パフォーマンス徹底比較

Seasar2 vs Spring

2006/04/12

株式会社電通国際情報サービス
ひがやすを
株式会社アークシステム
本間 宏崇

目的

- DIコンテナの実装によるパフォーマンスの違いを明らかにする
- DIコンテナが行う処理の中で、どこに時間が掛かるのかを明らかにする

ベンチマーク測定環境

- **ハードウェア**
  - HP ProLiant DL360 G4p
  - CPU: Intel Xeon 3.80GHz (2 CPU)
  - Memory: 4GB
- **ソフトウェア**
  - OS: Red Hat Enterprise Linux AS 4 Update 3 (x86)
  - Java: 1.5.0_06 (Sun)

測定アプリケーション

- DIコンテナ
  - Seasar 2.4 beta1 (2006/03/27)
  - Spring 2.0 M3 (2006/03/08)
- **ベンチマークプログラム**
  - **自作**

測定方法

- VMオプション
  - -Xmx1024M
  - -Xms1024M
  - -XX:PermSize=384M
  - -XX:MaxPermSize=384M
  - fork=true
    - JVMのキャッシュをクリアするため
- 5回実行し、最大･最小を除いた3回の平均値を採る

DIコンテナがやっていること

- コンテナ生成
  - XML読み込み(DOMやSAX)
- 定義からコンポーネントを組み立てる
  - DI
    - リフレクション
      - リフレクション情報を取得しキャッシュする
      - リフレクションを使用しプロパティへアクセスする
  - AOP
    - バイトコード作成

- それぞれの処理について、パフォーマンスを見ていきましょう

- まずは、XML読み込みを行っている、コンテナ生成処理からです。

コンテナ生成

- コンテナ生成時の内部処理
  - Seasar
    - SAX
    - リフレクション情報をキャッシュ
  - Spring
    - DOM
    - ※リフレクション処理はここでは行わない

- コンテナへ入力する設定ファイル
  - Seasar

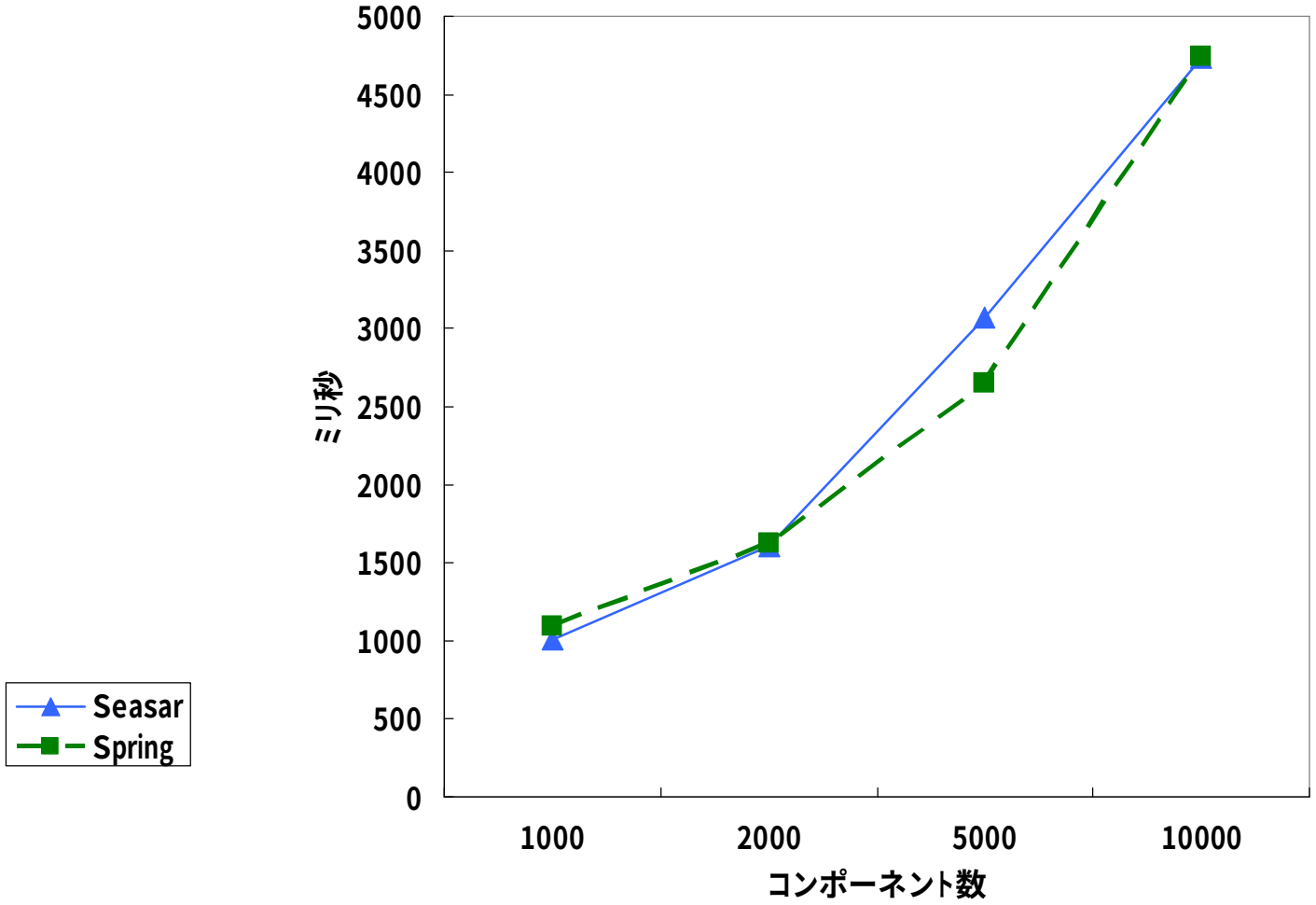
```
<components>
   <component name="nullBean00000" class="xxx.NullBean00000" />
   <component name="nullBean00001" class="xxx.NullBean00001" />
   …
```

  - Spring

```
<beans>
   <bean name="nullBean00000" class=“xxx.NullBean00000" />
   <bean name="nullBean00001" class=“xxx.NullBean00001" />
…
```

コンテナ生成

- コンテナ生成処理



コンテナ生成:結果

- Seasar　　Spring

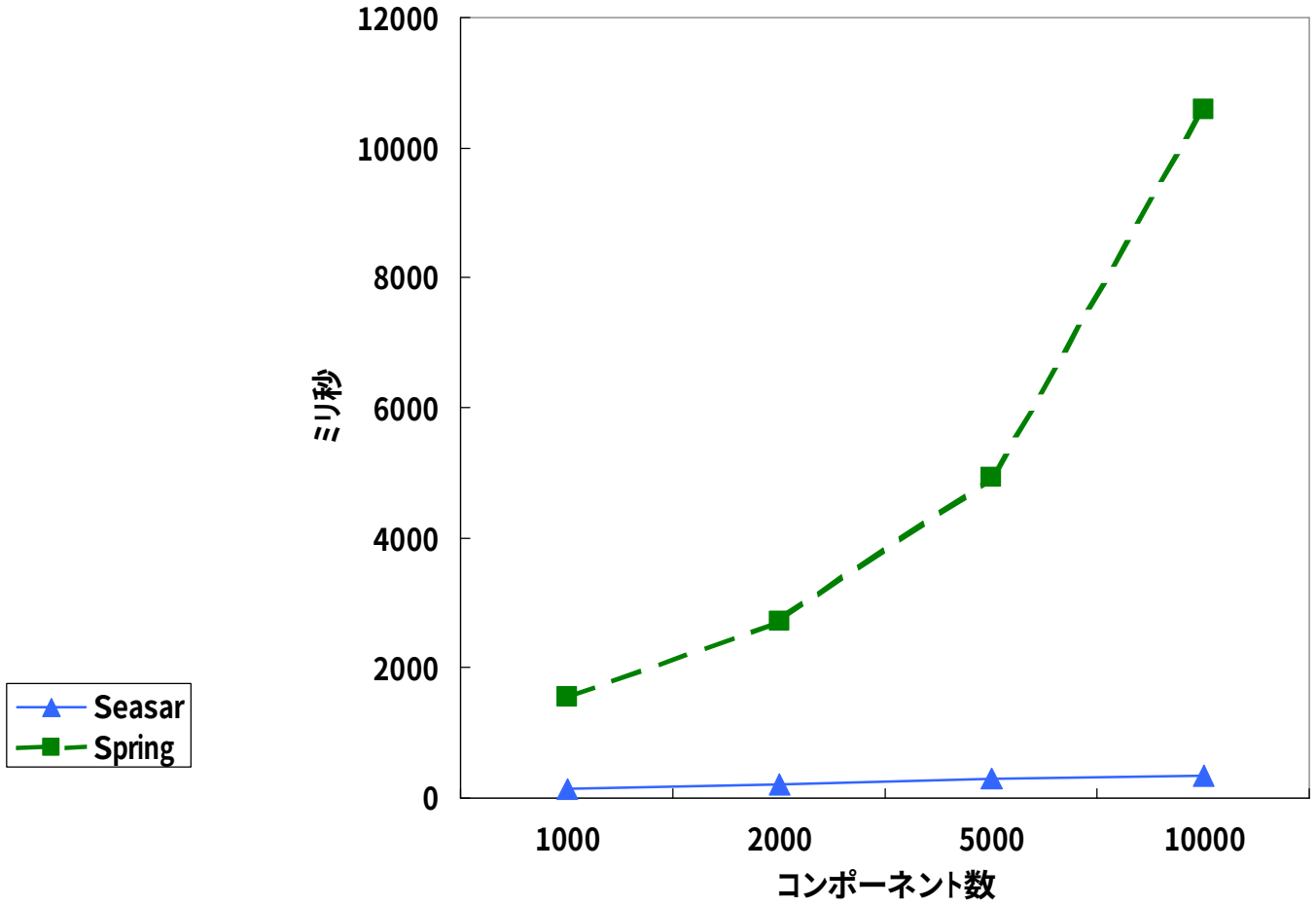
- **理由**
  - **リフレクション情報をキャッシュするぶん**Seasar**の方が多くの処理を行っていますが、**SAX**と**DOM**の性能差によって吸収されていると思われます。**

コンポーネント取得

- 次は、生成したコンテナからコンポーネントを取得する処理です

- コンテナに登録されている全てのコンポーネントを取得するのに掛かった時間を計測しました
  - DI・AOPは使用していません
  - 単純に、コンテナからコンポーネントを取得するのみです

コンポーネント取得



コンポーネント取得:結果

- Seasar ＞＞(10〜30倍)＞＞ Spring
  - 1000個で1400ms

- **コンポーネントを生成するという点ではどちらも一緒のはずですが、どうして差が出るのでしょうか?**

コンポーネント取得

- 理由
  - DIコンテナは、コンポーネントを生成するためにリフレクション情報を使用しています
  - Seasarはコンテナ生成時にリフレクション情報をキャッシュしています。コンポーネント生成時にはキャッシュした情報を使用しています
  - Springはコンポーネントを取得するときにリフレクション情報をキャッシュしています

コンポーネント取得

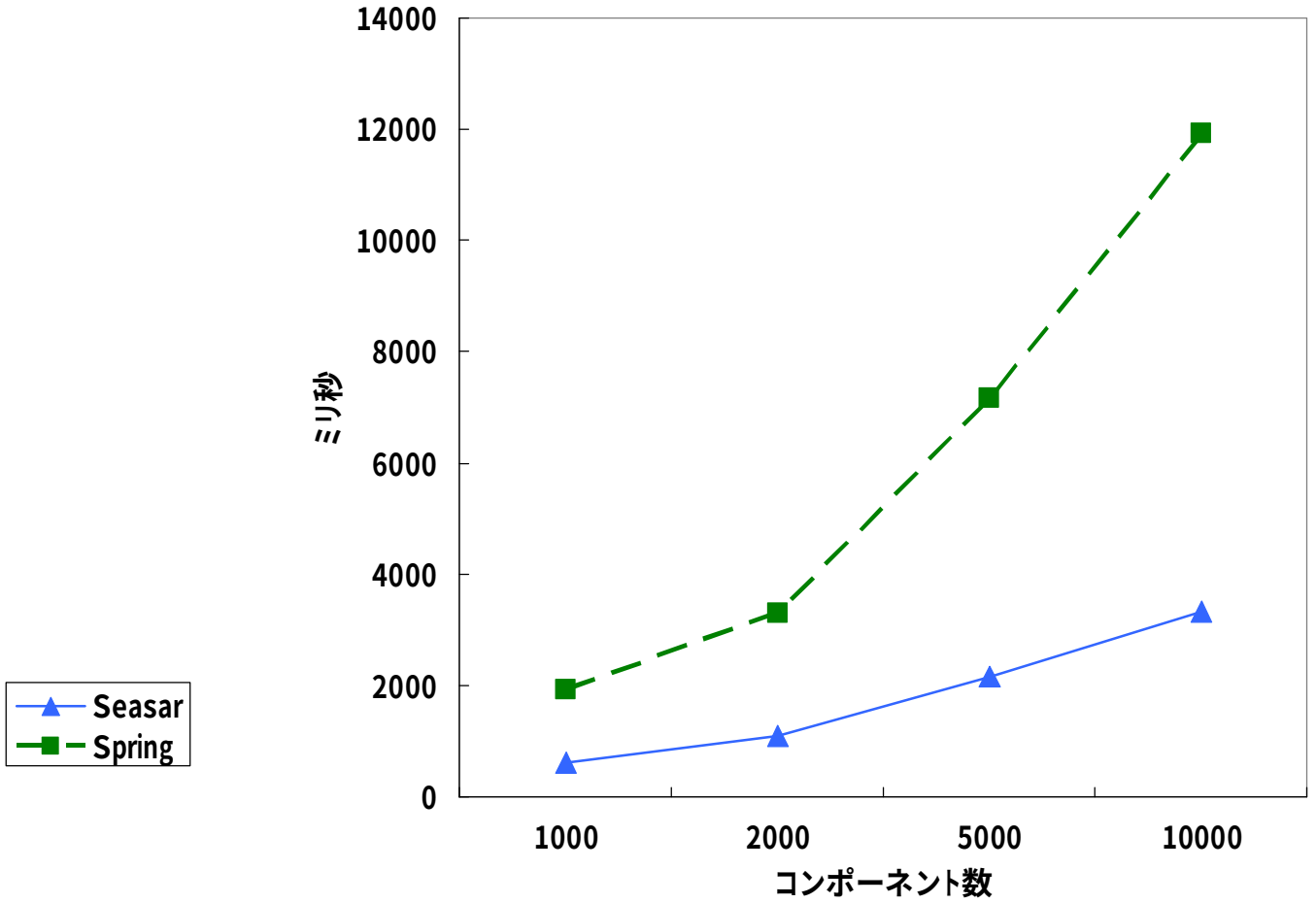
- 理由
  - Springはコンポーネント取得時にリフレクション処理を行っているため、遅くなります
  - Seasarはコンテナ生成時にリフレクション処理を行っていますが、SAXとDOMの性能差によってSpringとの差が無くなっています
  - そのため、コンポーネント取得時にSeasarの速さが際立っています

リフレクション処理

- **では、SeasarとSpringのリフレクション処理はどれくらい違うのでしょうか?**
  - **リフレクション処理を行うクラスを直接呼び出して測定しました。**
    - Seasar: BeanDescImpl
    - Spring: BeanWrapperImpl

リフレクション処理

- リフレクション情報をキャッシュ



リフレクション処理:結果

- Seasar >(3倍)> Spring
  - 1000回で1300ms

- **理由**
  - **Seasarはリフレクションキャッシュ処理を独自で実装しています。SpringのBeanWrapperImplはJDKのIntrospectorを使用しています。この違いが速度差となっていると思われます**
  - **キャッシュ実装の違い**
    - Seasar: HashMap
    - Spring: WeakHashMap

Seasarのコンテナinit処理

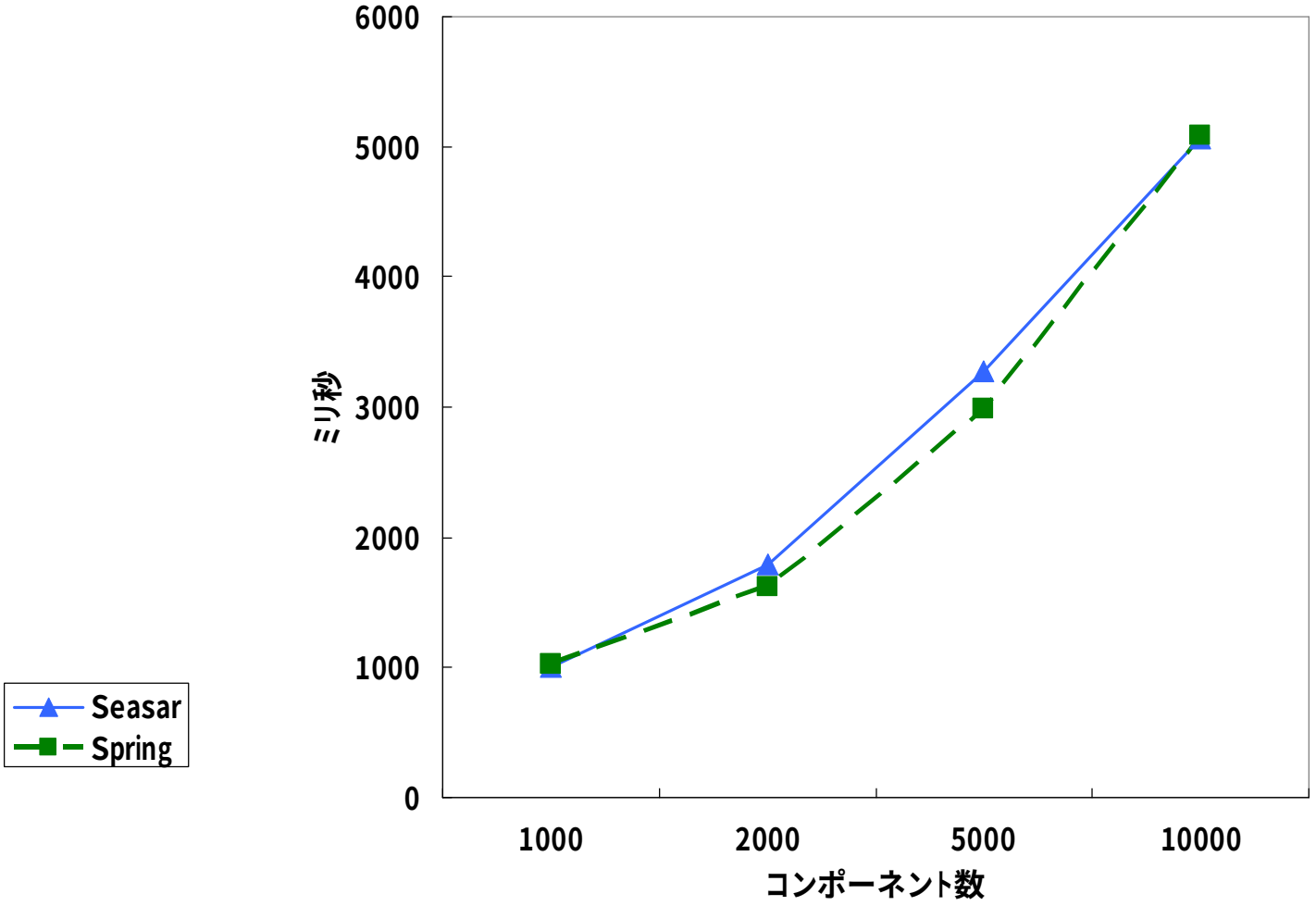
- Seasarではコンテナ生成直後にinit処理を行うことができます
  - 先ほどまではコンテナのinit処理を行っていませんでした
  - init処理を行わない場合は、1度目にコンポーネントを取得したタイミングで、そのコンポーネントがインスタンス化されます
- init処理ではsingletonのコンポーネントを作成することができます
  - singletonとは、コンポーネント生成は最初1度だけで、その後は最初に生成したコンポーネントを返すこと

Seasarのコンテナinit処理

- 実際の案件では、アプリケーション起動時にinit処理でsingletonのコンポーネントを生成した方が効率的です
  - Springにはこのような機能はありません
- init処理を含めた場合のコンテナ生成でのパフォーマンスを見てみましょう
  - Seasar: コンテナ生成 + init
  - Spring: コンテナ生成

Seasarのコンテナinit処理

- コンテナ生成(+ init処理)



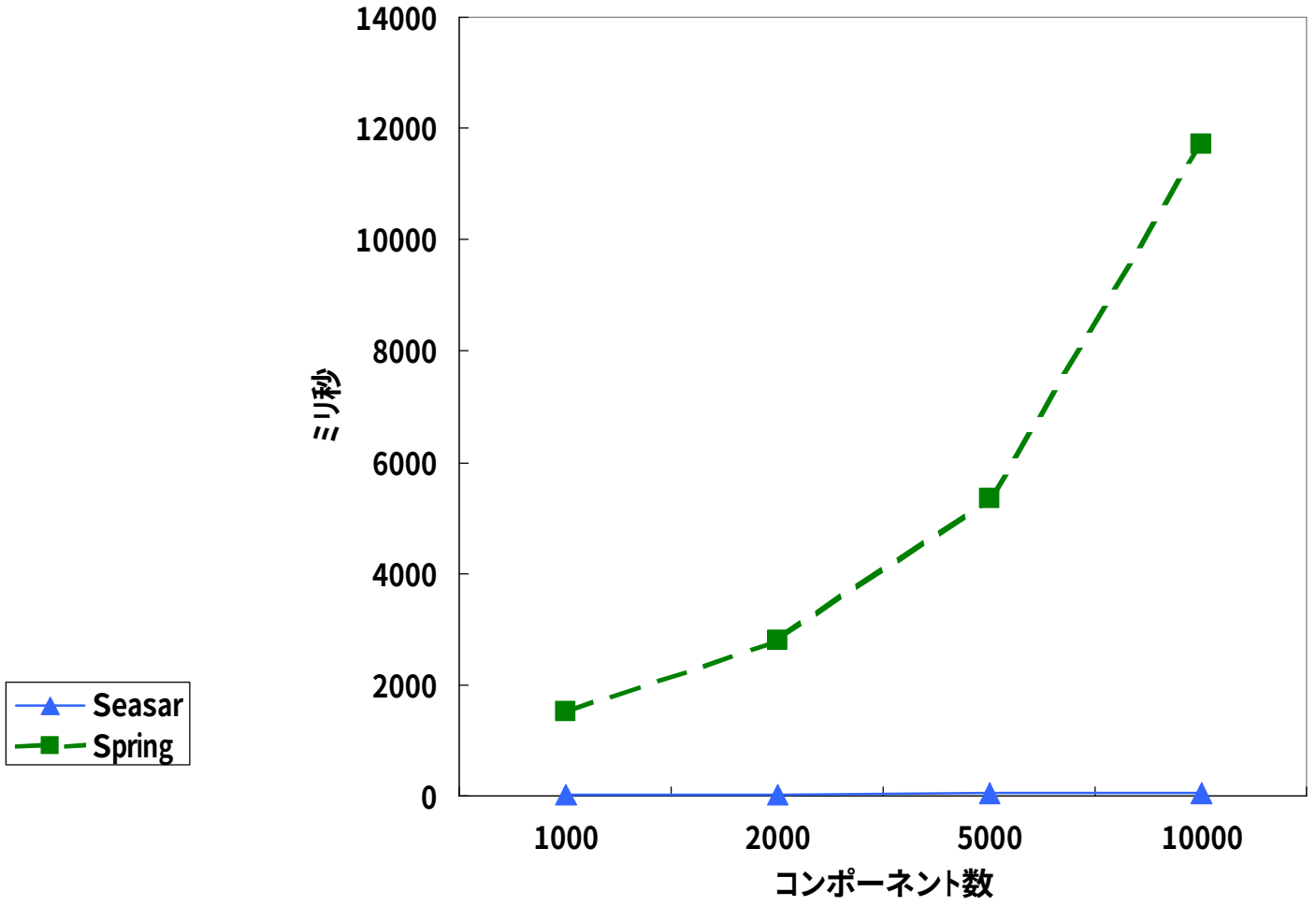
Seasarのコンテナinit処理:結果

- Seasar　　Spring

- **理由**
  - **init処理ではsingletonのオブジェクトを生成しているだけなので、それほど時間が掛かりません**
  - **コンテナ作成時の速度はSeasarの方が速いため、initでのオーバーヘッドをカバーできます**

- では…
- create + initした場合での、コンポーネント取得パフォーマンスは?

- create + initした後のコンポーネント取得処理



結果

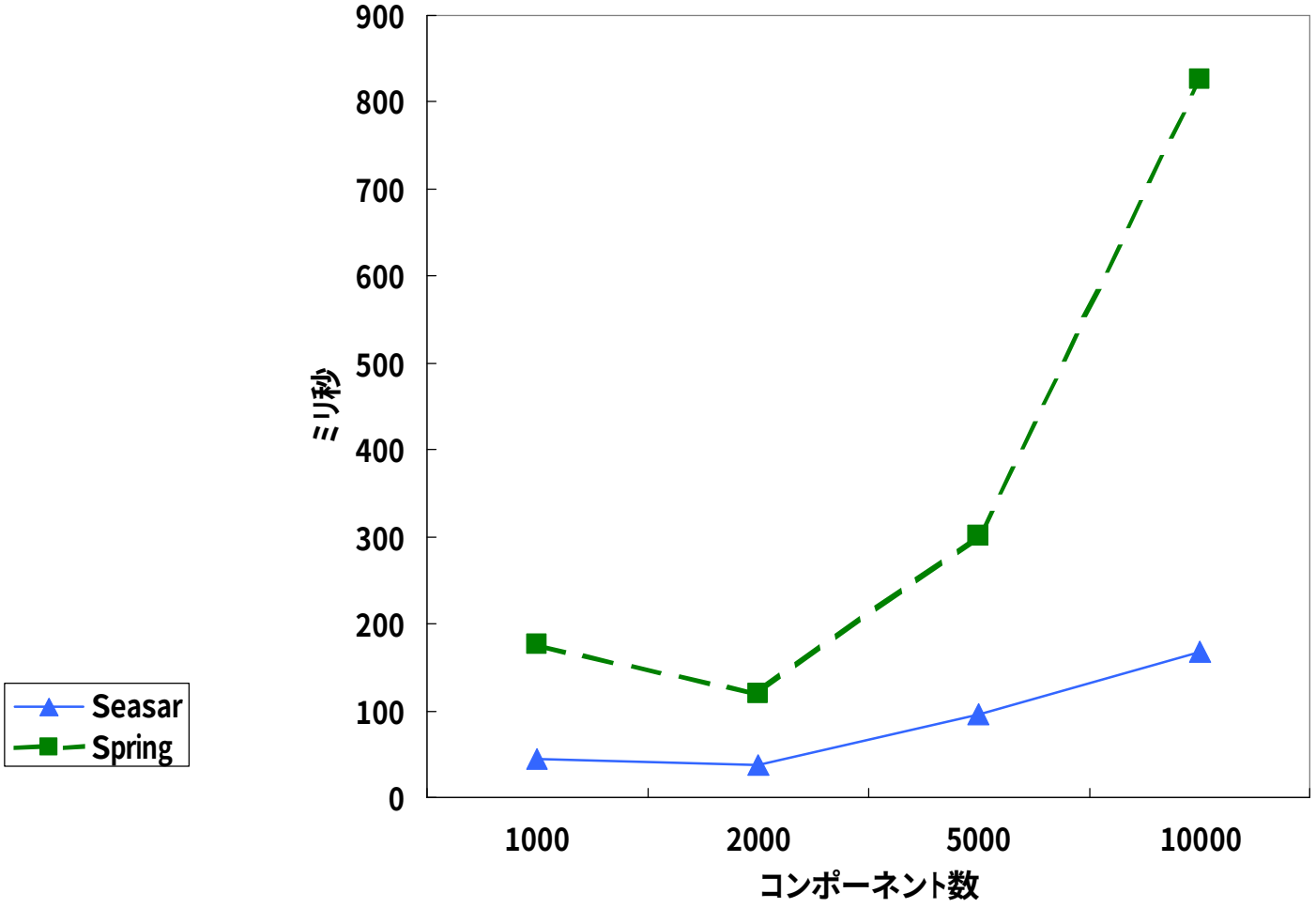
- Seasar ＞＞＞＞＞＞＞ (60〜200倍) ＞＞＞＞＞＞＞＞＞＞＞＞ Spring
  - 1000個で1500ms

- **実際の案件ではアプリケーション初期化時にcreate + init処理を行っているので、これが現実のプロジェクトで起こる結果を反映しています**
  - **ただし、コンテナから2回目に取り出すときは、SeasarもSpringもキャッシュしたオブジェクトを返すだけなので、差は付きません**

prototype

- 今まではsingletonの場合でした。今度はprototypeの場合を見てみましょう
  - prototypeとは、コンポーネントを取得するたびに新たに生成することです
- prototypeでも1度目の取得はsingletonと同様に圧倒的な差が出ます
- 2度目の取得で比べてみましょう

- prototypeで2度目のコンポーネント取得



prototype:結果

- Seasar ＞(3〜5倍)＞ Spring
  - 1000個で130ms

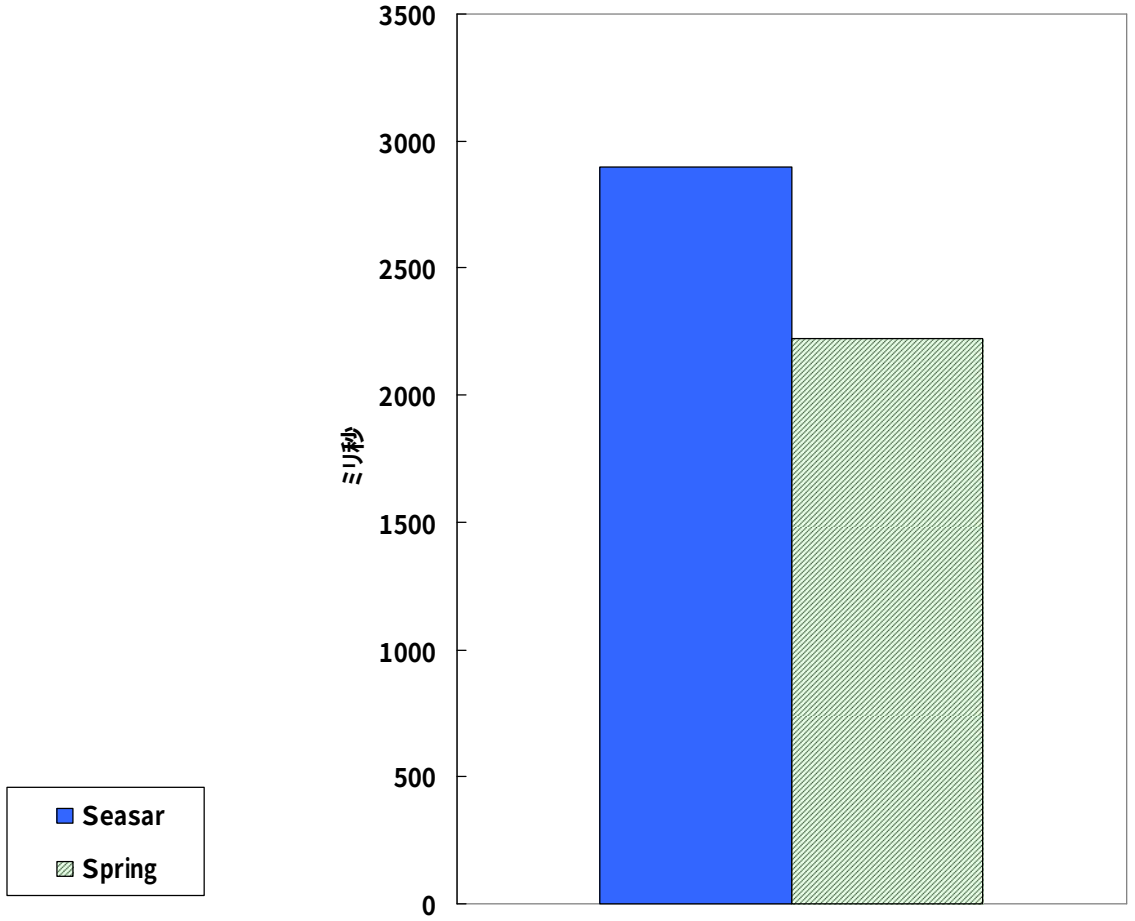
- **理由**
  - Springでは対象となるオブジェクトに加えてBeanWrapperImplを毎回作っていますが、Seasarでは対象となるオブジェクトしか作りません。これが原因の1つかもしれません。

DI (Manual)

- 次はDI処理について見てみましょう
  - DIとは、あるコンポーネントが必要とする他のコンポーネントを、コンテナがセットしてあげることです(ざっくり)

- 現実的な状況を反映させるため、最初にコンテナ生成とinitを実行した上で比較しています
  - コンテナへコンポーネントを2000個登録しています。2個で1組です。

- **コンテナ生成** (Seasarはinitを含む)



DI (Manual):結果

- Seasar ＜ Spring
  - **差は**600ms

- **前回コンテナ生成を比較した場合はほぼ一緒でしたが…**

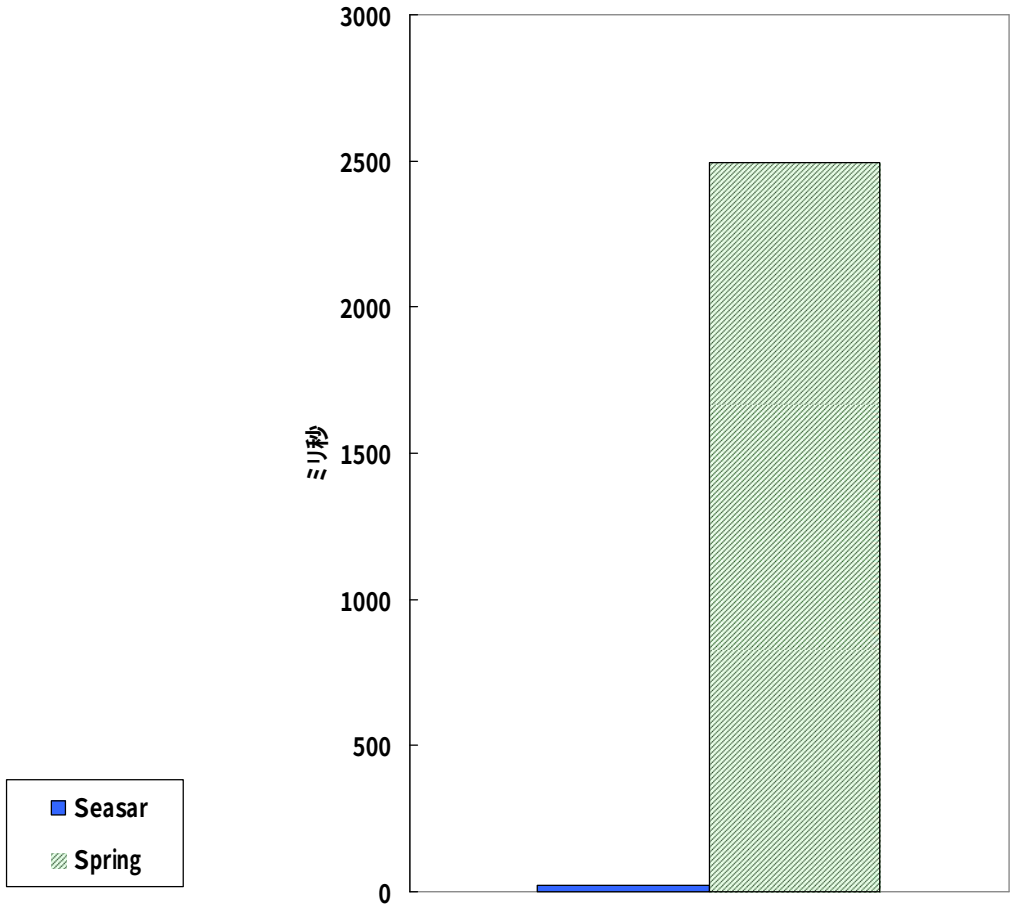
DI (Manual):結果

- 理由
  - 今回2000個のコンポーネントでコンテナ生成した場合は600ms差が出ています
  - この差はリフレクションキャッシュによるものです
  - 前回より1000個余分にキャッシュしていることが今回の600msの差につながっています
    - Seasarでリフレクションキャッシュ1000個と2000個を作成する時間の差が400msでしたので、若干違いますがほぼその差と思われます
- 差が大きくないのと、初期化時の処理であることを考えると、現実にはあまり問題にならないと思います

DI (Manual)

- 今度は実際にユーザに影響する部分である、DIしたコンポーネントを取得する処理を見てみましょう

- DIしたコンポーネントを取得(1000個)
  - Manual DI
  - singleton



DI (Manual):結果

- Seasar ＞＞＞＞＞＞ (100倍) ＞＞＞＞＞＞＞＞＞＞＞＞ Spring
  - 1000セットで2400ms

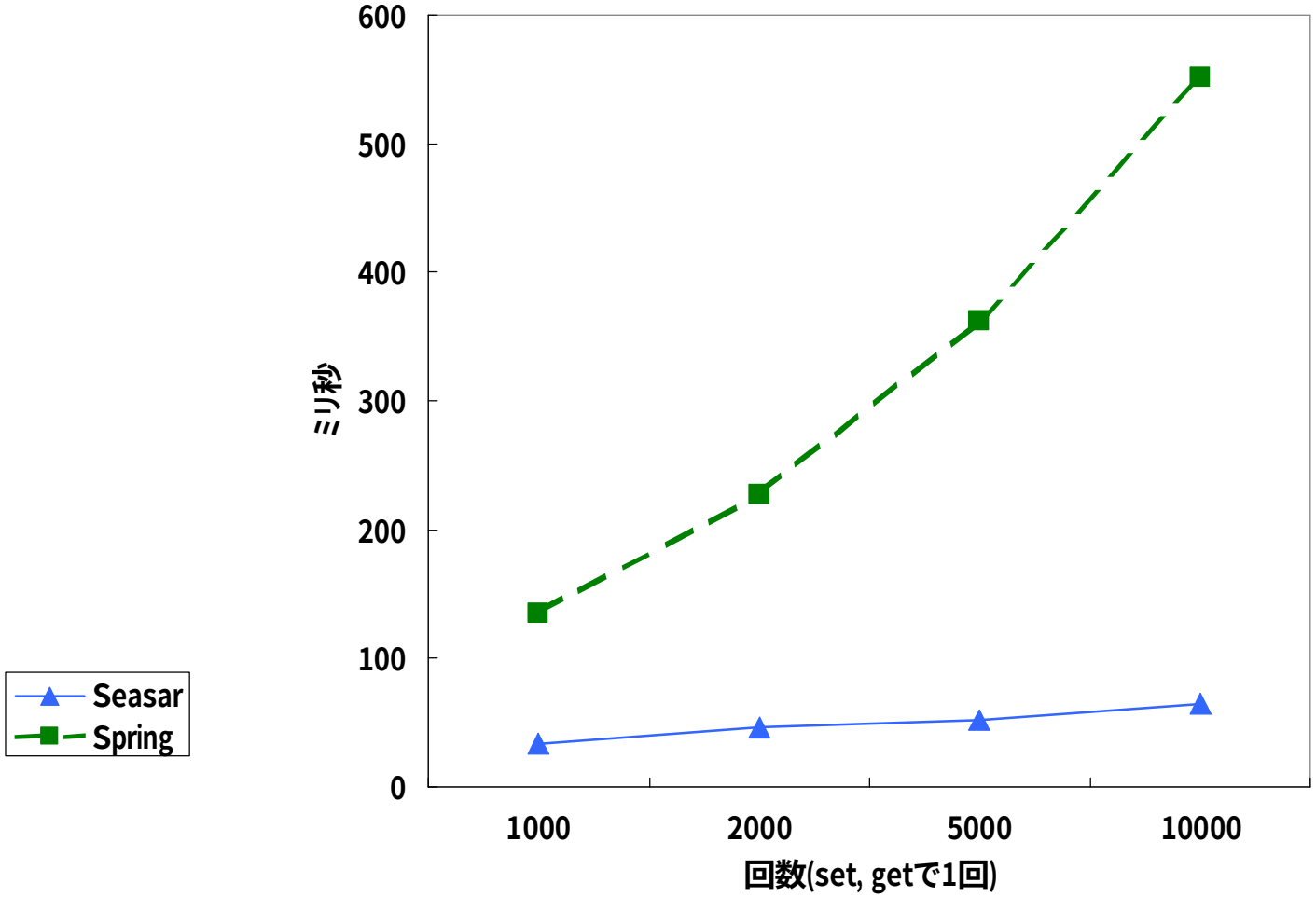
- DI無しの場合と比べると…
  - 今回は2000個から1000個取り出していますが、1000個から1000個取り出すのと速度は同じですので、そのときと比べてみましょう
- DI無しの場合は60倍、今回は100倍。DIすることによってさらに差が開いています。

DI (Manual):結果

- 理由
  - DIするときに、プロパティに対してリフレクションでアクセスしています
  - リフレクションを行うクラスの性能差が一因と思われます
- リフレクションでのアクセスがどれくらいか見てみましょう
  - 1プロパティへset, getして測定しました
    - Seasar: BeanDescImpl
    - Spring: BeanWrapperImpl

リフレクション

- リフレクションでのプロパティアクセス



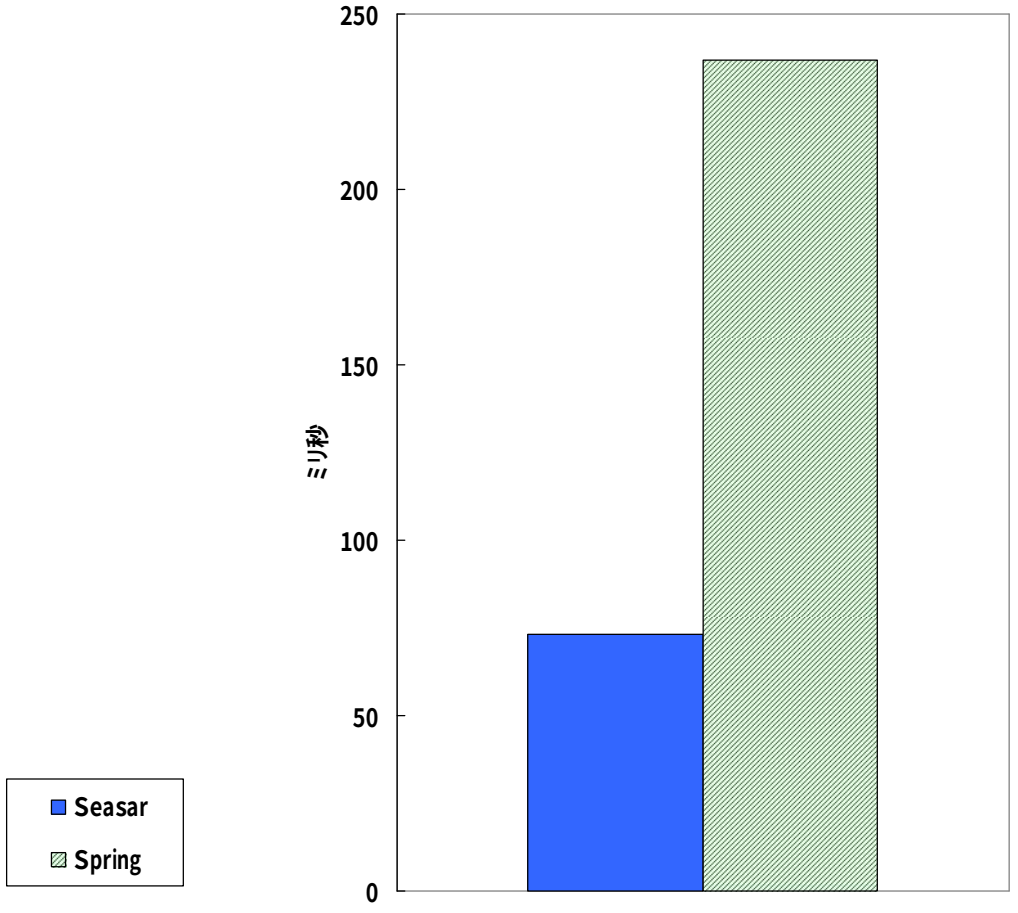
リフレクション:結果

- Seasar ＞ (4〜8倍) ＞ Spring
  - 1000回で100ms

- **理由**
  - BeanDescImplとBeanWrapperImplの差と思われます
    - BeanWrapperImplではネストしたプロパティをサポートしており、それ関連のオーバーヘッド(文字列操作とか)が大きいと思われます

- 次は、prototypeで明示的にDIを指定した場合の、2度目のアクセスについてです

- DIしたコンポーネントを取得(1000個)
  - Manual DI
  - prototype



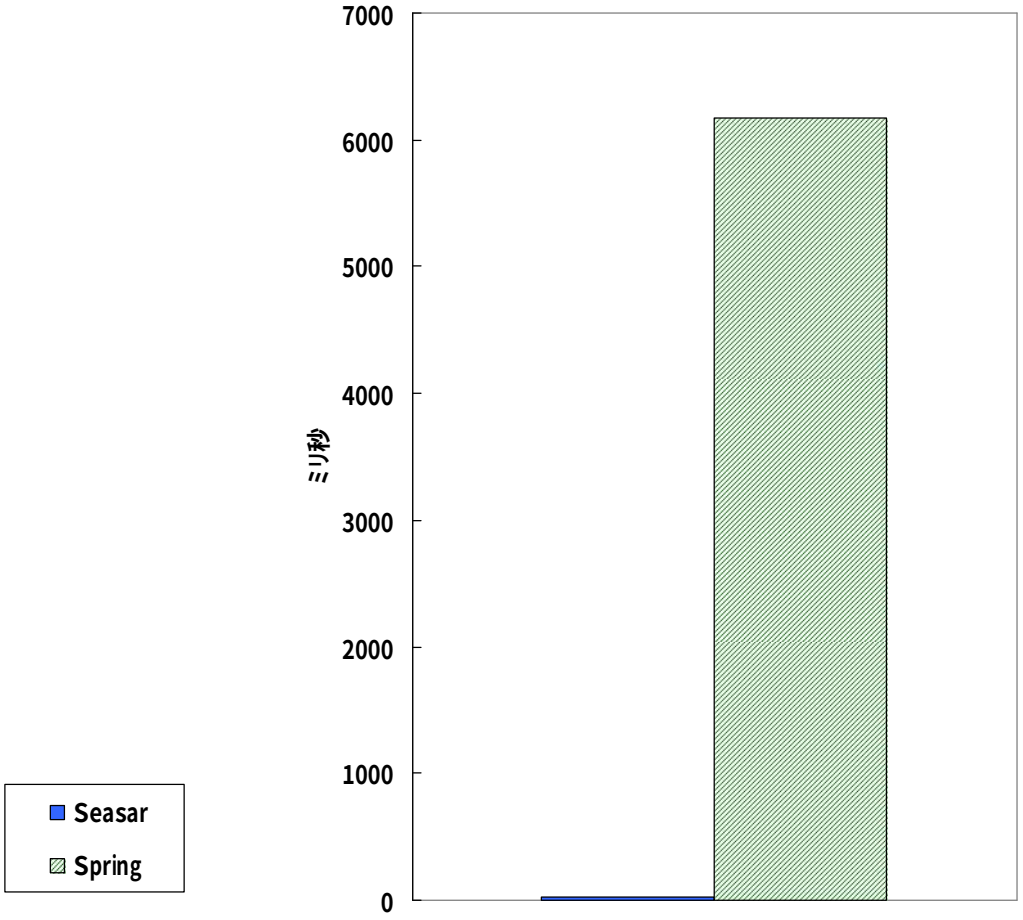
結果

- Seasar ＞(3倍)＞ Spring
  - 1000セットで150ms

- **DIしない場合でもprototypeでの2度目の取得は3〜5倍の差だったので、DI処理のぶん更に差が出ると思いましたが、想定したほどではありませんでした**

- 設定ファイルを少しでも少なくするために、autowireというものがあります
  - 設定ファイルにDIを指定するpropertyタグを書かなくて良くなります
  - autowireには幾つか種類がありますが、ここでは型によるDIを使用しています

- DIしたコンポーネントを取得(1000個)
  - autowire byType
  - singleton



autowire:結果

- Seasar ＞＞＞＞＞＞＞＞＞＞＞＞＞＞＞＞＞＞＞＞＞＞＞＞＞(300倍)＞＞＞＞＞＞＞＞＞＞＞ Spring
  - 1000セットで6000ms

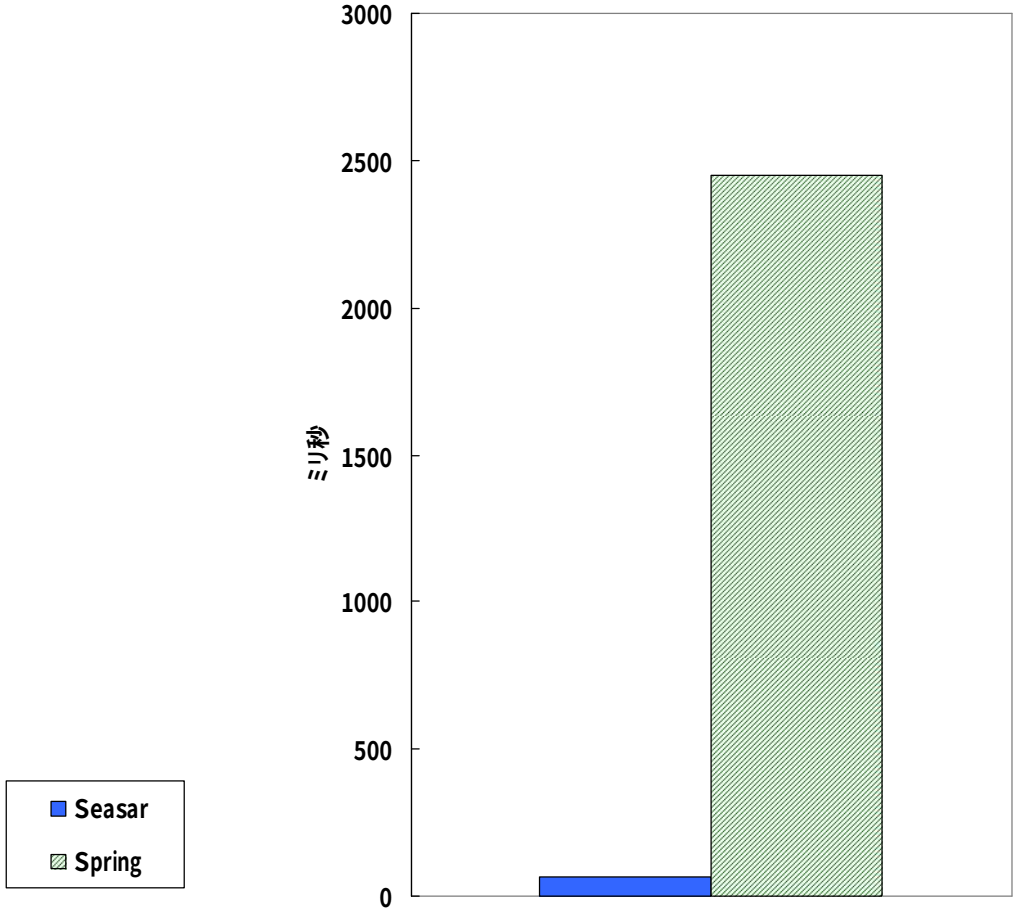
- Manualでは100倍の差でしたが、Autoにすると更に3倍の差が付きました

autowire:結果

- 理由
  - autowire時にはDI対象を探すロジックが実行されます
    - SpringではDIの度に、毎回コンテナへ登録されている全てのオブジェクトへアクセスします
      - コンテナには2000個登録されていて、1000回DIしているので、2000 * 1000回コンポーネント定義へアクセスしています。
    - Seasarはコンポーネントを登録するときにクラスの型をキー情報としてハッシュテーブルへ登録しているので、DIの度に1回のアクセスで済みます
  - つまりDIの度にListへ全件アクセスするのかHashMapへキーでアクセスするのかの差なので、差が付いて当たり前と言えるでしょう

- autowireでprototypeの場合はどうでしょうか?
  - 2回目のコンポーネント取得時

- DIしたコンポーネントを取得(1000個)
  - autowire byType
  - prototype



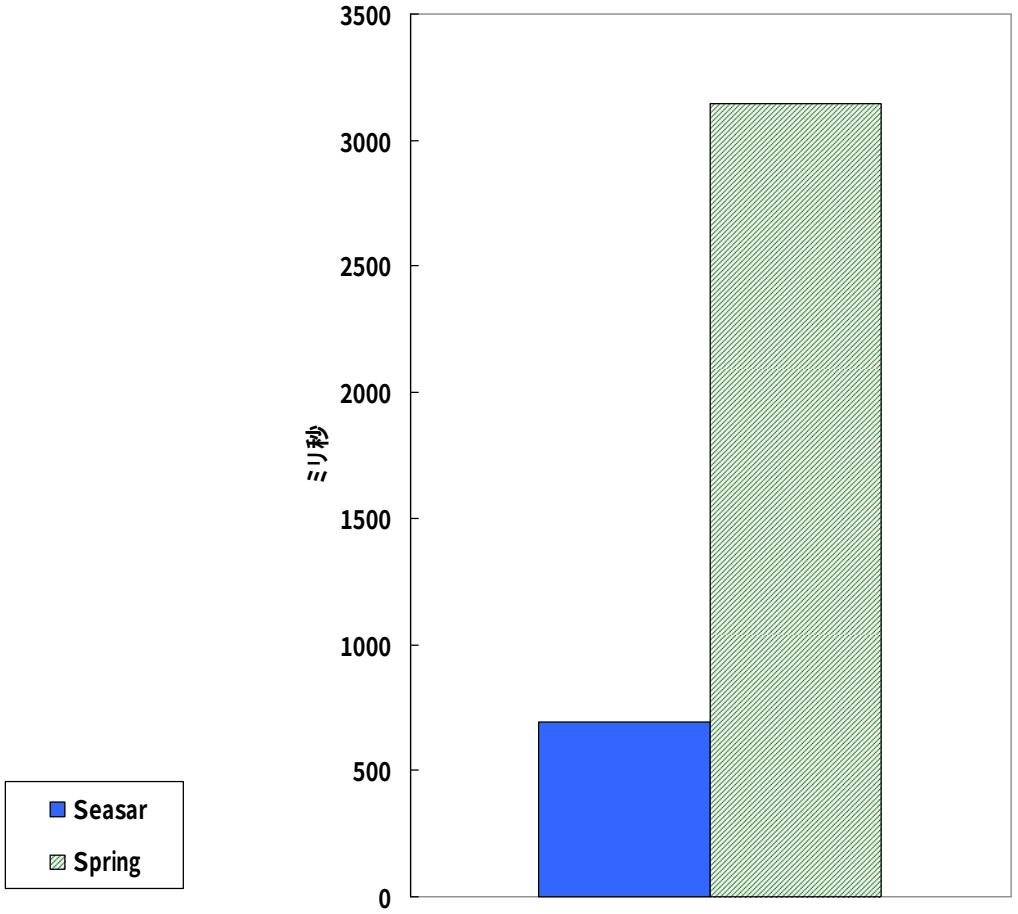
結果

- Seasar ＞＞＞＞ (35倍) ＞＞＞＞＞ Spring
  - 1000セットで2300ms

- **理由**
  - singletonと同じで、DI対象を探すロジックの差でしょう
- singletonほどではありませんが、大きな差が出ました

- AOPとは、バイトコードを操作し もともとの処理をカスタマイズするもの (ざっくり)
- AOPを掛けたメソッドを実行して、速度差を見てみましょう
  - 今回のAOPは文字列を返すだけの、非常にシンプルなものです。だからこそAOPのオーバーヘッドがわかりやすいと思います
  - 10,000,000回メソッドを実行
    - SeasarはJavassist
    - SpringはCGLIB (DynamicProxyよりも速い)

- AOPを仕掛けたメソッドを実行



AOP:結果

- Seasar ＞(3〜4倍)＞ Spring
  - 10,000,000回で2400ms

- **理由**
  - Seasarは2.1まではCGLIBで2.2からはJavassistに変えて、約3倍速くなったことがあります
  - CGLIBを使うと殆どチューニングの余地がありませんが、Javassistにはチューニングの余地があります
  - Seasarではかなりのチューニングを行っているので、速くなっていると思われます

AOP weaving

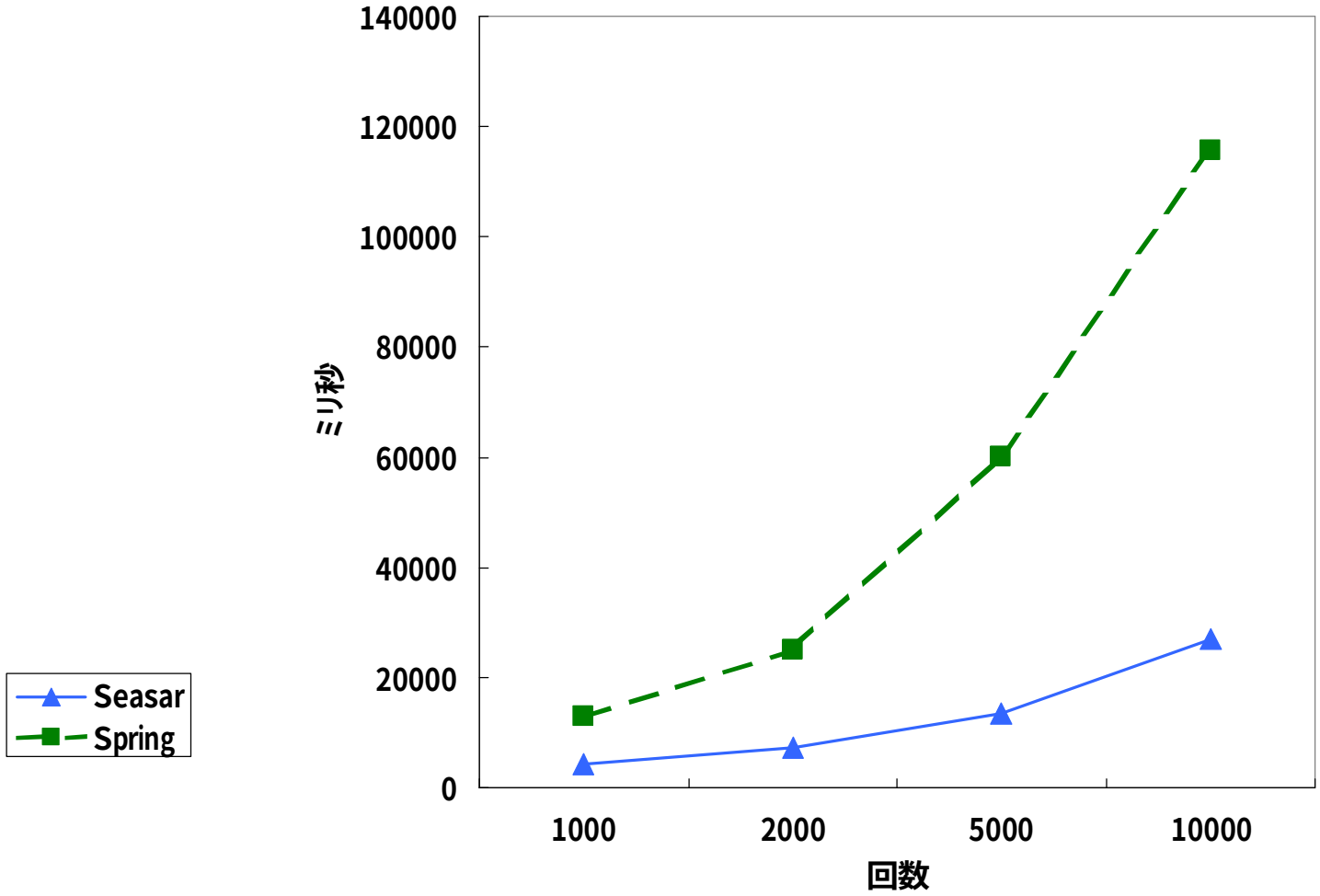
- AOPを組み込むバイトコード操作を、weavingと呼んでいます
- このweavingもパフォーマンスに与える影響があると考えたため、測定してみました

AOP weaving

- **まずは、weavingするクラスを直接呼び出して、速度差を比較しました**
  - Seasar: AopProxy
  - Spring: ProxyFactory

AOP weaving

- AOPのWeaving



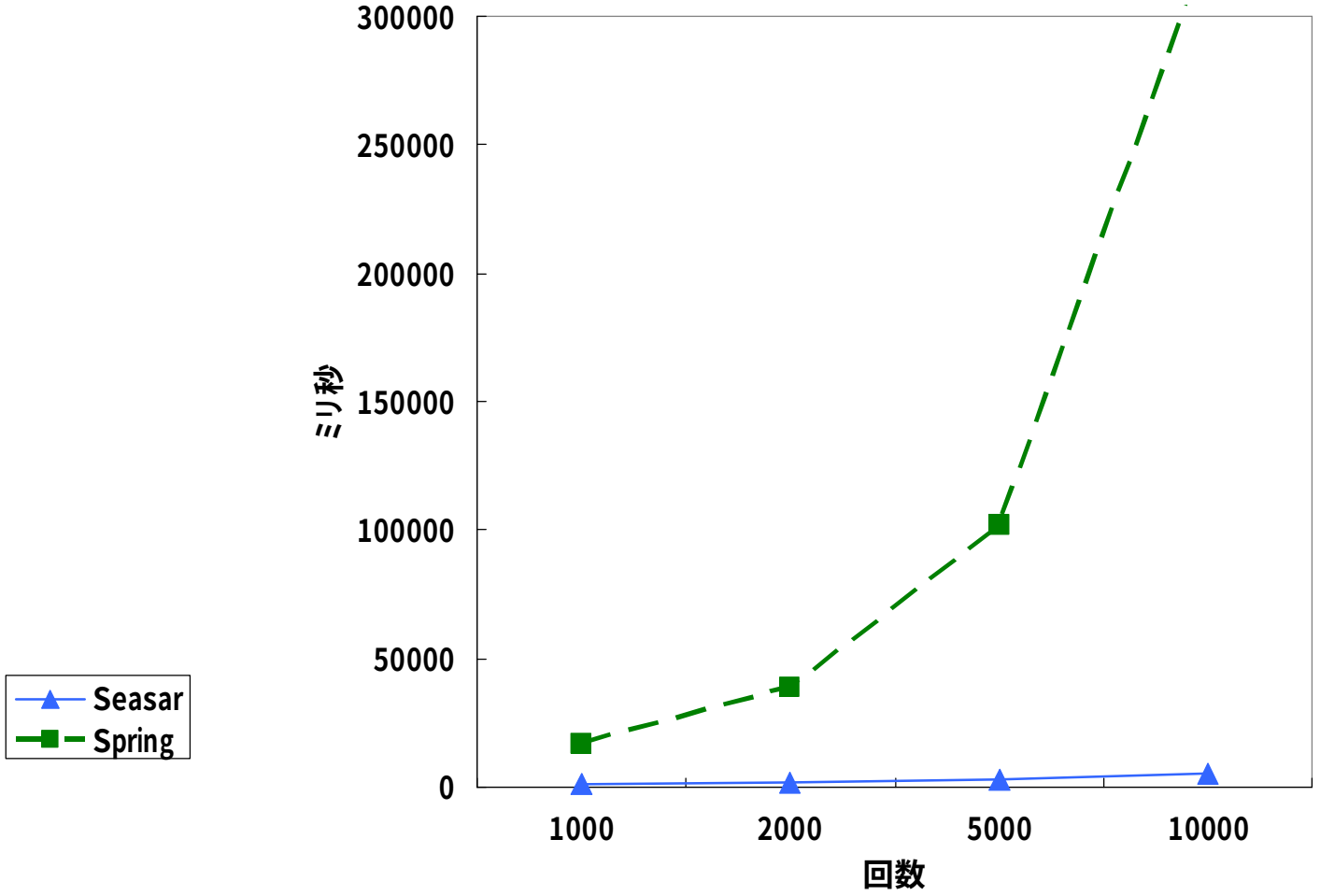
AOP weaving:結果

- Seasar ＞(3倍)＞ Spring
  - 1000回で8000ms

- **理由**
  - JavassistとCGLIBでのバイトコードweavingの速度差と思われます
- AOPのweavingにかかる絶対時間が大きいことがわかります (1000個で8秒!)

- 次は、登録されているコンポーネントへまとめてAspectを仕掛けて、コンテナを生成してみます
- まとめてAspectを仕掛ける機能
  - Seasar: AspectAutoRegister
  - Spring: AutoProxyCreator
- これらを使ってみました

- AOP自動登録でのコンテナ生成



結果

- Seasar ＞＞＞ (15～60倍) ＞ ＞ ＞ Spring
  - 1000個で15000ms

- **理由**
  - **リフレクション情報のキャッシュ**
  - AOP weaving
- **やはり、AOP weavingはDIコンテナの処理の中では重い部類に入ることがわかります**

- 補足情報
  - Springは(今回使用した方法で)AOPを登録すると、コンテナ生成時にリフレクション情報をキャッシュしコンポーネントを生成するようです
    - 1度目のコンポーネント取得時に発生していた負荷がコンテナ生成時に寄っています
    - そのぶん、コンポーネント取得時の速度はSeasarと同じくらいに速くなっています

まとめ

- DIという同じ技術を実装してこれほどの差が出るのはかなり驚きです
- ある程度、原因も指摘しているので、この結果を元にSpringのチューニングに役立ててもらえれば幸いです
- この結果およびテストプログラムはオープンソースとして公開する予定です

本日はご静聴いただき
ありがとうございました。