Keynote: Performance

«Что в имени тебе моём?»

Aleksey Shipilёv
aleksey@shipilev.net, @shipilev
@shipilev

Safe Harbor / Тихая Гавань

Anything on this or any subsequent slides may be a lie. Do not base your decisions on this talk. If you do, ask for professional help.

Всё что угодно на этом слайде, как и на всех следующих, может быть враньём. Не принимайте решений на основании этого доклада. Если всё-таки решите принять, то наймите профессионалов.

redhat.

Крупно

Крупно: Критерии успеха в разработке

redhat.

Крупно: Критерии успеха в разработке

redhat.

Крупно: Критерии успеха в разработке

redhat.

Крупно: Критерии успеха в разработке

2. Корректность реализации

redhat.

Крупно: Критерии успеха в разработке

2. Корректность реализации
3. Безопасность

redhat.

Крупно: Критерии успеха в разработке

1. Соответствие получившегося желаниям пользователя
2. Корректность реализации
3. Безопасность

redhat

Крупно: Критерии успеха в разработке

1. Соответствие получившегося желаниям пользователя
2. Корректность реализации
3. Безопасность
4. Быстрота и удобство разработки

redhat.

Крупно: Критерии успеха в разработке

1. Соответствие получившегося желаниям пользователя
2. Корректность реализации
3. Безопасность
4. Быстрота и удобство разработки
5. Производительность

redhat.

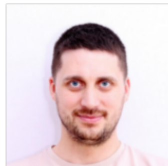
Крупно: Критерии успеха в разработке

1. Соответствие получившегося желаниям пользователя
2. Корректность реализации
3. Безопасность
4. Быстрота и удобство разработки
5. Производительность

Чаще всего производительность даже близко не главный приоритет.
А чаще всего её даже в критериях успеха нет.

redhat.

Крупно: Их разыскивает милиция



Siemargl 30 августа 2016 в 18:37 #

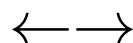
То есть из четырех экспертов никто не оценил Java, как быструю. Скорее, как достаточную и удовлетворяющую. Это вполне себе показательно.

redhat.

Крупно: Про критерии

«Корректная программа»:

«Быстрая программа»:



redhat.

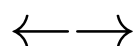
Крупно: Про критерии

«Корректная программа»:

1. не видно бесящих пользователя ошибок

«Быстрая программа»:

1. не видно бесящих пользователя тормозов

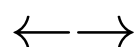


redhat.

Крупно: Про критерии

«Корректная программа»:

1. не видно бесящих пользователя ошибок
2. в критерии успеха вложились



«Быстрая программа»:

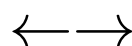
1. не видно бесящих пользователя тормозов
2. в критерии успеха вложились

redhat.

Крупно: Про критерии

«Корректная программа»:

1. не видно бесящих пользователя ошибок
2. в критерии успеха вложились
3. количество багов известно



«Быстрая программа»:

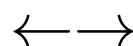
1. не видно бесящих пользователя тормозов
2. в критерии успеха вложились
3. перформансные проблемы известны

redhat.

Крупно: Про критерии

«Корректная программа»:

1. не видно бесящих пользователя ошибок
2. в критерии успеха вложились
3. количество багов известно
4. пути обхода и альтернативы известны

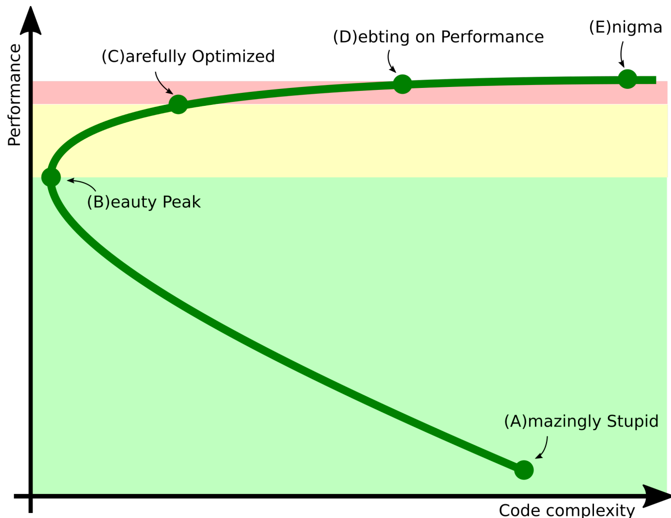


«Быстрая программа»:

1. не видно бесящих пользователя тормозов
2. в критерии успеха вложились
3. перформансные проблемы известны
4. пути обхода и альтернативы известны

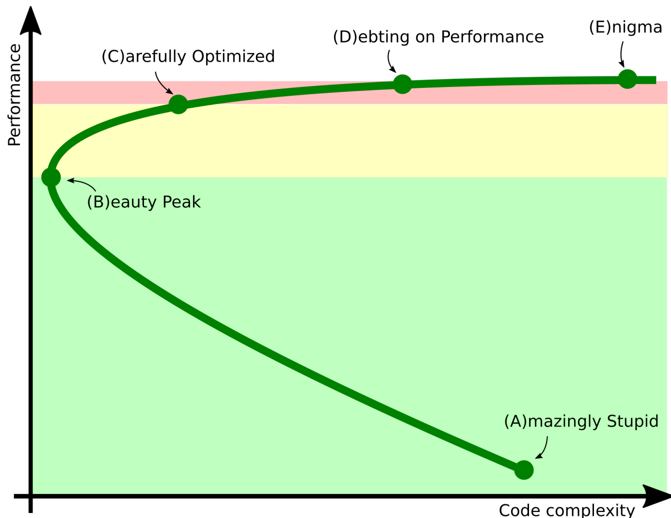
redhat.

Крупно: Куда кривая вывезет



redhat.

Крупно: Куда кривая вывезет

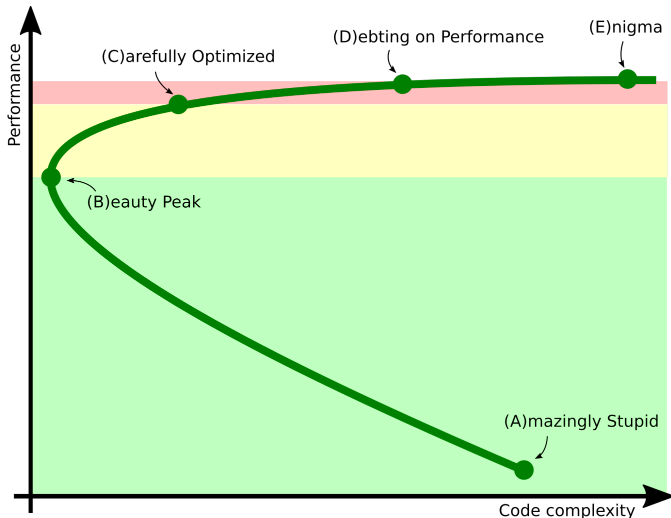


Вы в зелёной зоне:

Берёте профайлер и переписываете куски, которые написаны очевидно ужасно

redhat.

Крупно: Куда кривая вывезет

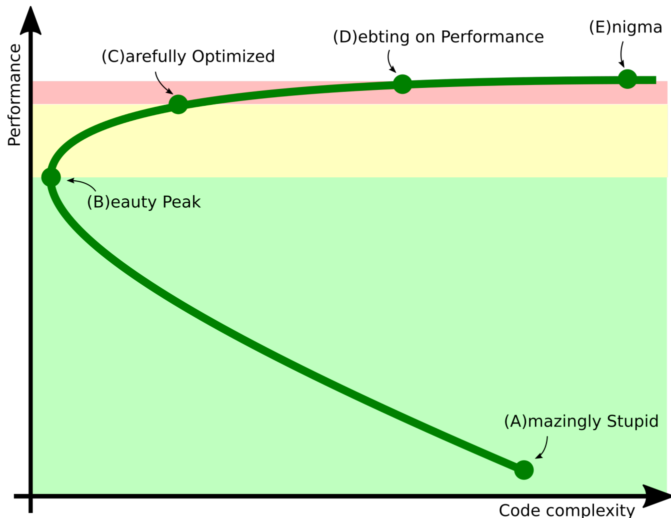


Вы в жёлтой зоне:

Берёте аккуратный профайлер, пишете таргет-бенчмарки, аккуратно закручиваете гайки

redhat

Крупно: Куда кривая вывезет

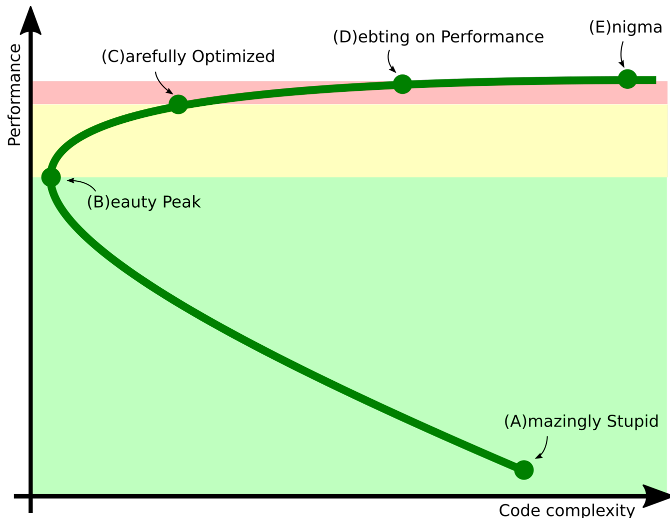


Вы в красной зоне:

Вас съел огр.

redhat.

Крупно: Куда кривая вывезет



Вы в красной зоне:

Идёте на JPoint/JokerConf/JBreak, пытаете разработчиков продуктов, как писать код, повторяющий кривизну нижних слоёв

redhat.

Зелёная зона

Зелёная зона: Мотивационная карточка

Зелёная зона:
борьба с ~~говнокодом~~ заусенцами
грубой силой

— Резать к чертовой матери, не дожидаясь перитонитов!
— Ты права, моя дорогая, с этим отростком пора кончать!

redhat.

Зелёная зона: Профилирование и диагностика

Ментальная ловушка:
«Профилировать нужно или нормально, или вообще никак»

На самом деле:

В зелёной зоне точность диагностики влияет очень мало! Вам нужно определить, какую часть говнокода выгодно перепилить в первую очередь.

redhat.

Зелёная зона: Диагностика

Linux Performance Observability Tools

strace
ss
Operating System
lsof
ltrace
netstat
sysdig
Hardware
Various:
sar /proc
dstat dmesg
pcstat
pidstat
perf
turbostat
rdmsr
mpstat
Applications
System Libraries
System Call Interface
CPU
Interconnect
perf
ftrace
stap
lttng
bcc
(BPF)
Linux Kernel
VFS
Sockets
Scheduler
File Systems
TCP/UDP
CPU
1
top ps
pidstat
Volume Manager
IP
Virtual
Memory
Memory
Bus
tiptop
perf
Block Device Interface
Ethernet
vmstat
slabtop
free
Device Drivers
DRAM
iostat
iotop
blktrace
perf tiptop
I/O Bus
I/O Bridge
iptraf
tcpdump
numastat
Expander Interconnect
I/O Controller
Network Controller
nicstat
netstat
ip
Interface Transports
Disk
Disk
Swap
Port
Port
swapon
ethtool
snmpget
lldptool

http://www.brendangregg.com/linuxperf.html 2016

Зелёная зона: Диагностика

Не бойтесь смотреть на ваше приложение!

Даже крупноклеточное понимание, какого ресурса не хватает, лучше, чем никакого понимания. Практически сразу показывает, куда копать:

- **много sys%**: копаем в сторону трединга и т.п.
- **много irq%:** копаем в сторону оффлоада прерываний и т.п.
- **много idle%:** ищем, где простаиваем
- **много iowait%:** копаем в сторону оптимизации I/O
- **много usr%:** аттачим профайлер, и...

redhat.

Зелёная зона: Профилирование

Наша цель:
примерно представлять, где мы проводим время

Даже самый наивный профайлер вам покажет все *ужасные* ошибки:

- хоть `jstack`
- хоть `perf top`
- хоть `VisualVM`
- да даже руками расставить `Stopwatch`-и уже хорошо
- да хоть из палок и желудей его соберите!

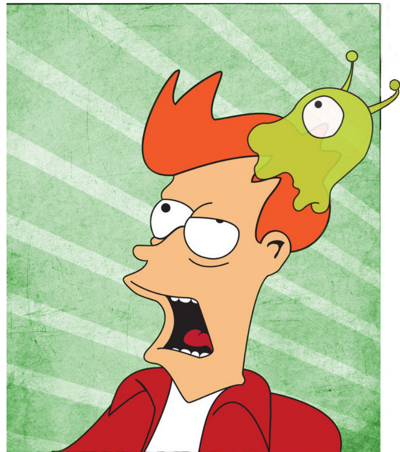
redhat.

Зелёная зона: Измерение производительности

Ментальная ловушка:
«Производительность нужно измерять или нормально, или вообще никак»

На самом деле:

В зелёной зоне улучшения в плюс-минус километр, можно и глазками увидеть. Да и какая разница, если всё равно переписываете говнокод?



redhat.

Зелёная зона: Мораль

Даже тривиальные нагрузочные тесты покажут вам *крупные огрехи*:

- Берёте production и публикуете ссылку в Твитторе
- Берёте staging и бешено тыкаете во всё подряд
- Берёте staging и хреначите в него Apache Bench’ем

Чем раньше в разработке вы получите перформансные данные, тем безболезненнее можно будет исправить огрехи (ещё до коммита)!

- Не надо сразу буйствовать и писать огромные сценарии
- Не надо тащить сложные нагрузочные генераторы
- Не надо пытаться обеспечить полную репрезентативность сценария

redhat.

Зелёная зона: Пример

Сюрпри-и-и-из:

JDK / JDK-8153229

JavacFiler.checkFileReopening drowns in exceptions after Modular Runtime Images change

Massive regression! Profiling jdk9b111 case yields this very hot branch:

```
| +- 10.080 (9%) org.openjdk.jmh.generators.annotations.APGeneratorDestinaton.newClass(java.lang.String)
| | +- 10.060 (9%) com.sun.tools.javac.processing.JavacFiler.createSourceFile(java.lang.CharSequence, javax.lang.model.element.Element[])
| | | +- 10.060 (9%) com.sun.tools.javac.processing.JavacFiler.createSourceOrClassFile(boolean, java.lang.String)
| | | +- 10.030 (9%) com.sun.tools.javac.processing.JavacFiler.checkFileReopening(javax.tools.FileObject, boolean)
| | | | +- 9.990 (8%) com.sun.tools.javac.file.JavacFileManager.isSameFile(javax.tools.FileObject, javax.tools.FileObject)
| | | | | +- 9.990 (8%) com.sun.tools.javac.file.PathFileObject.isSameFile(com.sun.tools.javac.file.PathFileObject)
| | | | | +- 9.990 (8%) java.nio.file.Files.isSameFile(java.nio.file.Path, java.nio.file.Path)
| | | | | +- 9.990 (8%) sun.nio.fs.UnixFileSystemProvider.isSameFile(java.nio.file.Path, java.nio.file.Path)
| | | | | +- 7.080 (6%) sun.nio.fs.UnixFileAttributes.get(sun.nio.fs.UnixPath, boolean)
| | | | | | +- 7.080 (6%) sun.nio.fs.UnixNativeDispatcher.stat(sun.nio.fs.UnixPath, sun.nio.fs.UnixFileAttributes)
| | | | | | +- 6.700 (6%) sun.nio.fs.UnixNativeDispatcher.stat0(long, sun.nio.fs.UnixFileAttributes)
| | | | | | | +- 2.750 (2%) sun.nio.fs.UnixException.<init>(int)
| | | | | | | +- 2.750 (2%) java.lang.Exception.<init>()
| | | | | | | +- 2.750 (2%) java.lang.Throwable.<init>()
| | | | | | | +- 2.740 (2%) java.lang.Throwable.fillInStackTrace()
| | | | | | | +- 2.740 (2%) java.lang.Throwable.fillInStackTrace(int)
```


redhat.

Зелёная зона: Пример

Сюрпри-и-и-из:

JDK / JDK-8153229

JavacFiler.checkFileReopening drowns in exceptions after Modular Runtime Images change

Massive regression! Profiling jdk9b111 case yields this very hot branch:

```
| +- 10.080 (9%) org.openjdk.jmh.generators.annotations.APGeneratorDestinaton.newClass(java.lang.String)
| | +- 10.060 (9%) com.sun.tools.javac.processing.JavacFiler.createSourceFile(java.lang.CharSequence, javax.lang.model.element.Element[])
```

This issue seems to blow up jcstress compilation time from 2 minutes to 8 minutes. This is a massive regression, please fix this in 9.

```
| | | | | | | +- 2.750 (2%) sun.nio.fs.UnixException.<init>(int)
| | | | | | | +- 2.750 (2%) java.lang.Exception.<init>()
| | | | | | | +- 2.750 (2%) java.lang.Throwable.<init>()
| | | | | | | +- 2.740 (2%) java.lang.Throwable.fillInStackTrace()
| | | | | | | +- 2.740 (2%) java.lang.Throwable.fillInStackTrace(int)
```


redhat.

Зелёная зона: Оптимизации

Ментальная ловушка:
«Преждевременная оптимизация – корень всего зла»

На самом деле:

Ну и какое зло в переписывании говно-кода? Использовать удобные модели данных и алгоритмы почти никогда не преждевременно.

redhat.

Зелёная зона: Заходы

Улучшение производительности в основном от переписывания плохого кода на хороший. Но «хорошесть» может быть и вкусовщиной, а может быть выстраданными приёмами:

- эффективнее структуры данных:
  `LinkedList` → `ArrayList`
- эффективнее алгоритмы:
  `ArrayList` → `HashMap`
  `keySet + get` → `entrySet`
  `bubbleSort` → `Collections.sort`
- меньше работы: что-то посчитать один раз и реиспользовать, етц

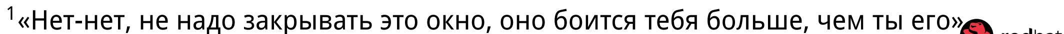
redhat.

Зелёная зона: Подитог

Профилирование – **необходимая** часть ежедневной разработки

Наблюдения:

- >95% проблем находится на первых же заходах
- >90% проблем тривиально разрешимы
- Чёткие инструкции по запуску профилировки **сильно** помогают: отлично, если есть однострочник, или однокнопочник, или APM
- Возьмёте девелопера за руку, и с ним один раз попрофилируете – это **уверенно** купирует боязнь базовой перформансной работы[1]



[1] «Нет-нет, не надо закрывать это окно, оно боится тебя больше, чем ты его»

Жёлтая зона

Жёлтая зона: Мотивационная карточка

Жёлтая зона:
~~нефть в обмен на продовольствие~~
усложение кода в обмен на производительность

Так вот, в момент, когда в голове у клиента происходит эта реакция, из кустов появляемся мы. Татарскому было очень приятно услышать это «мы». (солнце наше Пелевин, «Generation П[2]»)

[2] Перформанс

redhat.

Жёлтая зона: Профилирование и диагностика

Ментальная ловушка:
«Сейчас мы возмём профайлер, посмотрим что где, и как начнём оптимизировать»

На самом деле:
Возросшая цена ошибки предполагает, что мы будем вносить правильные изменения. Правильные изменения требуют продвинутой диагностики, и профилировка – только одна её часть!

redhat.

Жёлтая зона: Что собрались оптимизировать?

Hot Spots - Method	Self Time...	Self Time	Self Time (CPU)
java.lang.Object.**wait[native]** ()		390,047 ...(26.7%)	0.000 ms
sun.misc.Unsafe.**park[native]** ()		273,216 ...(18.7%)	0.000 ms
java.net.SocketInputStream.**socketRead0[native]** ()		244,844 ...(16.8%)	244,844 ms
java.net.PlainSocketImpl.**socketAccept[native]** ()		97,558 ms (6.7%)	0.000 ms
sun.management.ThreadImpl.**dumpThreads0[native]** ()		97,558 ms (6.7%)	97,558 ms
java.lang.Object.**hashCode[native]** ()		27,312 ms (1.9%)	27,312 ms
java.lang.System.**identityHashCode[native]** ()		25,978 ms (1.8%)	25,978 ms
com.sun.tools.javac.code.Type.**hasTag** ()		21,945 ms (1.5%)	21,945 ms
com.sun.tools.javac.comp.Attr$ResultInfo.**check** ()		17,751 ms (1.2%)	17,751 ms
com.sun.tools.javac.code.Types$DescriptorCache.**get** ()		13,326 ms (0.9%)	13,326 ms
com.sun.tools.javac.tree.JCTree$JCIdent.**accept** ()		10,730 ms (0.7%)	10,730 ms
com.sun.tools.javac.comp.Resolve$4.**argumentsAcceptable** ()		9,868 ms (0.7%)	9,868 ms
com.sun.tools.javac.code.Scope.**getIndex** ()		7,361 ms (0.5%)	7,361 ms
com.sun.tools.javac.code.Type$ClassType.**accept** ()		6,218 ms (0.4%)	6,218 ms
com.sun.tools.javac.code.Types$18.**visitClassType** ()		5,807 ms (0.4%)	5,807 ms
java.lang.Object.**clone[native]** ()		5,435 ms (0.4%)	5,435 ms

redhat.

Жёлтая зона: Modus Operandi

Теперь, оптимизируя, вы вынуждены объяснять, зачем вы это делаете – хотя бы себе, а может и PM-у

При этом желательно:

1. Иметь на руках численные оценки приростов
2. Иметь оценки приростов **до** того, как потратить всё ресурсы
3. Иметь понимание, что это самый дешёвый способ

redhat.

Оценки: Pop Quiz

Представим себе приложение с двумя отдельными частями:

- Часть A занимает 70% времени, разгоняема в 2 раза
- Часть B занимает 30% времени, разгоняема в 6 раз
- Какую часть будем разгонять?

redhat

Оценки: Pop Quiz

Представим себе приложение с двумя отдельными частями:

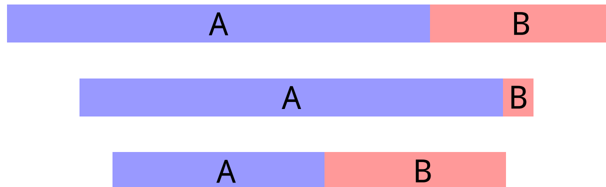
- Часть A занимает 70% времени, разгоняема в 2 раза
- Часть B занимает 30% времени, разгоняема в 6 раз
- Какую часть будем разгонять?

redhat

Оценки: Pop Quiz

Представим себе приложение с двумя отдельными частями:

- Часть A занимает 70% времени, разгоняема в 2 раза
- Часть B занимает 30% времени, разгоняема в 6 раз
- Какую часть будем разгонять?



redhat.

Оценки: Закон Амдала

$$Part_A = \frac{A}{A+B}$$
$$Part_B = \frac{B}{A+B}$$

Закон Амдала:

$$S = \frac{A+B}{\frac{A}{S_A}+B} = \frac{1}{\frac{Part_A}{S_A}+Part_B}$$

Следствия:

$$\lim_{Part_A \to 0} S = 1$$

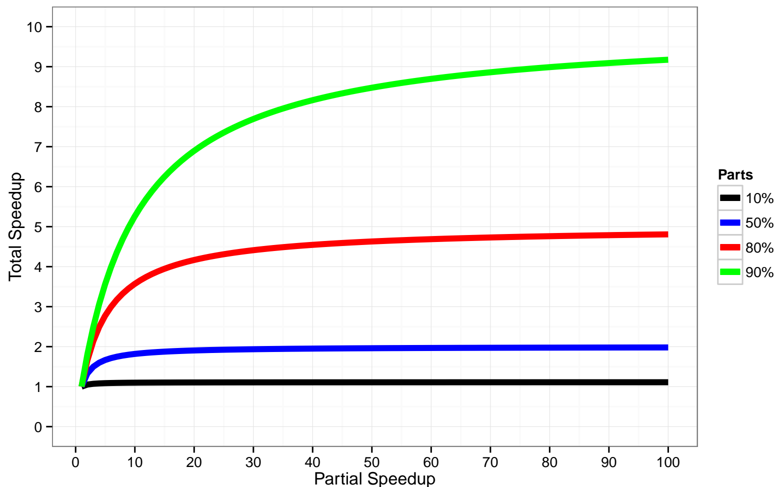
$$\lim_{Part_A \to 1} S = S_A$$

$$\lim_{S_A \to 0} S = 0$$

$$\lim_{S_A \to \infty} S = \frac{1}{Part_B}$$

redhat.

Закон Амдала: Поведение



redhat.

Закон Амдала: Обобщение

Немножко поиграем членами:

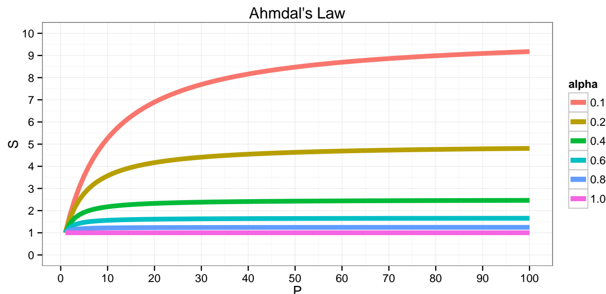
$$S = \frac{1}{\frac{P_A}{S_A} + P_B} = \frac{1}{\frac{1-P_B}{S_A} + P_B} = \frac{S_A}{1 - P_B + P_B S_A} = \frac{S_A}{1 + P_B(S_A - 1)}$$

или, после подстановки
$p = S_A$ (во сколько раз ускорили часть),
$\alpha = P_B$ (сколько весит всё остальное):

$$S = \frac{p}{\underbrace{1}_{concurrency} + \underbrace{\alpha(p-1)}_{contention}}$$

redhat.

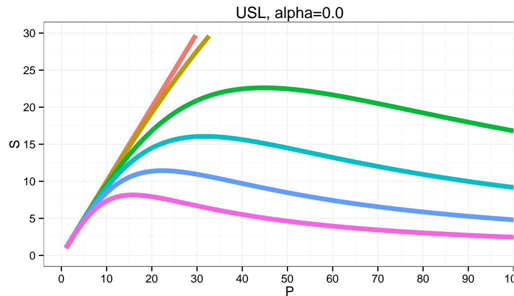
Закон Амдала: Поведение

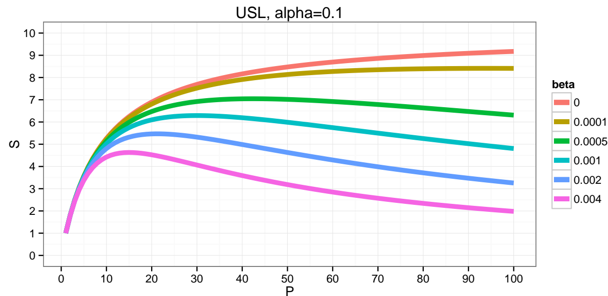


$$S = \frac{p}{\underbrace{1}_{concurrency} + \underbrace{\alpha(p-1)}_{contention}}$$

redhat.

USL: Universal Scalability Law





$$S = \frac{p}{\underbrace{1}_{concurrency} + \underbrace{\alpha(p-1)}_{contention} + \underbrace{\beta p(p-1)}_{coherence}}$$

redhat.

USL: Universal Scalability Law

$$S = \frac{p}{\underbrace{1}_{concurrency} + \underbrace{\alpha(p-1)}_{contention} + \underbrace{\beta p(p-1)}_{coherence}}$$

Наблюдения:

- USL хорошо натягивается на эмпирические данные
- При $\beta > 0$, с разгоном конкретной части не то, что может не становиться лучше, может становиться **хуже**
- Систем с $\alpha = 0$ и $\beta = 0$ практически не существует

redhat.

Жёлтая зона: Измерение производительности

Ментальная ловушка:
«Да мы накодим и посмотрим, что нам скажут бенчмарки»

На самом деле:
Перформанс-тестирование в норме дико дорогое, и всё не протестируешь.

redhat.

Жёлтая зона: Тестирование

Перформансное тестирование – дорогое удовольствие!

- Один тест проходит минуту $\Rightarrow$ сотни машинных часов на коммит?

redhat.

Жёлтая зона: Тестирование

Перформансное тестирование – дорогое удовольствие!

- Один тест проходит минуту $\Rightarrow$ сотни машинных часов на коммит?
- Требуют изоляции $\Rightarrow$ нельзя шарить HW?

redhat.

Жёлтая зона: Тестирование

Перформансное тестирование – дорогое удовольствие!

- Один тест проходит минуту $\Rightarrow$ сотни машинных часов на коммит?
- Требуют изоляции $\Rightarrow$ нельзя шарить HW?
- Не бинарная метрика $\Rightarrow$ человеко-часы на разбор данных?

redhat.

Жёлтая зона: Тестирование

Перформансное тестирование – дорогое удовольствие!

- Один тест проходит минуту ⇒ сотни машинных часов на коммит?
- Требуют изоляции ⇒ нельзя шарить HW?
- Не бинарная метрика ⇒ человеко-часы на разбор данных?
- Ошибки тестирования находятся только после разбора данных...

redhat.

Жёлтая зона: Тестирование

Перформансное тестирование – дорогое удовольствие!

- Один тест проходит минуту $\Rightarrow$ сотни машинных часов на коммит?
- Требуют изоляции $\Rightarrow$ нельзя шарить HW?
- Не бинарная метрика $\Rightarrow$ человеко-часы на разбор данных?
- Ошибки тестирования находятся только после разбора данных...
- Бенчмарки дают **данные**, а хочется-то **результатов**

redhat.

Жёлтая зона: Тестирование

Перформансное тестирование – дорогое удовольствие!

- Один тест проходит минуту ⇒ сотни машинных часов на коммит?
- Требуют изоляции ⇒ нельзя шарить HW?
- Не бинарная метрика ⇒ человеко-часы на разбор данных?
- Ошибки тестирования находятся только после разбора данных...
- Бенчмарки дают **данные**, а хочется-то **результатов**

Выводы:

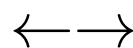
1. В активном проекте практически невозможно тестировать всё!
2. Нужно всё-таки уметь разбираться, куда копать...

redhat.

Бенчмарки: Много их, мой друг Горацио

Макробенчмарки

1. берём большой сайт, приложение, библиотеку целиком
2. пишем большой сценарий
3. измеряем от начала и до конца



Микробенчмарки

1. берём маленькую часть сайта, приложения, библиотеки
2. делаем мелкий изолированный тест
3. измеряем конкретную часть

redhat

Бенчмарки: Макробенчмарки

Голоса в голове у разработчика ему шепчут:

1. Макробенчмарк отражает реальный мир
    - ...и любой макробенчмарк – хороший
    - ...запустил макробенчмарк – и это «real world»

redhat.

Бенчмарки: Макробенчмарки

Голоса в голове у разработчика ему шепчут:

1. Макробенчмарк отражает реальный мир
   - ...и любой макробенчмарк – хороший
   - ...запустил макробенчмарк – и это «real world»

2. Для любой крутой фичи, макробенчмарк даст крутое улучшение
   - ...если макробенчмарк не показывает улучшения, то фича – так себе
   - ...если макробенчмарк показывает улучшение, то фича – золото

redhat.

Бенчмарки: Макробенчмарки

Голоса в голове у разработчика ему шепчут:

1. Макробенчмарк отражает реальный мир
   - ...и любой макробенчмарк – хороший
   - ...запустил макробенчмарк – и это «real world»
2. Для любой крутой фичи, макробенчмарк даст крутое улучшение
   - ...если макробенчмарк не показывает улучшения, то фича – так себе
   - ...если макробенчмарк показывает улучшение, то фича – золото
3. Для любого крутого бага, макробенчмарк даст крутую регрессию
   - ...если макробенчмарк не показал регрессии, то бага и нет
   - ...если макробенчмарк показывает регрессию, то баг жуткий

redhat

Бенчмарки: Микробенчмарки

Голоса в голове у разработчика ему шепчут:

1. Микробенчмарки – зло
   - Очень удобно: раз зло, значит можно не обращать внимания

redhat.

Бенчмарки: Микробенчмарки

Голоса в голове у разработчика ему шепчут:

1. Микробенчмарки – зло
   - Очень удобно: раз зло, значит можно не обращать внимания
2. Микробенчмарк можно написать какой угодно
   - Регрессия на микробенчмарке ничего не значит
   - Улучшение на микробенчмарке ничего не значит

redhat.

Бенчмарки: Микробенчмарки

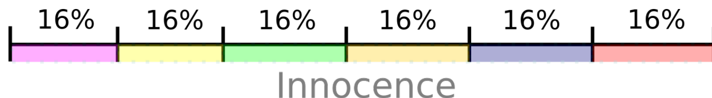
Голоса в голове у разработчика ему шепчут:

1. Микробенчмарки – зло
   - Очень удобно: раз зло, значит можно не обращать внимания
2. Микробенчмарк можно написать какой угодно
   - Регрессия на микробенчмарке ничего не значит
   - Улучшение на микробенчмарке ничего не значит
3. Микробенчмарки пишут враги, чтобы опорочить наш продукт
   - Говори, что микробенчмарк неправильный – и кодь дальше

redhat.

Бенчмарки: Жизненный цикл бенчмарков

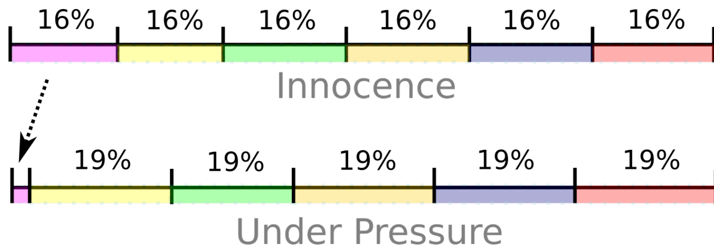
Все, все, все бенчмарки проходят эти стадии жизненного цикла:



redhat.

Бенчмарки: Жизненный цикл бенчмарков

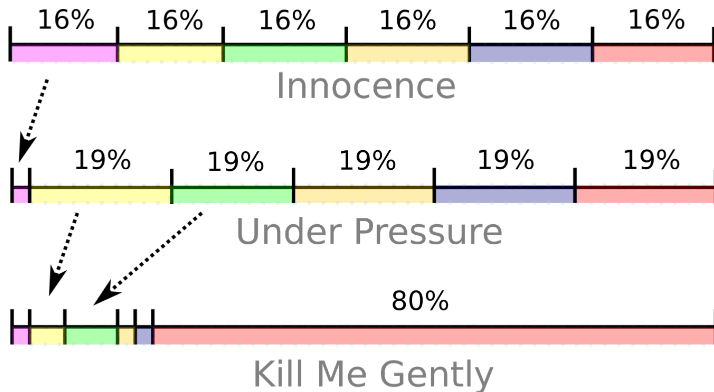
Все, все, все бенчмарки проходят эти стадии жизненного цикла:



redhat.

Бенчмарки: Жизненный цикл бенчмарков

Все, все, все бенчмарки проходят эти стадии жизненного цикла:

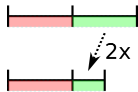


Даже если ты начал как макробенчмарк, ты кончишь микробенчмарком

redhat.

Бенчмарки: ...но есть и свои плюсы

Улучшения:



S = $Speedup$

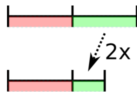
p = $Speedup_{part}$

$$S = \frac{p}{\underbrace{1}_{concurrency} + \underbrace{\alpha(p-1)}_{contention}}$$

redhat.

Бенчмарки: ...но есть и свои плюсы

Улучшения:

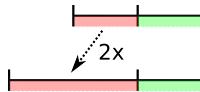


S = $Speedup$

p = $Speedup_{part}$

$$S = \frac{p}{\underbrace{1}_{concurrency} + \underbrace{\alpha(p-1)}_{contention}}$$

Регрессии:



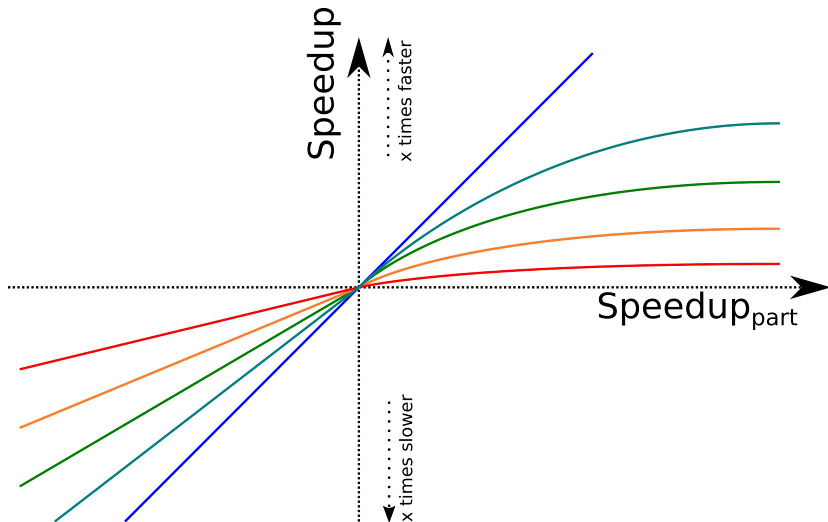
$$R = \frac{1}{S}$$

$$r = \frac{1}{p}$$

$$R = \underbrace{\alpha}_{save} + \underbrace{r(1-\alpha)}_{regresssion}$$

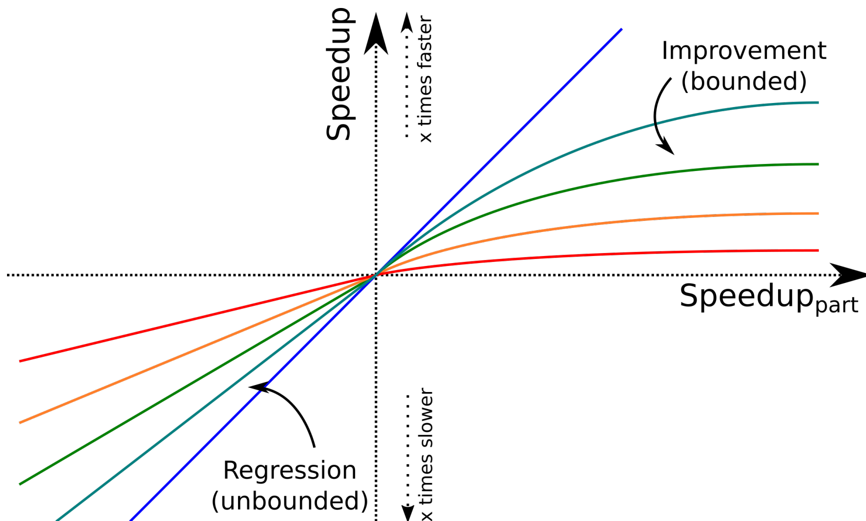
redhat.

Бенчмарки: Подставы



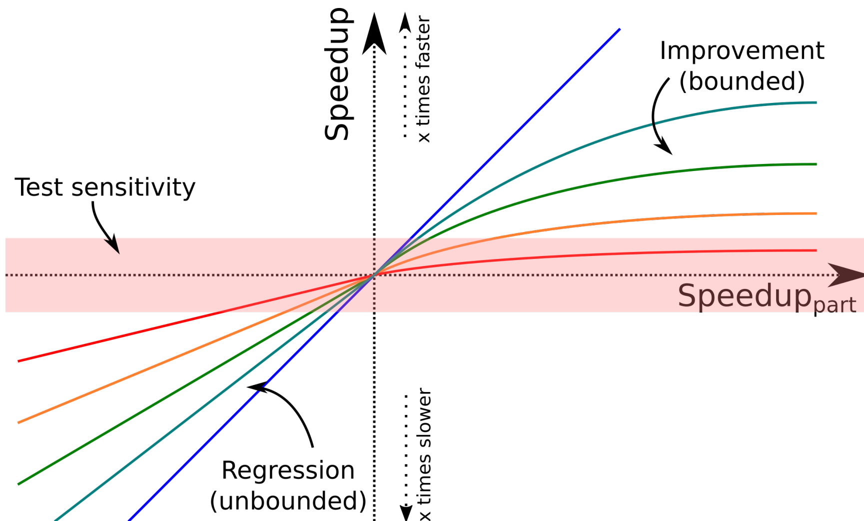
redhat.

Бенчмарки: Подставы



redhat.

Бенчмарки: Подставы



redhat.

Бенчмарки: Наблюдения

Макробенчмарки:

1. Их мало, они написаны непонятно кем и непонятно как
2. Сначала показывают интересное, но потом вырождаются
3. В конце жизни применимы для **регрессионного** тестирования

Микробенчмарки:

1. Их много, написаны непонятно как и кем, но их можно *исправить*
2. Почти всегда показывают интересное (но часто не то)
3. Отлично реагируют и на **регрессии**, и на **улучшения**

Оба этих класса не взаимозаменяемы!

redhat.

Бенчмарки: Микробенчмарки

Как вы не крутитесь,
а учиться микробенчмаркать вам придётся!

- «Просто возьми JMH» не работает
  - Нужно грамотно планировать эксперименты
  - Грамотно их проводить
  - Грамотно анализировать данные
  - Делать правильные выводы
- Главная задача – **построить правдоподобную модель**
  - Что и как влияет на производительность
  - Как конкретный сценарий прилежит к остальным сценариям

redhat

Жёлтая зона: Оптимизации

Ментальная ловушка:
«Да мы вот попробовали, оно улучшило метрики. Наверное, потому что (далее следует наукообразное рассуждение)»

На самом деле:
Часто бывает, что тупо повезло: с компонентом, с рабочей нагрузкой, с патчем, с фазой луны. Чуть везение пропадёт, всё вернётся на круги своя.

redhat

Жёлтая зона: И вот вижу я улучшение

Варианты:

1. Косяк в моём коде
   Реакция: исправление + посыпание головы пеплом

redhat.

Жёлтая зона: И вот вижу я улучшение

Варианты:

1. Косяк в моём коде
   Реакция: исправление + посыпание головы пеплом

2. Косяк в моём использовании библиотеки/рантайма
   Реакция: исправление + PR в документацию

redhat.

Жёлтая зона: И вот вижу я улучшение

Варианты:

1. Косяк в моём коде
   Реакция: исправление + посыпание головы пеплом

2. Косяк в моём использовании библиотеки/рантайма
   Реакция: исправление + PR в документацию

3. Исправимый косяк в библиотеке/рантайме
   Реакция: временная заплатка, с зарубкой на память

redhat.

Жёлтая зона: И вот вижу я улучшение

Варианты:

1. Косяк в моём коде
   Реакция: исправление + посыпание головы пеплом

2. Косяк в моём использовании библиотеки/рантайма
   Реакция: исправление + PR в документацию

3. Исправимый косяк в библиотеке/рантайме
   Реакция: временная заплатка, с зарубкой на память

4. Неисправимый косяк в библиотеке/рантайме
   Реакция: постоянная заплатка, с внесением в анналы

redhat.

Жёлтая зона: Типичные штуки – опции JVM

Идея: зная что-то специальное о нашем приложении, подскажем JVM, в каком режиме работать

Радости: Синергия, механическая симпатия, всё такое

Проблемы: Ну как бэ...

-Xmx1G -Xms1G -Xmn128m -XX:+DisableExplicitGC -XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:+UseNUMA -XX:+CMSParallelRemarkEnabled -XX:MaxTenuringThreshold=15 -XX:MaxGCPauseMillis=30 -XX:GCPauseIntervalMillis=150 -XX:+UseAdaptiveGCBoundary -XX:-UseGCOverheadLimit -XX:+UseBiasedLocking -XX:SurvivorRatio=8 -XX:TargetSurvivorRatio=90 -XX:MaxTenuringThreshold=15 -Dfml.ignorePatchDiscrepancies=true -Dfml.ignoreInvalidMinecraftCertificates=true -XX:+UseFastAccessorMethods -XX:+UseCompressedOops -XX:+OptimizeStringConcat -XX:+AggressiveOpts -XX:ReservedCodeCacheSize=2048m -XX:+UseCodeCacheFlushing -XX:SoftRefLRUPolicyMSPerMB=2000 -XX:ParallelGCThreads=10

redhat.

Жёлтая зона: Типичные штуки – параллелизм

Идея: неважно, что там где написано, берём `parallelStream()`, `Executor.submit`, `new Thread`, и параллелим

Радости:

1. Много ума не надо – раз, и готово!

Проблемы:

1. Синхронизация же – уверены, что всё работает?
2. Оверхеды же – уверены, что работы достаточно?
3. Куча внешнего параллелизма – внутренний параллелизм не нужен

redhat

Жёлтая зона: Типичные штуки – структуры данных

Идея: идут в печь эти `Collection<Integer>`,
будем делать `int[]`

Радости:

1. Плотненько так, упаковано, ням-ням

Проблемы:

1. Конверсии обратно во врапперы – уверены, что не сожрёт?
2. Конверсии туда-обратно всех коллекций – уверены?
3. Вставки-удаления-тормошения?
4. Оптимизации самой JDK? К Valhalla готовы?

redhat.

Жёлтая зона: Подитог

Поддержание правдоподобной перформансной модели проекта – **необходимое** условие развития проекта

Наблюдения:

- >50% потенциальных изменений делаются не там, где стоит
- >80% изменений делаются в нужном месте после исследования
- Умение *исследовать* и обновлять свои знания о проекте помогают принимать правильные решения – *поощряйте* это у девелоперов. Если проект большой, то вам нужны *выделенные роли* на такую работу.

redhat

Красная зона

Красная зона: Мотивационная карточка

Красная зона: эксплуатация кривизны нижних слоёв, грязные хаки и залезание в кишки

Здесь Паша Эмильевич, обладавший сверхъестественным чутьем, понял, что сейчас его будут бить, может быть, даже ногами.

redhat.

Красная зона: Внимание

В абсолютном большинстве проектов сюда ходить не надо!
Вменяемый техлид, проджект-менеджер или заботливая мама
должны сказать:

redhat.

Красная зона: Внимание

В абсолютном большинстве проектов сюда ходить не надо!
Вменяемый техлид, проджект-менеджер или заботливая мама
должны сказать:

ОСТАНОВИТЕСЬ!
ХВАТИТ!
ЛОПНЕТЕ!

redhat.

Красная зона: Основная идея

Улучшение производительности от эксплуатации **особенностей реализации** нижних слоёв

- дёрганье скрытых приватных методов
- дёрганье в «нужном» порядке публичных методов
- доступ к целым кускам приватного API
- хаки для обхода особенностей библиотек и рантаймов
- микроархитектурные оптимизации

redhat.

Красная зона: Профилирование и диагностика

Ментальная ловушка:
«Если долго смотреть в профайл, можно увидеть там решение»

На самом деле:
Умение хакать вырастает из понимания взаимодействия всех движущихся деталек.

redhat.

Красная зона: Нарабатывание корпуса

Перформансники –
это те люди, которые умеют копаться во всех слоях сразу

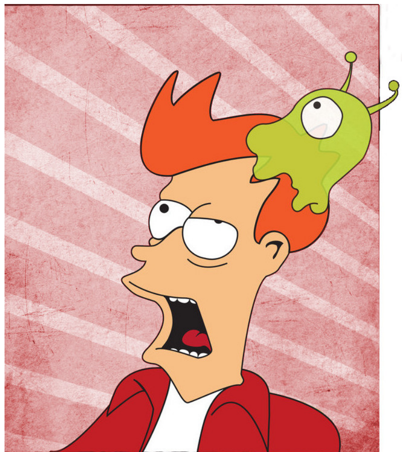
- Имеют наработанный корпус хаков, знают границы их применимости
- **Не впадают в ступор**, когда видят незнакомую хрень, а начинают её изучать
- «Изучать» = читать документацию, искать упоминаний в статьях, смотреть на другие части кода и историю проекта, делать эксперименты, спрашивать коллег-специалистов, а не *спрашивать на StackOverflow «ой, а чо это такое?»*

redhat.

Красная зона: Подходы к исправлению

Ментальная ловушка:
«Если долго ездить по конференциям, то когда-нибудь там расскажут трюк, дающий 5x перформанса, и мы его тут же применим»

На самом деле:
В конкретном случае спасёт один низкоуровневый трюк из тысячи! Его проще найти самостоятельно, чем ждать у моря погоды.



redhat.

Красная зона: Откроем, значит, StackOverflow...

В программистской тусовке б**о**льшая часть обсуждений об этом!

- `(i++)` или `(++i)`?
- `for (int c = 0; c < L; c++)` или `while (c --> 0)`?
- `Math.pow(x, 2)` или `x*x`?
- `(x*2)` или `(x << 1)`?
- `(a*b != 0)` или `(a != 0 && b != 0)`?
- `(a & b)` или `(a && b)`?
- `String.isEmpty()` или `String.trim().length() == 0`?
- `System.arraycopy` или таки `моя-любимая-идиома`

redhat

Красная зона: Костыли

Хаки должны быть *временными* заплатками, а не основой вашего проекта

- Указывают на проблему в слое ниже: нужно или исправить, или изучить, или зарепортить!
- Учите хакам студентов – сами вбиванете гвозди в гроб IT
- Условный Куксенко рассказывает про низкоуровневую вакханалию для нарабатывания корпуса *возможных* хаков, а не чтобы вы их везде использовали

redhat.

Красная зона: Пример: ArrayList.Itr

```
public class ArrayList<E> implements Iterable<E> {
  public Iterator<E> iterator() {
    return new Itr();
  }

  private class Itr implements Iterator<E> { }
```


redhat.

Красная зона: Пример: ArrayList.Itr

```
public class ArrayList<E> implements Iterable<E> {
  public Iterator<E> iterator() {
    return new Itr();
  }

  private class Itr implements Iterator<E> { }

  public [bridge] Itr(java.util.ArrayList, java.util.ArrayList$1);
```


redhat.

Красная зона: Пример: ArrayList.Itr

```
public class ArrayList<E> implements Iterable<E> {
  public Iterator<E> iterator() {
    return new Itr();
  }

  private class Itr implements Iterator<E> { }

  public [bridge] Itr(java.util.ArrayList, java.util.ArrayList$1);

/*
@ 3   j.u.ArrayList::iterator (10 bytes)  inlined (hot)
- @ 6   j.u.ArrayList$Itr::<init> (6 bytes) unloaded signature classes
*/
}
```


redhat

Красная зона: Пример: ArrayList.Itr

```
public class ArrayList<E> implements Iterable<E> {
  public Iterator<E> iterator() {
    return new Itr();
  }

  // HACK: Create explicit constructor to avoid generating a bad one
  // Needed because javac generates synthetic bridge with "unloaded class"
  // arguments, see JDK-xxxxxxxx
  Itr() {};

  private class Itr implements Iterator<E> { }
}
```


redhat.

Красная зона: Подходы к исправлениям

Ментальная ловушка:
«Сейчас мы быстро подкрутим вот здесь, и будет зашибись»

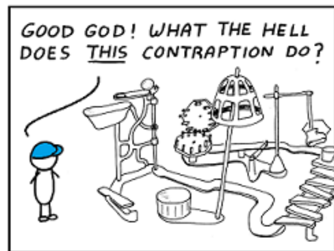
На самом деле:
В красной зоне **никогда** не будет зашибись.

redhat.

(три года назад)

redhat

Красная зона: Технический долг!



I hate reading other people's code.

Не обманывайте себя: работая в красной зоне, вы вносите технический долг.

Всегда, всегда, всегда документируйте:

1. Из-за чего хак выполнен
2. В каких условиях он применим
3. Как проверить, что хак больше не нужен
4. Какие upstream-баги вы ждёте, етц

redhat.

Красная зона: Подитог

Удержание численности и плотности хаков в проекте – **необходимое** условие выживания проекта

Наблюдения:

- Дай волю, и весь проект порастёт...
- Умение работать с *upstream*-ами и другими компонентами сильно облегчают долговременную судьбу: конверсия хаков в реальные патчи улучшает глобальное положение
- Умение разбираться во всех слоях увеличивает тренируется «на кошках»

redhat

Напутствие

Напутствие: На хардкорной конференции

Пользователям продуктов:

1. Обновить запас и применимости хаков из **красной зоны**
2. Найти/обсудить подходы работы в **жёлтой зоне**
3. Обсудить, что из **жёлтой зоны** пора завещать в **зелёную зону**

redhat.

Напутствие: На хардкорной конференции

Пользователям продуктов:

1. Обновить запас и применимости хаков из **красной зоны**
2. Найти/обсудить подходы работы в **жёлтой зоне**
3. Обсудить, что из **жёлтой зоны** пора завещать в **зелёную зону**

Разработчикам продуктов:

1. Обсудить подходы работы в **жёлтой зоне**
2. Понять, какие хаки из **красной зоны** перетащить в скрижали **жёлтой зоны**, или вообще в **зелёную зону**.

redhat.

Напутствие: На хардкорной конференции

Пользователям продуктов:

1. Обновить запас и применимости хаков из **красной зоны**
2. Найти/обсудить подходы работы в **жёлтой зоне**
3. Обсудить, что из **жёлтой зоны** пора завещать в **зелёную зону**

Разработчикам продуктов:

1. Обсудить подходы работы в **жёлтой зоне**
2. Понять, какие хаки из **красной зоны** перетащить в скрижали **жёлтой зоны**, или вообще в **зелёную зону**.

Помните: основная перф работа проходит в **зелёной зоне**

redhat.

Конец

redhat.

Конец?

redhat.