韧性策略在线适应何时会适得其反：Resilience4j 中的安全探索与测量陷阱

Rajat M. Barot 和 Arjun S. Borkhatariya

arXiv:2611.32015v1 [cs.CR] 24 Nov 2026

摘要—**Java 微服务中的断路器、重试和舱壁设置通常只调整一次便固定下来，尽管最佳设置取决于依赖项当前呈现的故障。我们研究在线学习器能否在运行时选择这些设置。AdaptR4j 可在 Spring Boot 3 中原子地切换 Resilience4j 策略，并让上下文老虎机根据 Prometheus 信号在完整的 CircuitBreaker/Retry/Bulkhead 配置中进行选择。**

我们的首个朴素设计在其优化的奖励指标下，在四个可胜故障场景中的三个场景里，表现不如保持默认策略：包含激进断路器的策略菜单会拒绝大多数流量，而此时学习器仍在学习。两项安全防护措施解决了这一问题：安全策略菜单（没有任何断路器比延迟服务等级目标更严格）以及带有界奖励的安全探索。采用这些措施后，AdaptR4j 在每个可胜场景中都优于生产默认策略（$p < 0.001$，每个场景运行三次，运行间标准差为 ≤ 0.03）；当故障随时间变化时，它优于最佳固定策略，并且与朴素设计相比，将用户可见故障减少了最多 3.7× 。在相同的安全策略菜单上，无上下文老虎机的表现从统计上看没有显著差异，因此，主要带来收益的是安全防护措施，而不是学习器。

我们还报告了测试框架早期版本产生的一个看似可信但实际上错误的结果：单位不匹配使每个断路器 100× 过于严格，而被拒绝的调用未被指标捕获，因此 94% 的用户请求失败了，但延迟和错误指标看起来却非常理想。

索引词—**断路器、重试、舱壁、Resilience4j、上下文多臂老虎机、安全探索、性能测量、微服务。**

I. 引言

Resilience4j [1] 是 Java 微服务的标准容错库。它的断路器（CB）、重试和舱壁模式会相互影响：重试会加重性能降级的依赖项所承受的负载，过于严格的断路器会将短暂的错误激增变成全面中断，而容量较小的舱壁会在负载较高时拒绝健康流量 [2], [3]。团队通常只在 YAML 中设置一次这些参数，之后便不再调整；然而，最佳设置取决于依赖项当前是响应缓慢、发生故障还是运行正常。服务网格层 [4], [5]已有自适应方案，并且针对并发限制等单一参数 [6], [7], 也有自适应方案，但应用内部尚无能够选择整套 Resilience4j 配置的方案。

（通讯作者：Rajat M. Barot。）Rajat M. Barot 就职于纽约州立大学宾厄姆顿分校计算机科学系，地址：美国纽约州宾厄姆顿。电子邮箱：rbarot2@binghamton.edu Arjun S. Borkhatariya 就职于亚利桑那州立大学软件工程系，地址：美国亚利桑那州坦佩。电子邮箱：aborkhat@asu.edu 代码和数据：https://github.com/rajatbarot24/adaptr4j。

我们构建了 AdaptR4j，以检验在线学习器能否做到这一点。本文介绍了这一尝试的进展，包括遇到的问题：

- **朴素的在线适应可能比什么都不做更糟**。一个在包含 30 项策略的策略菜单上运行的 LinUCB 多臂老虎机 [8] 在四个可胜场景中的三个场景中输给了生产默认策略，因为探索过程反复应用了几乎拒绝全部流量的断路器（§V-A）。
- **两项简单的安全防护措施使其稳定地发挥优势**。仅配置规则即可限定安全策略菜单，而采用有界奖励的安全探索则让 AdaptR4j 在所有能够击败默认策略的场景中都胜过默认策略，并在时变故障下胜过最佳修正后策略（§V）。
- **学习器的重要性低于安全防护措施**。在相同安全策略菜单上运行的无上下文 UCB1 多臂老虎机与 AdaptR4j 在统计上无显著差异；上下文特征在一种场景中有所帮助，但在另一种场景中反而有害（§V-C）。
- **测量结果可能颠倒结论**。我们的第一个测试框架报告了出色的延迟和错误率，但实际上 94% 的用户请求都失败了（§IV）。我们描述了导致这一结果的两个错误，以及能够发现它们的检查措施。

II. ADAPTR4J

策略原子切换。Spring Boot 3 的配置刷新在运行时不会传递到 Resilience4j 注册表实例。策略持有器通过原子引用保存活动的（熔断器、重试、舱壁）三元组；每次调用都按照 Resilience4j 的标准顺序 Retry(CB(Bulkhead(call)))装饰下游调用，而切换操作则通过一次原子写入发布新构建的实例。已经在执行中的调用会在旧实例上完成。重试不会重试库自身的拒绝（断开的断路器、已满的舱壁）。

控制器。 控制器每 60 秒读取六项 Prometheus 特征（边缘端超过服务等级目标的 p99 和 p50、错误率、重试放大因子 RAF、熔断器开启标志、归一化吞吐量），使用 LinUCB [8]，选择一个策略并将其推送到服务。前五个时间窗口处于影子模式（仅观测）。某个时间窗口的奖励为

$$r = \begin{cases} -2 & \text{if } p99 > 500\,\text{ms}, \\ \max(-2,\, u) & \text{otherwise}, \end{cases}$$

$$u = 1 - \frac{p99}{500\,\text{ms}} - \frac{\text{err}}{0.05} - 0.5\max(0, RAF - 1),$$

因此，延迟余量最高可获得 $+1$ 分，达到 5% 服务等级目标的错误率会扣除整整 1 分，而没有任何窗口比违反服务等级目标更糟糕

。延迟在边缘端进行测量（即用户发出的请求），任何原因导致的失败调用——包括熔断器和舱壁拒绝——都计为错误（§IV）。

护栏机制。当重试次数超过请求速率的两倍时，RAF 硬停止会冻结策略；如果连续三个时间窗口的奖励为 < -1，则回滚会切换至迄今为止表现最佳的策略。

朴素设计失败后添加的安全防护措施（§V-A）。安全策略菜单：我们只保留熔断器故障率阈值至少为 45%、且慢调用时长超过 500 毫秒延迟服务等级目标的策略。更严格的断路器会将符合服务等级目标的调用或普通错误突发都计为故障，并在流量健康时打开。该规则从菜单中的 30 个策略里保留了 14 个，其中包括每种场景下的最佳固定策略。安全探索：如果某个策略近期的平均奖励比最佳策略低 1.0 以上，则在经过 30 个时间窗口的休整前不会再次选中它，因此在故障变化后仍可重新测试。这是一种简单的保守探索形式 [9], [10]。有界奖励：此前，错误项没有上界，一个完全拒绝请求的时间窗口得分约为 -20，压过了所有平均值。

三、实验设置

测试平台。服务A（边缘端，部署AdaptR4j的切换机制）调用服务B；服务B的延迟和错误率通过HTTP端点在运行时设置。两者均采用 Spring Boot 3.3.5 / Resilience4j 2.2.0，运行于 Java 21，并部署在同一台笔记本电脑上（Intel i5-8250U、4 核、8 GB 内存、Windows 11）。Prometheus 2.52 每 15 秒抓取一次数据。Locust 2.43 驱动 50 个用户，请求间思考时间为 0.5–2 秒；每次运行前先发送 200 个预热请求。每次运行从两个服务全新启动开始，持续 80 个一分钟窗口。

故障场景。S1 中等故障（50 ms，4% 错误）；S2 严重故障（200 ms，10%）；S3 纯延迟故障（500 ms，0%）；S4 错误风暴（0 ms，30%）；S5 延迟尖峰（每 5 分钟出现一次 $0 \leftrightarrow 500$ ms）。服务等级目标为 p99 ≤ 500 ms，错误率为 ≤ 5 %。S3 无法达标：每次调用本身就需要 500 ms，因此所有策略都会错过延迟服务等级目标。我们保留它作为合理性检查。

基线方法。默认策略：菜单中的均衡策略 p01（3 次尝试、等待 500 毫秒、熔断器 50%、40 个并发调用）在整个运行期间始终生效。最佳固定策略（预言机）：进行 65 个时间窗口的扫描，每种菜单策略应用两个时间窗口；然后在事后选出表现最佳的策略，运行 80 个时间窗口。*UCB1* [11] 在安全策略菜单上运行，不使用上下文。无上下文：使用恒定上下文向量的 AdaptR4j。朴素设计：AdaptR4j 使用包含 30 项的完整菜单，不进行安全探索，并使用有界奖励重新评分。

统计。一次运行中的连续时间窗口存在自相关，因此我们使用移动区块自助法（区块长度为 10 个时间窗口，重抽样 5000 次），根据平均奖励之差比较各次运行。每个可胜场景运行 AdaptR4j 三次；基线方法各运行一次。

IV. 测量如何误导了我们

该研究的第一版报告了$+0.72$ （S1）和 $+0.81$ （S2）的平均奖励、相较于默认策略降低 9–12$\times$ 的 p99，以及 0.1% 的错误率。将控制器的视角与负载生成器的视角进行比较后发现，**94.5%（S1）且 97.4 %（S2）的用户请求均已失败**，返回 HTTP 500错误（表 I）。两个错误叠加在一起：

表 I 控制器所见与用户体验（AdaptR4j；每窗口 p99 和错误率为整个运行期间的平均值；用户使用 = Locust 失败，统计整个运行）。

测试框架	场景	观测到的 p99	观测到的错误	用户失败
原始	S1	138 毫秒	0.10%	**94.5%**
原始	S2	106 毫秒	0.06%	**97.4%**
修正后	S1	280 毫秒	0.12%	0.1%
修正后	S2	854 毫秒	3.64 %	6.0 %

1. 单位不匹配。策略菜单将断路器阈值表示为小数（0.50）；而 Resilience4j 需要百分比。因此，每个策略实际都以 0.2–0.8% 的阈值运行，并且每个断路器都在首次注入错误时打开。唯一能良好服务用户的配置是一个准入控制基线，恰好将其阈值以 90.0 的形式发送。

2. 不可见的拒绝。延迟和错误是在韧性链内部的下游调用上记录的。被已断开的断路器拒绝的调用根本不会到达那里：它们增加的耗时几乎为零，从而拉低了 p99，而且也从未被计为错误。因此，奖励更倾向于拒绝最多流量的策略。

另外三个较小的缺陷让情况雪上加霜：不规范的装饰器顺序（舱壁在每次重试退避期间都一直占用其槽位）、一种会悄然覆盖所请求用户数量的负载生成器形状类别，以及一次将影子模式时间窗口也计入的回滚。修复所有这些问题后，在验证运行中，控制器的错误率与用户的失败率平均相差约 0.1 个百分点。

我们现在采用的检查措施。(1) 在用户看到的最外层进行测量；统计每个异常，包括库自身拒绝的请求。(2) 每次运行都自动将控制器指标与负载生成器进行交叉核对。(3) 每次策略变更后，从库中读取实时配置（/actuator/circuitbreakers）。(4) 对于看起来过于 理想的结果——在注入延迟时 p99 接近零——在查明原因之前，都应视为漏洞。

V. 结果

A. 朴素设计不如默认策略

在修正后的测试框架上，朴素设计（包含完整的 30 项菜单）在 S1 中得分为 $+0.44$，但在 S2、S4 和 S5 中分别为 -1.86、-0.85 和 -0.76 （表 II）。使用原始无界奖励时，它在 S2、S4 和 S5 中的表现都不如默认 p01。扫描结果解释了原因：30 项策略中有 16 项会导致至少一个场景中超过半数的请求失败，而且这全都是严格断路器造成的；探索过程还不断回到这些策略。

B. 安全设计

可复现。在三次独立运行中，平均奖励的标准差最多为 0.026，且每次运行中出现频率最高的策略都相同（S1 和 S4 中为 p24，S2 中为 p04，S5 中为 p21）。在 S1 和 S4 中，策略最终稳定在 p24（84% 和

表 II 平均奖励（越高越好； -2 = 服务等级目标在每个时间窗口均未达成）和用户可见故障率（Locust：HTTP 5XX 或响应 >1 秒）。AdaptR4j：三次运行，平均值 ± 标准差；“延迟” = 时间窗口为 40–80。 †：区块自助法 $p<0.05$ 与 AdaptR4j 第 1 次运行的比较。

场景	AdaptR4j（安全设计）			默认 p01		最佳固定策略		UCB1，安全策略菜单		朴素设计	
	奖励	late	失败率	奖励	失败率	奖励	失败率	奖励	失败率	奖励	失败率
S1	$+0.257 \pm 0.002$	$+0.518$	0.1	$-2.000^{\dagger}$	0.1	$+0.518$	0.0	$+0.050$	0.8	$+0.441$	0.3
S2	-1.821 ± 0.021	-1.759	5.8	$-2.000^{\dagger}$	1.0	$-1.438^{\dagger}$	9.4	-1.823	4.0	-1.862	9.4
S3	-2.000 （1 次运行）	-2.000	0.8	-2.000	0.0	-2.000	0.0	–	–	-2.000	10.6
S4	-0.315 ± 0.010	-0.035	9.4	$-2.000^{\dagger}$	32.5	$+0.249^{\dagger}$	2.6	-0.526	14.0	-0.848	31.4
S5	-0.451 ± 0.026	-0.409	2.3	$-0.842^{\dagger}$	0.0	$-0.763^{\dagger}$	0.0	-0.603	0.4	-0.764	8.5

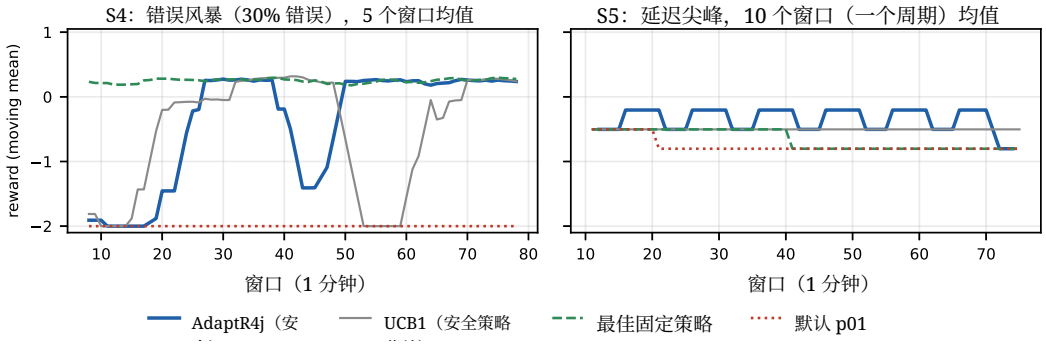

图1. S4 和 S5 中奖励随时间的变化（移动平均），第 1 次运行。在 S4 中，AdaptR4j 在约 25 个时间窗口后达到最佳固定策略的水平；在故障交替变化的 S5 中，它始终优于所有固定策略。整个 S4 期间，默认 p01 保持在 -2 。

74% 的时间窗口）；在 S2 和 S5 中，策略则持续切换，因为没有哪一种策略明显更优。

在适用的所有情况下都优于默认策略。AdaptR4j 在 S1、S2、S4 和 S5 中均优于 p01（各为 $p<0.001$ ）。默认策略在 S1、S2 和 S4 的每个时间窗口都未达到延迟服务等级目标，因为其重试等待时间使 p99 升至 0.7–1.1 秒。它在 S2 和 S5 中的用户可见故障确实更少：它以延迟换取成功率，而奖励会同时权衡两者（§VII）。

故障变化时优于最佳固定策略。在 S5 中，延迟每五分钟交替变化，没有任何固定策略适合这两个阶段。AdaptR4j 的得分为 -0.451 ，优于最佳固定策略（$p=0.011$）的 -0.763 ，也优于默认策略（图1右侧）的 -0.842 。

在稳态故障下，缓慢逼近最佳固定策略。在 S1 中，AdaptR4j 最终选定最佳固定策略 p24，并在时间窗口 40 后与之完全匹配（$+0.518$）；但先尝试全部 14 种策略会降低奖励：其整体均值（$+0.257$）低于朴素设计（$+0.441$）。后者碰巧只尝试了五种策略，便选定了一个不错的策略（时间窗口 10，而非 19）。在 S4 中，它约在 25 个时间窗口后达到最佳固定策略的水平（图1，左）；时间窗口 41–47 的下滑，是对约 30 个时间窗口前被排除策略的定期重新测试。在 S2 中，最佳固定策略仍明显更优（-1.44 对比 -1.82）。

用户可见故障少于朴素设计：S4 中为 9.4 % 对 31.4 %，S5 中为 2.3 % 对 8.5%，S2 中为 5.8 % 对 9.4 %。

C. 安全防护措施与学习器

在相同的安全策略菜单上，不使用任何上下文的 UCB1 在 S1、S2、S4 和 S5 中的得分分别为$+0.050$、 -1.823、 -0.526和 -0.603 。AdaptR4j 的平均值分别高出 0.21、0.00、0.21 和 0.15，但根据区块自助法，这些差异均不显著（$p=0.34$，0.79、0.47、0.17）：奖励在不同时间窗口之间波动很大，而且每种基线都只运行了一次。相比之下，安全策略菜单对每个学习器都有帮助：从完整菜单切换到安全策略菜单时，UCB1 在 S1 中的表现从 -0.353 提升至 $+0.050$ 。上下文特征的作用也不明确：移除这些特征会降低 S1 中的平均值（$+0.186$对比 $+0.257$），但会提高 S4 中的平均值（-0.140 对比 -0.315）。因此，我们的证据支持安全防护措施是改进的来源，而非上下文学习。

VI. 相关工作

Sedghpour等人表明，不存在一种适用于所有工作负载且始终占优的静态断路器设置 [2]，并通过规则 [4]调整服务网格断路器；Sun等人在Istio中使用深度Q学习 [5]。RetryGuard [12] 通过分析模型防止重试风暴。Netflix的并发限制 [6]，、Envoy 的自适应并发过滤器 [7]，、客户端限流 [13],以及TopFull [14] 和Rajomon [15] 等准入控制机制，都会调整单个控制参数。Autothrottle [16] 和OPPerTune [17] 在线学习资源和配置设置。相比之下，AdaptR4j在应用程序内部，以MAPE-K循环的形式选择完整的Resilience4j配置 [18], [19]。我们的首个负面结果印证了安全强化学习文献 [10], [9]中的观点：在真实运行的系统上进行无约束探索，其代价可能超过适应机制所能挽回的损失。

VII. 尚待解决的问题与后续步骤

该菜单规则是在事后发现的。我们根据用于评估的同一组扫描结果推导出了安全菜单规则。它只使用操作员预先知晓的配置值，但必须在新的服务和故障类型上进行验证。

探索成本。 将每种安全策略都尝试一次需要 15–20 个时间窗口；S1 展示了这一代价。显而易见的下一步，是用默认策略的观测奖励或离线扫描结果为多臂老虎机提供初始数据。

奖励应如何衡量？在 S2 和 S5 中，默认策略成功处理的请求更多，但速度较慢；我们的奖励函数更偏好快速的部分故障。正确的权衡取决于服务级决策，因此应当支持配置。

S2 和 S3 中不存在可行策略。 当没有任何菜单策略能够满足服务等级目标时，控制器只能选择最不差的方案；生成菜单时应涵盖此类情形。

有效性威胁。 所有运行都在一台笔记本电脑上进行，使用了合成故障和单一工作负载；基线方法只运行了一次；我们用朴素设计进行的扇出、漂移和网络延迟实验，尚未用安全设计重复进行。

可复现性与可用性。 代码、策略菜单、所有决策日志和负载生成器输出，以及生成本文每个数字的脚本，均可在 https://github.com/rajatbarot24/adaptr4j 获取，并采用 MIT许可证。

VIII. 结论

只有在探索变得安全之后，Resilience4j 策略的在线选择才有益处。在具备安全策略菜单、安全探索和有界奖励的情况下，简单多臂老虎机在我们测试的每个可胜场景中都胜过生产默认策略；当故障随时间变化时，它还胜过最佳固定策略；而没有这些条件时，它甚至不如什么都不做。equally重要的是，我们自己的首次评估表明，韧性实验很容易在大多数用户失败的情况下，仍然报告出优秀的指标。

参考文献

[1] R. Winkler 等，“Resilience4j：轻量级容错库”，https://github.com/resilience4j/resilience4j，2024年。[2] M. R. S. Sedghpour、C. Meiklejohn 和 P. Leitner，“带服务网格的微服务系统中的性能反模式实证研究”，载于*ACM/SPEC*国际性能工程会议（*ICPE*）论文集。美国纽约州纽约市：ACM，2022年，第45–56页。[3] C. Meiklejohn、R. Padhye 和 M. Sevitsky，“方法重载电路”，载于*ACM*云计算研讨会（*SoCC*）论文集。美国纽约州纽约市：ACM，2022年，第1–14页。[4] M. R. S. Sedghpour、D. Garlan 和 P. Leitner，“打破恶性循环：面向韧性模式的自适应协调”，载于*IEEE*国际云工程会议（*IC2E*）论文集。美国马萨诸塞州波士顿：IEEE，2023年，第1–10页。[5] J. Sun、W. Li 和 H. Zheng，“基于深度Q学习的微服务系统熔断”，载于计算机与信息科学通信（*CCIS*）。瑞士钱：Springer，2024年，第1–15页。[6] Netflix，“concurrency-limits：实现并集成TCP拥塞控制概念、自动检测并发限制的Java库”，https://github.com/Netflix/concurrency-limits，2018年。[7] Envoy项目，“自适应并发HTTP过滤器”，https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/adaptive_concurrency_filter，2020年。[8] L. Li、W. Chu、J. Langford 和 R. E. Schapire，“个性化新闻文章推荐的上下文老虎机方法”，载于国际万维网会议（*WWW*）论文集。美国纽约州纽约市：ACM，2010年，第661–670页。

[9] Y. Wu、R. Shariff、T. Lattimore 和 C. Szepesv´ari，“保守型多臂老虎机”，载于第*33*届国际机器学习会议（*ICML*）论文集，2016年，第1254–1262页。[10] J. Garc´ıa 和 F. Fern´andez，“安全强化学习综述”，机器学习研究期刊，第16卷，第1437–1480页，2015年。[11] P. Auer、N. Cesa-Bianchi 和 P. Fischer，“多臂老虎机问题的有限时间分析”，《机器学习》，第47卷，第2–3期，第235–256页，2002年。[12] J. Tavori、A. Bremler-Barr、H. Levy 和 O. Lavi，“RetryGuard：防止云微服务应用中的自发重试风暴”，*arXiv*预印本 *arXiv:2511.23278*，2025年。[13] B. Beyer、C. Jones、J. Petoff 和 N. R. Murphy 编，《站点可靠性工程：谷歌如何运行生产系统》。O’Reilly Media，2016年，第21章：过载处理。[14] J. Park、J. Park、Y. Jung、H. Lim、H. Yeo 和 D. Han，“TopFull：面向服务级别目标的微服务的自适应自顶向下过载控制”，载于*ACM SIGCOMM 2024*会议论文集。ACM，2024年，第876–890页。[15] M. Wang、Z. Tan、J. Zhang、A. Akella 和 M. Swift，“Rajomon：微服务中过载管理的协调式透明准入控制”，载于*USENIX*网络系统设计与实现研讨会（*NSDI*）论文集。美国宾夕法尼亚州费城：USENIX，2025年，第1–18页。[16] Z. Wang、P. Li、C.-J. M. Liang、F. Wu 和 F. Y. Yan，“Autothrottle：面向服务级别目标的微服务资源管理实用双层方法”，载于第*21*届*USENIX*网络系统设计与实现研讨会（*NSDI*）论文集。USENIX，2024年，第1–18页。[17] G. Somashekar、K. Tandon、A. Kini 等，“OPPerTune：轻松实现服务的部署后配置调优”，载于第*21*届*USENIX*网络系统设计与实现研讨会（*NSDI*）论文集。USENIX，2024年，第1–18页。[18] J. O. Kephart 和 D. M. Chess，“自主计算的愿景”，《计算机》，第36卷，第1期，第41–50页，2003年。[19] D. Garlan、S.-W. Cheng、A.-C. Huang、B. Schmerl 和 P. Steenkiste，“Rainbow：基于架构的自适应与可复用基础设施”，《计算机》，第37卷，第10期，第46–54页，2004年。