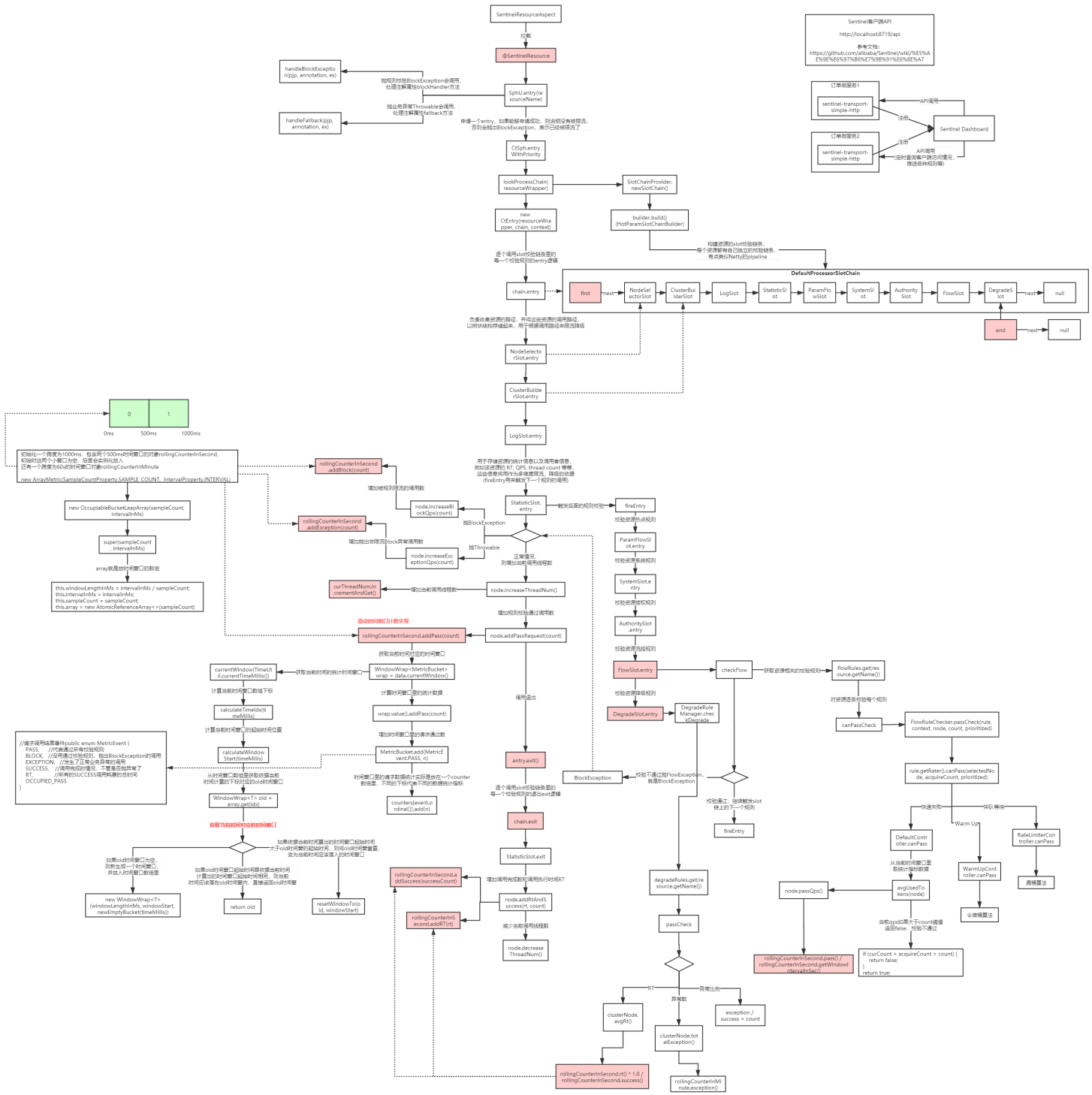
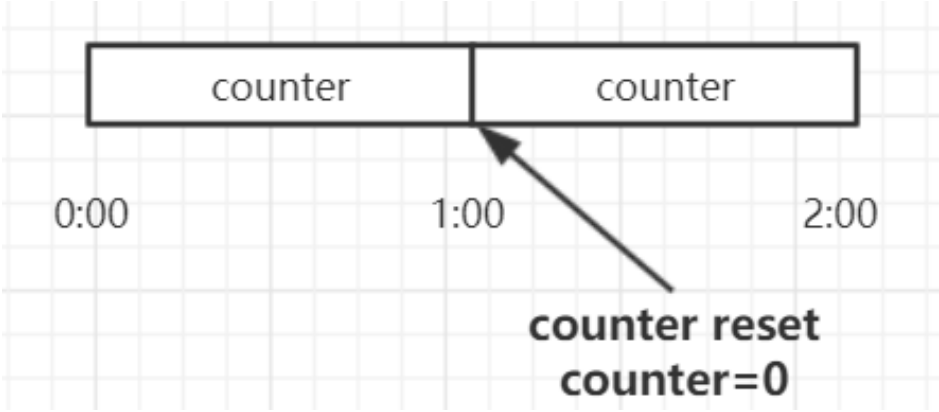
Sentinel限流、熔断降级源码走向图



常见限流算法精讲

计数器法

计数器法是限流算法里最简单也是最容易实现的一种算法。比如我们规定，对于A接口来说，我们1分钟的访问次数不能超过100个。那么我们可以这么做：在一开始的时候，我们可以设置一个计数器counter，每当一个请求过来的时候，counter就加1，如果counter的值大于100并且该请求与第一个 请求的间隔时间还在1分钟之内，那么说明请求数过多；如果该请求与第一个请求的间隔时间大于1分钟，且counter的值还在限流范围内，那么就重置 counter。



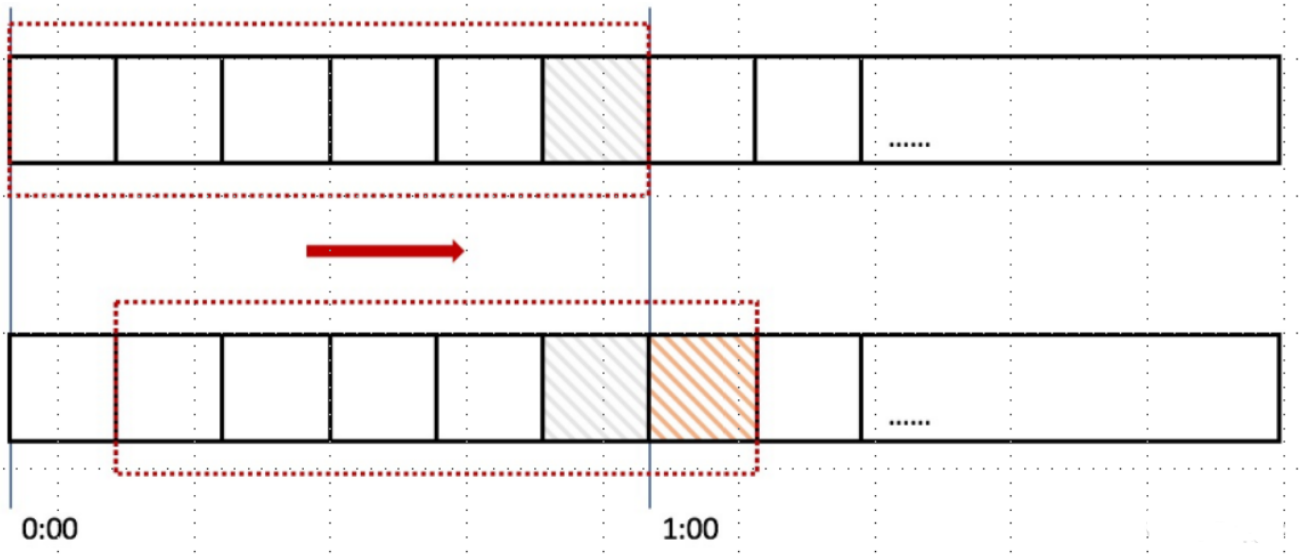
具体算法的伪代码：

```
/**
 * 最简单的计数器限流算法
 */
public class Counter {
    public long timeStamp = System.currentTimeMillis();  // 当前时间
    public int reqCount = 0;  // 初始化计数器
    public final int limit = 100; // 时间窗口内最大请求数
    public final long interval = 1000 * 60; // 时间窗口ms

     public boolean limit() {
         long now = System.currentTimeMillis();
         if (now < timeStamp + interval) {
             // 在时间窗口内
             reqCount++;
             // 判断当前时间窗口内是否超过最大请求控制数
             return reqCount <= limit;
         } else {
             timeStamp = now;
             // 超时后重置
             reqCount = 1;
             return true;
         }
     }
}
```

滑动时间窗口算法

滑动时间窗口，又称rolling window。为了解决计数器法统计精度太低的问题，引入了滑动窗口算法。下面这张图，很好地解释了滑动窗口算法：



在上图中，整个红色的矩形框表示一个时间窗口，在我们的例子中，一个时间窗口就是一分钟。然后我们将时间窗口进行划分，比如图中，我们就将滑动窗口划成了6格，所以每格代表的是10秒钟。每过10秒钟，我们的时间窗口就会往右滑动一格。每一个格子都有自己独立的计数器counter，比如当一个请求 在0:35秒的时候到达，那么0:30~0:39对应的counter就会加1。

计数器算法其实就是滑动窗口算法。只是它没有对时间窗口做进一步地划分，所以只有1格。

由此可见，当滑动窗口的格子划分的越多，那么滑动窗口的滚动就越平滑，限流的统计就会越精确。

具体算法的伪代码：

```
/**
 * 滑动时间窗口限流实现
 * 假设某个服务最多只能每秒钟处理100个请求，我们可以设置一个1秒钟的滑动时间窗口，
 * 窗口中有10个格子，每个格子100毫秒，每100毫秒移动一次，每次移动都需要记录当前服务请求的次数
 */
public class SlidingTimeWindow {
    //服务访问次数，可以放在Redis中，实现分布式系统的访问计数
    Long counter = 0L;
    //使用LinkedList来记录滑动窗口的10个格子。
     LinkedList<Long> slots = new LinkedList<Long>();

     public static void main(String[] args) throws InterruptedException {
         SlidingTimeWindow timeWindow = new SlidingTimeWindow();

         new Thread(new Runnable() {
             @Override
             public void run() {
                 try {
                     timeWindow.doCheck();
                 } catch (InterruptedException e) {
                     e.printStackTrace();
                 }
             }
         }).start();

         while (true){
```

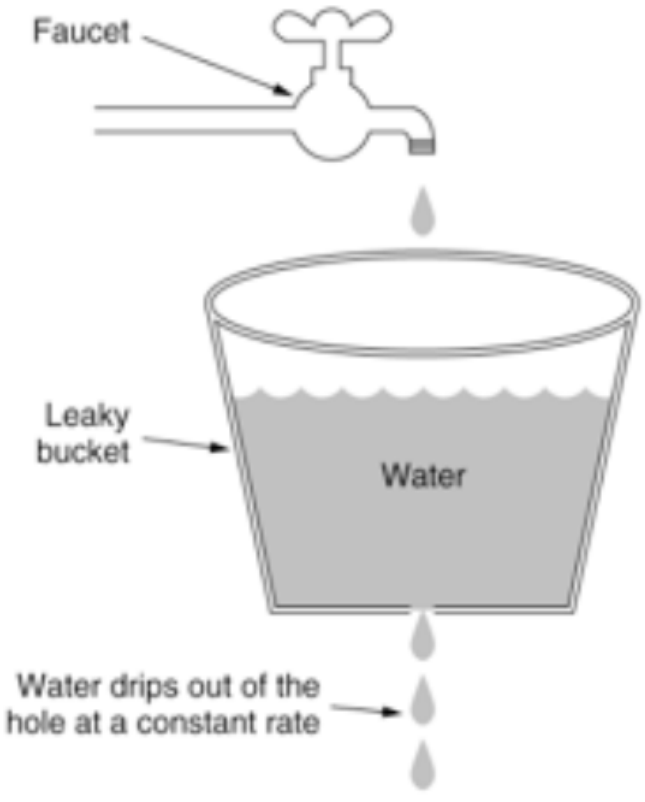
```
            //TODO 判断限流标记
            timeWindow.counter++;
            Thread.sleep(new Random().nextInt(15));
        }
    }

    private void doCheck() throws InterruptedException {
        while (true) {
            slots.addLast(counter);
            if (slots.size() > 10) {
                slots.removeFirst();
            }
            //比较最后一个和第一个，两者相差100以上就限流
            if ((slots.peekLast() - slots.peekFirst()) > 100) {
                System.out.println("限流了。。");
                //TODO 修改限流标记为true
            }else {
                //TODO 修改限流标记为false
            }

            Thread.sleep(100);
        }
    }
}
```

漏桶算法

漏桶算法，又称leaky bucket。



从图中我们可以看到，整个算法其实十分简单。首先，我们有一个固定容量的桶，有水流进来，也有水流出去。对于流进来的水来说，我们无法预计一共有多少水会流进来，也无法预计水流的速度。但是对于流出去的水来说，这个桶可以固定水流出的速率。而且，当桶满了之后，多余的水将会溢出。

我们将算法中的水换成实际应用中的请求，我们可以看到漏桶算法天生就限制了请求的速度。当使用了漏桶算法，我们可以保证接口会以一个常速速率来处理请求。所以漏桶算法天生不会出现临界问题。

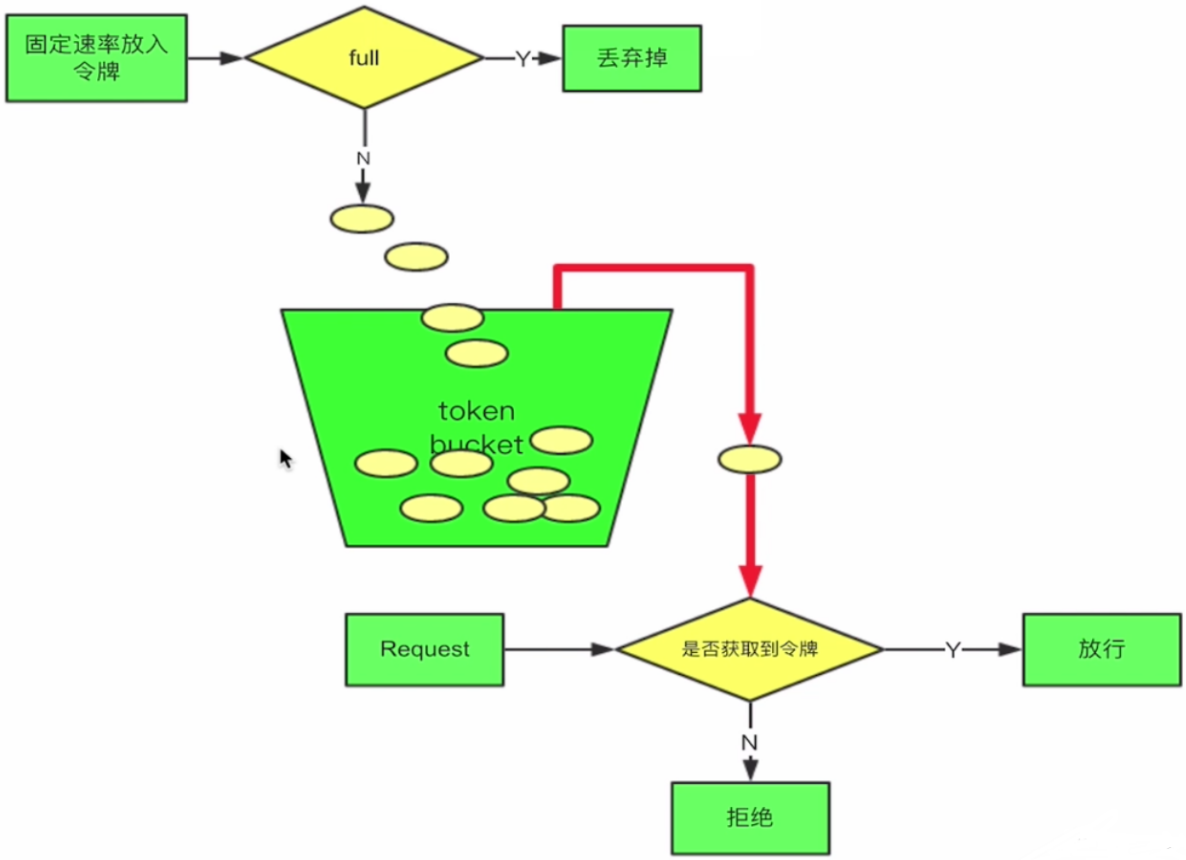
具体的伪代码如下：

```
/**
 * 漏桶限流算法
 */
public class LeakyBucket {
        public long timeStamp = System.currentTimeMillis();  // 当前时间
        public long capacity; // 桶的容量
        public long rate; // 水漏出的速度(每秒系统能处理的请求数)
        public long water; // 当前水量(当前累积请求数)

         public boolean limit() {
             long now = System.currentTimeMillis();
             water = Math.max(0, water - ((now - timeStamp)/1000) * rate); // 先执行漏水，计
             timeStamp = now;
             if ((water + 1) < capacity) {
                 // 尝试加水,并且水还未满
                 water += 1;
                 return true;
             } else {
                 // 水满，拒绝加水
                 return false;
         }
     }
}
```

令牌桶算法

令牌桶算法，又称token bucket。同样为了理解该算法，我们来看一下该算法的示意图：



从图中我们可以看到，令牌桶算法比漏桶算法稍显复杂。首先，我们有一个固定容量的桶，桶里存放着令牌（token）。桶一开始是空的，token以 一个固定的速率r往桶里填充，直到达到桶的容量，多余的令牌将会被丢弃。每当一个请求过来时，就会尝试从桶里移除一个令牌，如果没有令牌的话，请求无法通过。

具体的伪代码如下：

```
/**
 * 令牌桶限流算法
 */
public class TokenBucket {
    public long timeStamp = System.currentTimeMillis();  // 当前时间
    public long capacity; // 桶的容量
    public long rate; // 令牌放入速度
    public long tokens; // 当前令牌数量

     public boolean grant() {
        long now = System.currentTimeMillis();
        // 先添加令牌
        tokens = Math.min(capacity, tokens + (now - timeStamp) * rate);
        timeStamp = now;
        if (tokens < 1) {
            // 若不到1个令牌,则拒绝
            return false;
        } else {
            // 还有令牌，领取令牌
            tokens -= 1;
            return true;
        }
    }
```

```
24  }
```

限流算法小结

计数器 VS 滑动窗口：

```
1  计数器算法是最简单的算法，可以看成是滑动窗口的低精度实现。
2  滑动窗口由于需要存储多份的计数器（每一个格子存一份），所以滑动窗口在实现上需要更多的存储空间。
3  也就是说，如果滑动窗口的精度越高，需要的存储空间就越大。
```

漏桶算法 VS 令牌桶算法：

```
1  漏桶算法和令牌桶算法最明显的区别是令牌桶算法允许流量一定程度的突发。
2  因为默认的令牌桶算法，取走token是不需要耗费时间的，也就是说，假设桶内有100个token时，那么可以瞬间
3  当然我们需要具体情况具体分析，只有最合适的算法，没有最优的算法。
```

文档：06-Sentinel限流、熔断降级源码剖析

链接：http://note.youdao.com/noteshare?id=19206aab917935d3a101fa94d2b68700&sub=E2FAAB34BDBA4CA5AA8FF90217407055